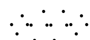
Dernocua

Kragen Javier Sitaker
Buenos Aires
December, 02021
Public domain work



Dernocua is my notebook from 02021 CE, the second year of the covid pandemic. It's the sequel to Dercuano and Derctuo. It contains about 410000 words, about 1100 pages, in 267 notes on various different topics: programming, machining, electronics, digital fabrication, math, physics, economics, history, and so on. These are mostly notes I made while I was figuring things out, so they have a lot of errors in them because of things I understood incorrectly, and many of them are incomplete.

Dernocua's canonical form is a downloadable tarball of HTML files . For computers that can't handle tarballs of HTML, I've also hacked together an inferior PDF rendering of it, formatted for comfortable reading on small screens like those on hand computers. (In particular, a lot of the Unicode-art stuff is screwed up in the PDF, and so are tables.) Errata will be published at the GitLab repository.

Public-domain dedication

As far as I'm concerned, everyone is free to redistribute Dernocua, in whole or in part, modified or unmodified, with or without credit; I waive all rights associated with it to the maximum extent possible under applicable law. Where applicable, I abandon its copyright to the public domain. I wrote and published Dernocua in Argentina in 02021 (more conventionally called 2021, or 2021 AD).

The exception to the above public-domain dedication is the ET Book font family used, licensed under the X11 license (p. 18). This doesn't impede you from redistributing or modifying Dernocua but does prohibit you from removing the font's copyright notice and license (unless you also remove the font). The PDF embeds part of FreeFont and of the DejaVu fonts, whose copyright notices are also included (p. 19), but DejaVu and FreeFont are not used in the HTML tarball.

Notes

02021-01

• Fan noise would be less annoying if intermittent (p. 21) 02021-01-03 (updated 02021-01-04) (1 minute)
• Principled APL redux (p. 22) 02021-01-03 (updated 02021-12-31) (12 minutes)

• First class locations (p. 27) 02021-01-04 (3 minutes)
• Layout typescript (p. 29) 02021-01-04 (5 minutes)
• Fibonacci scan (p. 31) 02021-01-10 (updated 02021-01-15) (1 minute)
• Relayout with heaps (p. 32) 02021-01-10 (updated 02021-01-15) (6 minutes)
• Transactional editor (p. 35) 02021-01-14 (updated 02021-01-15) (73 minutes)
• Trie PEGs (p. 53) 02021-01-15 (4 minutes)
• Chat over a content-centric network (p. 55) 02021-01-15 (updated 02021-01-16) (3 minutes)
• Some notes on compiling and notations for grammars, starting from the inspiring RPN example in Parson (p. 57) 02021-01-15 (updated 02021-12-31) (15 minutes)
• Can transactions solve the N+1 performance problem on web pages? (p. 67) 02021-01-16 (8 minutes)
• Notes on simulating a ZVS converter (Baxandall converter) (p. 70) 02021-01-16 (6 minutes)
• A ghetto linear voltage regulator from discrete components (p. 73) 02021-01-21 (updated 02021-01-27) (10 minutes)
• Intel engineering positions considered as a dollar auction (p. 78) 02021-01-21 (updated 02021-01-27) (1 minute)
• Duplicating Durham's Rock-Hard Putty (p. 79) 02021-01-22 (updated 02021-01-27) (1 minute)
• iPhone replacement cameras as 6-μs streak cameras (p. 80) 02021-01-22 (updated 02021-12-30) (2 minutes)
• Compiling machine-code loops to pipelined dataflow graphs (p. 81) 02021-01-23 (updated 02021-01-27) (2 minutes)
• Some preliminary notes on the amazing RISC-V architecture (p. 82) 02021-01-24 (updated 02021-07-27) (29 minutes)
• Trying to design a simple switchmode power supply using Schmitt-trigger relaxation oscillators (p. 92) 02021-01-26 (updated 02021-01-27) (32 minutes)
• Trying and failing to design an efficient index for folksonomy data based on BDDs (p. 108) 02021-01-26 (updated 02021-01-27) (7 minutes)

02021-02

• The use of silver in solar cells (p. 112) 02021-02-02 (updated 02021-09-11) (8 minutes)
• Snap logic, revisited, and four-phase logic (p. 115) 02021-02-08 (9 minutes)
• Can you do direct digital synthesis (DDS) at over a gigahertz? (p. 119) 02021-02-08 (updated 02021-02-24) (30 minutes)
• ASCII art, but in Unicode, with Braille and other alternatives (p. 128) 02021-02-10 (updated 02021-02-24) (9 minutes)
• Skew tilesets (p. 135) 02021-02-14 (updated 02021-02-24) (7 minutes)
• Threechet (p. 140) 02021-02-16 (updated 02021-02-24) (4 minutes)

• Thumbnail views in a Unicode character-cell terminal with Braille (p. 142) 02021-02-17 (updated 02021-02-24) (1 minute)
• Energy autonomous computing (p. 143) 02021-02-18 (updated

02021-12-30) (58 minutes)
• How do you fit a high-level language into a microcontroller? Let's look at BBN Lisp (p. 160) 02021-02-23 (updated 02021-08-18) (76 minutes)
• Panelization in PCB manufacturing (p. 193) 02021-02-25 (updated 02021-02-26) (7 minutes)

02021-03

• Some notes on IPL-VI, Lisp's 01958 precursor (p. 196) 02021-03-02 (4 minutes)
• Refreshing Flash memory periodically for archival (p. 198) 02021-03-02 (1 minute)
• Variable length unaligned bytecode (p. 199) 02021-03-02 (updated 02021-03-03) (4 minutes)
• A survey of imperative programming operations' prevalence (p. 201) 02021-03-02 (updated 02021-09-11) (61 minutes)
• Vaughan Pratt and Henry Baker's COMFY control-flow combinators (p. 234) 02021-03-04 (updated 02021-03-20) (8 minutes)
• Generating novel unique pronounceable identifiers with letter frequency data (p. 239) 02021-03-10 (updated 02021-03-22) (11 minutes)
• Garbage-collected allocation performance on current computers (p. 245) 02021-03-13 (updated 02021-04-08) (4 minutes)
• How Lao Yuxi painted a cock (p. 247) 02021-03-19 (updated 02021-04-14) (7 minutes)
• Bench supply (p. 250) 02021-03-19 (updated 02021-12-30) (25 minutes)
• Recursive residue number systems? (p. 259) 02021-03-20 (updated 02021-03-22) (8 minutes)
• Brute force speech (p. 262) 02021-03-21 (updated 02021-03-22) (7 minutes)
• When is it better to compute by moving atoms rather than electrons? (p. 265) 02021-03-21 (updated 02021-03-22) (5 minutes)
• Veskeno is a "fantasy platform" like TIC-80 (p. 267) 02021-03-21 (updated 02021-03-22) (3 minutes)
• Some notes on reading Chris Seaton's TruffleRuby dissertation (p. 269) 02021-03-21 (updated 02021-03-22) (16 minutes)
• .xosm: experimental obvious stack machine (p. 274) 02021-03-21 (updated 02021-03-24) (20 minutes)
• Open coded primitives (p. 283) 02021-03-22 (26 minutes)
• Failing to stabilize the amplitude of an opamp phase-delay oscillator (p. 298) 02021-03-23 (updated 02021-03-24) (10 minutes)
• Running scripts once per frame for guaranteed GUI responsivity (p. 303) 02021-03-23 (updated 02021-10-12) (7 minutes)
• Minor improvements to pattern matching (p. 306) 02021-03-24 (updated 02021-04-08) (10 minutes)
• Why Bitcoin is puzzling to people in rich countries (p. 312) 02021-03-31 (updated 02021-07-27) (10 minutes)

02021-04

• Statistics on the present and future of energy in the People's

Republic of China (p. 316) 02021-04-01 (updated 02021-04-08) (10 minutes)
• Geneva wheel stopwork (p. 321) 02021-04-07 (updated 02021-04-08) (6 minutes)
• A bargain-basement Holter monitor with a BOM under US$2.50 (p. 323) 02021-04-07 (updated 02021-07-27) (33 minutes)
• Locking telescope (p. 333) 02021-04-07 (updated 02021-12-30) (2 minutes)
• Logarithmic low-power SERDES (p. 334) 02021-04-08 (4 minutes)

• :fqozl, a normal-order text macro language (p. 336) 02021-04-09 (updated 02021-07-27) (14 minutes)
• Forming steel with copper instead of vice versa (p. 344) 02021-04-16 (updated 02021-06-12) (2 minutes)
• Notes on pricing of locally available oscilloscopes (p. 346) 02021-04-16 (updated 02021-07-27) (2 minutes)
• Can you get JLCPCB to fabricate a CPU for you affordably from “basic” parts? (p. 347) 02021-04-17 (updated 02021-12-30) (9 minutes)
• Safe FORTH with the FORTRAN memory model? (p. 351) 02021-04-21 (updated 02021-06-12) (2 minutes)
• Manually writing code in static single assignment (SSA) form, inspired by Kemeny’s DOPE, isn’t worth it (p. 353) 02021-04-21 (updated 02021-06-12) (3 minutes)
• Diskstrings: Bernstein’s netstrings for single-pass streaming output (p. 356) 02021-04-21 (updated 02021-07-27) (4 minutes)
• Phased-array imaging sonar from a mesh network of self-localizing sensor nodes (p. 358) 02021-04-27 (updated 02021-12-30) (8 minutes)

• A boiler for submillisecond steam pulses (p. 361) 02021-04-28 (updated 02021-12-30) (10 minutes)
• Three phase logic (p. 364) 02021-04-30 (updated 02021-07-27) (9 minutes)

02021-05

• How fast do von Neumann probes need to reproduce to colonize space in our lifetimes? (p. 368) 02021-05-04 (updated 02021-06-12) (5 minutes)
• List of random GUI ideas (p. 370) 02021-05-04 (updated 02021-07-27) (6 minutes)
• Thixotropic electrodeposition (p. 372) 02021-05-04 (updated 02021-12-31) (2 minutes)
• Cheap cutting jig (p. 373) 02021-05-06 (updated 02021-12-30) (1 minute)
• Differential filming (p. 374) 02021-05-07 (updated 02021-12-30) (1 minute)
• Planetary roller screw worm drive (p. 375) 02021-05-07 (updated 02021-12-30) (4 minutes)
• Fresnel mirror electropolishing (p. 377) 02021-05-08 (updated 02021-12-30) (6 minutes)
• Leaf hypertext (p. 380) 02021-05-08 (updated 02021-12-30) (3 minutes)
• Weighing an eyelash on an improvised Kibble balance (p. 382)

02021-05-08 (updated 02021-12-30) (3 minutes)
• Precisely measuring out particulates with a trickler (p. 384) 02021-05-09 (updated 02021-12-30) (17 minutes)
• A four-dimensional keyboard matrix made of linear voltage differential transformers (LVDTs) to get 30 or 180 keys on five pins (p. 390) 02021-05-12 (updated 02021-12-30) (4 minutes)
• Planetary screw potentiometer (p. 392) 02021-05-12 (updated 02021-12-30) (1 minute)
• 3-D printing in carbohydrates (p. 393) 02021-05-16 (updated 02021-12-30) (10 minutes)
• Clay-filled PLA filament for firing to ceramic (p. 396) 02021-05-17 (updated 02021-12-30) (1 minute)
• Multicolor filament (p. 397) 02021-05-17 (updated 02021-12-30) (5 minutes)
• Acicular low binder pastes (p. 399) 02021-05-19 (updated 02021-12-30) (1 minute)
• Cutting clay (p. 400) 02021-05-19 (updated 02021-12-30) (10 minutes)
• Scaling laws (p. 404) 02021-05-19 (updated 02021-12-30) (8 minutes)
• Selectively curing one-component silicone by injecting water (p. 408) 02021-05-19 (updated 02021-12-30) (2 minutes)
• Clay wire cutter (p. 409) 02021-05-21 (updated 02021-12-30) (2 minutes)
• Electroforming rivets (p. 410) 02021-05-22 (updated 02021-12-30) (2 minutes)
• Metal welding fuel (p. 411) 02021-05-23 (updated 02021-12-30) (6 minutes)
• Aluminum foil (p. 413) 02021-05-24 (updated 02021-09-11) (14 minutes)
• The nature of mathematical discourse (p. 418) 02021-05-27 (updated 02021-12-30) (5 minutes)
• Designing curiosity and dreaming into optimizing systems (p. 420) 02021-05-30 (updated 02021-12-30) (6 minutes)
• Omnidirectional wheels (p. 422) 02021-05-30 (updated 02021-12-30) (1 minute)
• Ghetto electrical discharge machining (EDM) (p. 423) 02021-05-31 (updated 02021-12-30) (5 minutes)

02021-06

• Broken hard disks are the cheapest source of ultraprecision components (p. 425) 02021-06-02 (updated 02021-06-12) (3 minutes)

• Micro impact driver (p. 427) 02021-06-02 (updated 02021-06-12) (2 minutes)
• Minkowski deconvolution (p. 428) 02021-06-02 (updated 02021-12-30) (6 minutes)
• Greek operating systems (p. 430) 02021-06-04 (updated 02021-06-12) (4 minutes)
• The algebra of N-ary relations (p. 432) 02021-06-14 (updated 02021-07-27) (4 minutes)
• Nuclear energy is the Amiga of energy sources (p. 434) 02021-06-14 (updated 02021-07-27) (3 minutes)

• Flux-gate downconversion in a loopstick antenna? (p. 436) 02021-06-15 (updated 02021-07-27) (2 minutes)
• PEG-like flexibility for parsing right-to-left? (p. 437) 02021-06-16 (updated 02021-07-27) (2 minutes)
• How little code can a filesystem be? (p. 438) 02021-06-16 (updated 02021-07-27) (1 minute)
• Notes on the PDF file format (p. 439) 02021-06-16 (updated 02021-07-27) (15 minutes)
• Notes on what would be needed to drive a PS/2 keyboard from an Arduino (p. 447) 02021-06-20 (updated 02021-12-30) (12 minutes)
• Self hosting kernel (p. 452) 02021-06-21 (updated 02021-12-30) (1 minute)
• Stack syntax (p. 453) 02021-06-22 (updated 02021-07-27) (4 minutes)
• Bead hypertext (p. 455) 02021-06-22 (updated 02021-12-30) (1 minute)
• Does USB bitstuffing create a timing-channel vulnerability? (p. 456) 02021-06-22 (updated 02021-12-31) (1 minute)
• Verstickulite (p. 457) 02021-06-23 (updated 02021-07-27) (3 minutes)
• Simple linear-time linear-space nested delimiter parsing (p. 459) 02021-06-24 (updated 02021-12-30) (1 minute)
• Economic history (p. 460) 02021-06-25 (updated 02021-07-27) (17 minutes)
• More cements (p. 466) 02021-06-26 (updated 02021-08-15) (5 minutes)
• Base 3 gage blocks (p. 468) 02021-06-27 (updated 02021-12-30) (5 minutes)
• Multiple counter-rotating milling cutters to eliminate side loading (p. 470) 02021-06-27 (updated 02021-12-30) (7 minutes)
• Layered ECM (p. 473) 02021-06-27 (updated 02021-12-30) (2 minutes)
• A kernel you can type commands to (p. 474) 02021-06-27 (updated 02021-12-30) (1 minute)
• Can you use stabilized cubic zirconia as an ECM cathode in molten salt? (p. 475) 02021-06-27 (updated 02021-12-30) (3 minutes)
• Electrolytic glass machining (p. 477) 02021-06-28 (updated 02021-12-30) (6 minutes)
• Super ghetto digital fabrication (p. 479) 02021-06-28 (updated 02021-12-30) (33 minutes)
• Glass powder-bed 3-D printing (p. 490) 02021-06-29 (updated 02021-12-30) (20 minutes)
• Rator-port GUIs (p. 496) 02021-06-29 (updated 02021-12-30) (26 minutes)
• Sulfur jet metal cutting (p. 504) 02021-06-30 (updated 02021-12-30) (6 minutes)
• Powder-bed 3-D printing with a sacrificial binder (p. 506) 02021-06-30 (updated 02021-12-30) (12 minutes)

02021-07

• Stochastically generated self-amalgamating tape variations for composite fabrication (p. 510) 02021-07-02 (updated 02021-12-30) (26 minutes)

• Spin-coating clay-filled plastics to make composites with high anisotropic filler loadings (p. 521) 02021-07-02 (updated 02021-12-30) (4 minutes)
• ECM for machining nonmetals? (p. 523) 02021-07-05 (updated 02021-07-27) (11 minutes)
• Sonic screwdriver resonance (p. 527) 02021-07-06 (updated 02021-12-30) (11 minutes)
• Subnanosecond thermochromic light modulation for real-time holography and displays (p. 531) 02021-07-06 (updated 02021-12-30) (8 minutes)
• Notes on Richards et al.'s nascent catalytic ROS water treatment process (p. 534) 02021-07-07 (updated 02021-07-27) (14 minutes)
• Memory view (p. 539) 02021-07-09 (updated 02021-12-30) (6 minutes)
• Spreadtool (p. 543) 02021-07-10 (updated 02021-12-30) (30 minutes)
• Smolsay: the Ur-Lisp, but with dicts instead of conses (p. 552) 02021-07-12 (updated 02021-07-27) (22 minutes)
• Electrolytic berlinite (p. 561) 02021-07-12 (updated 02021-12-30) (7 minutes)
• Making mirabilite and calcite from drywall (p. 564) 02021-07-12 (updated 02021-12-30) (4 minutes)
• Potential local sources and prices of refractory materials (p. 566) 02021-07-14 (updated 02021-09-11) (9 minutes)
• Faygoo: a yantra-smashing ersatz version of Piumarta and Warth's COLA (p. 570) 02021-07-14 (updated 02021-12-30) (17 minutes)
• Firing talc (p. 576) 02021-07-14 (updated 02021-12-30) (17 minutes)
• Improvements on C for low-level programming such as block arguments (p. 584) 02021-07-14 (updated 02021-12-30) (8 minutes)
• Fiberglass CMCs? (p. 588) 02021-07-15 (updated 02021-07-27) (8 minutes)
• My ideal workshop (unfinished) (p. 591) 02021-07-16 (updated 02021-07-27) (2 minutes)
• Can you 3-D print Sorel cement by inhibiting setting with X-rays? (p. 592) 02021-07-16 (updated 02021-07-27) (1 minute)
• Tetrahedral expanded metal (p. 593) 02021-07-16 (updated 02021-07-27) (3 minutes)
• Glass foam (p. 595) 02021-07-16 (updated 02021-08-15) (17 minutes)
• Leaf vein roof (p. 600) 02021-07-16 (updated 02021-09-11) (9 minutes)
• Aluminum fuel (p. 603) 02021-07-17 (updated 02021-12-30) (2 minutes)
• Boosters for self-propagating high-temperature synthesis (SHS) (p. 604) 02021-07-17 (updated 02021-12-30) (4 minutes)
• Compressed appendable file (p. 606) 02021-07-19 (updated 02021-07-27) (5 minutes)
• SHS of magnesium phosphate (p. 608) 02021-07-22 (updated 02021-07-27) (3 minutes)
• Back-drivable differential windlass (p. 610) 02021-07-23 (updated 02021-07-27) (15 minutes)
• Synthesizing reactive magnesia? (p. 615) 02021-07-25 (updated 02021-08-15) (4 minutes)

• Synthesizing amorphous magnesium silicate (p. 617) 02021-07-25 (updated 02021-08-15) (6 minutes)
• Ropes with constant-time concatenation and equality comparisons with monoidal hash consing (p. 619) 02021-07-27 (15 minutes)
• Compilation of block arguments to high-performance code (p. 624) 02021-07-29 (updated 02021-12-30) (19 minutes)
• Material observations (p. 633) 02021-07-29 (updated 02021-12-30) (194 minutes)

02021-08

• Dipropylene glycol (p. 687) 02021-08-01 (updated 02021-08-15) (2 minutes)
• The ayurvedic "fire mud" of Bhudeb Mookerji and modern castable refractories (p. 688) 02021-08-05 (updated 02021-08-15) (22 minutes)

• Arc maker (p. 695) 02021-08-07 (updated 02021-12-30) (11 minutes)
• Argentine pricing of PEX pipe and alternatives for phase-change fluids (p. 699) 02021-08-07 (updated 02021-12-30) (2 minutes)
• Power transistors (p. 700) 02021-08-07 (updated 02021-12-30) (12 minutes)
• Pocket kiln (p. 704) 02021-08-09 (updated 02021-08-15) (7 minutes)
• Cola flavor (p. 707) 02021-08-10 (updated 02021-08-15) (2 minutes)
• Constant current buck (p. 708) 02021-08-10 (updated 02021-08-15) (4 minutes)
• Methane bag (p. 710) 02021-08-10 (updated 02021-08-15) (8 minutes)
• Iodine patterning (p. 713) 02021-08-11 (updated 02021-08-15) (1 minute)
• Heating a shower tank with portable TCES? (p. 714) 02021-08-11 (updated 02021-08-15) (6 minutes)
• Subset of C for the simplest self-compiling compiler (p. 717) 02021-08-12 (updated 02021-12-30) (6 minutes)
• A compact bytecode sketch that should average about 3 bytes per line of C (p. 720) 02021-08-17 (updated 02021-09-13) (66 minutes)
• Wiki models (p. 751) 02021-08-19 (updated 02021-12-30) (1 minute)
• Residual stream windowing (p. 752) 02021-08-21 (updated 02021-09-11) (5 minutes)
• Sandwich panel optimization (p. 754) 02021-08-21 (updated 02021-09-11) (3 minutes)
• Glass wood (p. 755) 02021-08-21 (updated 02021-12-30) (4 minutes)
• Maximizing phosphate density from aqueous reaction (p. 757) 02021-08-21 (updated 02021-12-30) (8 minutes)
• Lazy heapsort (p. 761) 02021-08-22 (updated 02021-09-11) (6 minutes)
• Recursive bearings (p. 764) 02021-08-23 (updated 02021-12-30) (1 minute)
• A construction set using SHS (p. 765) 02021-08-24 (updated 02021-09-11) (5 minutes)

• Sorption vacuum pumps really can't operate continuously (p. 767) 02021-08-24 (updated 02021-09-11) (5 minutes)
• Electrodeposition welding (p. 769) 02021-08-25 (updated 02021-09-11) (2 minutes)
• Better screw head designs? (p. 770) 02021-08-25 (updated 02021-09-11) (4 minutes)
• Dense fillers (p. 772) 02021-08-25 (updated 02021-12-30) (7 minutes)
• Selective laser sintering of copper (p. 775) 02021-08-30 (updated 02021-12-30) (6 minutes)
• Negative feedback control to prevent runaway positive feedback in 3-D MIG welding printing (p. 777) 02021-08-30 (updated 02021-12-30) (3 minutes)

02021-09

• Fast electrolytic mineral accretion (seacrete) for digital fabrication? (p. 779) 02021-09-01 (updated 02021-12-30) (52 minutes)
• Patterning metal surfaces by coating decomposition with lasers or plasma? (p. 795) 02021-09-03 (updated 02021-12-30) (7 minutes)
• Rock-wool-filled composites (p. 798) 02021-09-03 (updated 02021-12-30) (2 minutes)
• Weighing balance design (p. 799) 02021-09-06 (updated 02021-12-30) (9 minutes)
• Fast-slicing ECM (p. 802) 02021-09-08 (updated 02021-12-30) (3 minutes)
• Switching kiloamps in microseconds (p. 804) 02021-09-09 (updated 02021-12-30) (1 minute)
• Spot welding (p. 805) 02021-09-09 (updated 02021-12-30) (8 minutes)
• Qfitzah: a minimal term-rewriting language (p. 809) 02021-09-10 (updated 02021-12-31) (62 minutes)
• A short list of the most useful Unix CLI tools (p. 841) 02021-09-15 (updated 02021-09-16) (2 minutes)
• Three phase differential data (p. 843) 02021-09-22 (updated 02021-12-30) (4 minutes)
• Waterglass "Loctite"? (p. 845) 02021-09-22 (updated 02021-12-30) (1 minute)
• Qfitzah internals (p. 846) 02021-09-24 (updated 02021-12-30) (2 minutes)
• Blowing agents (p. 847) 02021-09-29 (updated 02021-12-30) (4 minutes)
• Compliance spectroscopy (p. 849) 02021-09-29 (updated 02021-12-30) (4 minutes)
• Planning Apples to Apples, instead of Planning Poker (p. 851) 02021-09-29 (updated 02021-12-30) (6 minutes)
• Liquid dielectrics for hand-rolled self-healing capacitors (p. 853) 02021-09-30 (updated 02021-12-30) (3 minutes)

02021-10

• Deriving binary search (p. 855) 02021-10-01 (updated 02021-12-30) (5 minutes)
• The sol-gel transition and selective gelling for 3-D printing (p. 858)

02021-10-03 (updated 02021-12-30) (6 minutes)
• Some notes on perusing the Udanax Green codebase (p. 860) 02021-10-05 (updated 02021-10-08) (12 minutes)
• Fung's "I can't believe it can sort" algorithm and others (p. 864) 02021-10-05 (updated 02021-12-30) (5 minutes)
• Spanish phonology (p. 867) 02021-10-05 (updated 02021-12-31) (15 minutes)
• Some notes on learning Rust (p. 874) 02021-10-06 (updated 02021-10-10) (39 minutes)
• PBKDF content addressing with keyphrase hashcash: a non-blockchain attack on Zooko's Triangle (p. 896) 02021-10-08 (24 minutes)
• Wordlists for maximum drama (p. 904) 02021-10-08 (updated 02021-12-30) (16 minutes)
• The spark-pen pointing device (p. 921) 02021-10-10 (updated 02021-10-12) (1 minute)
• Beyond overstrike (p. 922) 02021-10-10 (updated 02021-12-30) (13 minutes)
• Pipelined piece chain painting (p. 926) 02021-10-10 (updated 02021-12-30) (23 minutes)
• An algebra of partial functions for interactively composing programs (p. 933) 02021-10-10 (updated 02021-12-30) (3 minutes)
• Beyond op streams (p. 935) 02021-10-11 (updated 02021-12-30) (3 minutes)
• Inverse perspective (p. 937) 02021-10-11 (updated 02021-12-30) (1 minute)
• Ranking MOSFETs for, say, rapid localized electrolysis to make optics (p. 938) 02021-10-11 (updated 02021-12-30) (8 minutes)
• The relation between solar-panel efficiency for air conditioning and insulation thickness (p. 941) 02021-10-11 (updated 02021-12-30) (3 minutes)
• An even simpler offline power supply than a capacitive dropper, with a 7¢ BOM (p. 943) 02021-10-14 (updated 02021-12-30) (7 minutes)
• Trying to quantify relative speeds of different digital fabrication processes with "matter bandwidth" (p. 946) 02021-10-15 (updated 02021-12-30) (5 minutes)
• Balanced ropes (p. 948) 02021-10-16 (updated 02021-12-30) (7 minutes)
• Flexural mounts for self-aligning bushings (p. 952) 02021-10-18 (updated 02021-12-30) (3 minutes)
• Triggering a spark gap with an exploding wire (p. 953) 02021-10-19 (updated 02021-12-30) (1 minute)
• Triggering a spark gap with low jitter using ultraviolet LEDs? (p. 954) 02021-10-20 (updated 02021-10-23) (8 minutes)
• Binomial coefficients and the dimensionality of spaces of polynomials (p. 957) 02021-10-20 (updated 02021-12-30) (4 minutes)

• Finite element analysis with sparse approximations (p. 959) 02021-10-20 (updated 02021-12-30) (2 minutes)
• Implementation and applications of low-voltage Marx generators with solid-state avalanche breakdown? (p. 960) 02021-10-20 (updated 02021-12-31) (39 minutes)
• The astounding UI responsivity of PDP-10 DDT on ITS (p. 972)

02021-10-22 (updated 02021-10-23) (28 minutes)
• Example based regexp (p. 984) 02021-10-24 (updated 02021-12-30) (5 minutes)
• Adversarial control (p. 987) 02021-10-25 (updated 02021-12-30) (13 minutes)
• Constant weight dithering (p. 991) 02021-10-28 (updated 02021-12-30) (5 minutes)
• Hashing text with recursive shingling to find duplication efficiently (p. 993) 02021-10-30 (updated 02021-12-30) (6 minutes)

02021-11

• My Heathkit H8 (p. 996) 02021-11-03 (updated 02021-12-30) (2 minutes)
• Orthogonal rational vectors (p. 997) 02021-11-04 (updated 02021-12-30) (4 minutes)
• Thread rolling roller screw (p. 999) 02021-11-04 (updated 02021-12-30) (1 minute)
• Viscoelastic probing (p. 1000) 02021-11-04 (updated 02021-12-30) (2 minutes)
• An aluminum pencil for marking iron? (p. 1001) 02021-11-06 (updated 02021-12-30) (2 minutes)
• Embedding runnable code in text paragraphs for numerical modeling (p. 1002) 02021-11-06 (updated 02021-12-30) (6 minutes)
• Paeth prediction and vector quantization (p. 1005) 02021-11-06 (updated 02021-12-30) (1 minute)
• Wire brush microscope (p. 1006) 02021-11-06 (updated 02021-12-30) (1 minute)
• New nuclear power in the People's Republic of China (p. 1007) 02021-11-09 (updated 02021-12-30) (2 minutes)
• Ivan Miranda's snap-pin fasteners and similar snaps (p. 1009) 02021-11-11 (updated 02021-12-30) (3 minutes)
• Rendering 3-D graphics with PINNs and GANs? (p. 1010) 02021-11-11 (updated 02021-12-30) (10 minutes)
• Aqueous scanning probe microscopy (p. 1013) 02021-11-12 (updated 02021-12-30) (7 minutes)
• Redundancy in self-replicating systems such as hundred-eyed chickens (p. 1016) 02021-11-12 (updated 02021-12-30) (4 minutes)
• DSLs for calculations on dates (p. 1018) 02021-11-14 (updated 02021-12-30) (1 minute)
• Some notes on reading parts of Reuleaux's engineering handbook (p. 1019) 02021-11-17 (updated 02021-12-30) (7 minutes)
• A simple 2-D programmable graphics pipeline to unify tiles and palettes (p. 1022) 02021-11-18 (updated 02021-12-30) (6 minutes)
• Interesting works that entered the public domain in 02021, in the US and elsewhere (p. 1024) 02021-11-20 (updated 02021-12-30) (15 minutes)
• At small scales, electrowinning may be cheaper than smelting (p. 1029) 02021-11-21 (updated 02021-12-30) (25 minutes)
• Micro ramjet (p. 1038) 02021-11-22 (updated 02021-12-30) (3 minutes)
• Vernier indicator (p. 1040) 02021-11-22 (updated 02021-12-30) (6 minutes)
• Some notes on Bhattacharyya's ECM book (p. 1043) 02021-11-25

(updated 02021-12-30) (11 minutes)
• Chording commands (p. 1047) 02021-11-26 (updated 02021-12-30) (7 minutes)

02021-12

• Exotic steel analogues in other metals (p. 1050) 02021-12-01 (updated 02021-12-30) (8 minutes)
• Simplest blinker (p. 1053) 02021-12-01 (updated 02021-12-30) (9 minutes)
• Capacitive linear encoder sensors (p. 1056) 02021-12-11 (updated 02021-12-30) (7 minutes)
• Two finger multitouch (p. 1059) 02021-12-11 (updated 02021-12-30) (3 minutes)
• The Habitaculum: a modular dwelling machine (p. 1061) 02021-12-13 (updated 02021-12-31) (16 minutes)
• Against subjectivism (p. 1066) 02021-12-15 (updated 02021-12-30) (36 minutes)
• Solid rock on a gossamer skeleton through exponential deposition (p. 1076) 02021-12-15 (updated 02021-12-30) (11 minutes)
• 3-D printing in poly(vinyl alcohol) (p. 1080) 02021-12-15 (updated 02021-12-30) (2 minutes)
• Ghetto electrochromic displays for ultra-low-power computing? (p. 1082) 02021-12-16 (updated 02021-12-30) (9 minutes)
• Electrolytic 2-D cutting and related electrolytic digital fabrication processes (p. 1085) 02021-12-16 (updated 02021-12-30) (48 minutes)
• Layers plus electroforming (p. 1100) 02021-12-16 (updated 02021-12-30) (7 minutes)
• MOSFET body diodes as Geiger counter avalanche detectors? (p. 1103) 02021-12-17 (updated 02021-12-30) (1 minute)
• The user interface potentialities of a barcoded paper notebook (p. 1104) 02021-12-18 (updated 02021-12-30) (6 minutes)
• Aluminum refining (p. 1106) 02021-12-20 (updated 02021-12-30) (3 minutes)
• Regenerative muffle kiln (p. 1108) 02021-12-21 (updated 02021-12-30) (19 minutes)
• Is liberal democracy's stability conditioned on historical conditions that no longer obtain? (p. 1114) 02021-12-22 (updated 02021-12-30) (16 minutes)
• Xerogel compacting (p. 1119) 02021-12-22 (updated 02021-12-30) (12 minutes)
• Photoemissive power (p. 1124) 02021-12-23 (updated 02021-12-28) (15 minutes)
• Toggling eccentrics for removing preload from spring clamps (p. 1129) 02021-12-28 (updated 02021-12-31) (22 minutes)
• Safe decentralized cloud storage (p. 1135) 02021-12-30 (10 minutes)

Topics

• Materials (p. 1138) (59 notes)
• Programming (p. 1141) (49 notes)
• Contrivances (p. 1143) (45 notes)

- Electronics (p. 1145) (39 notes)
- Pricing (p. 1147) (35 notes)
- Digital fabrication (p. 1149) (31 notes)
- Manufacturing (p. 1151) (29 notes)
- History (p. 1153) (24 notes)
- Performance (p. 1155) (22 notes)
- Human-computer interaction (p. 1156) (22 notes)
- Physics (p. 1157) (18 notes)
- Electrolysis (p. 1158) (18 notes)
- Mechanical (p. 1159) (17 notes)
- 3-D printing (p. 1160) (17 notes)
- Filled systems (p. 1161) (16 notes)
- Experiment report (p. 1162) (14 notes)
- Algorithms (p. 1163) (14 notes)
- Strength of materials (p. 1164) (13 notes)
- Machining (p. 1165) (13 notes)
- Python (p. 1166) (12 notes)
- Pulsed machinery (p. 1167) (12 notes)
- Frrickin' lasers! (p. 1168) (12 notes)
- Ghettobotics (p. 1169) (12 notes)
- Energy (p. 1170) (12 notes)
- Bootstrapping (p. 1171) (12 notes)
- Safe programming languages (p. 1172) (11 notes)
- Math (p. 1173) (11 notes)
- Lisp (p. 1174) (11 notes)
- Assembly-language programming (p. 1175) (11 notes)
- Power supplies (p. 1176) (10 notes)
- Graphics (p. 1177) (10 notes)
- Compilers (p. 1178) (10 notes)
- Clay (p. 1179) (10 notes)
- Aluminum (p. 1180) (10 notes)
- Welding (p. 1181) (9 notes)
- Virtual machines (p. 1182) (9 notes)
- Precision (p. 1183) (9 notes)
- Phosphates (p. 1184) (9 notes)
- Foam (p. 1185) (9 notes)
- ECM (p. 1186) (9 notes)
- Composites (p. 1187) (9 notes)
- Composability (p. 1188) (9 notes)
- Waterglass (p. 1189) (8 notes)
- Small is beautiful (p. 1190) (8 notes)
- Sensors (p. 1191) (8 notes)
- Programming languages (p. 1192) (8 notes)
- Ceramic (p. 1193) (8 notes)
- C (p. 1194) (8 notes)
- Real time (p. 1195) (7 notes)
- Higher order programming (p. 1196) (7 notes)
- Hand tools (p. 1197) (7 notes)
- Falstad's circuit simulator (p. 1198) (7 notes)
- Facepalm (p. 1199) (7 notes)
- Argentina (p. 1200) (7 notes)
- 2-D cutting (p. 1201) (7 notes)
- Terminals (p. 1202) (6 notes)
- Solar (p. 1203) (6 notes)

• Self replication (p. 1204) (6 notes)
• Systems architecture (p. 1205) (6 notes)
• Protocols (p. 1206) (6 notes)
• Post-teletype terminal design (p. 1207) (6 notes)
• Physical computation (p. 1208) (6 notes)
• Optics (p. 1209) (6 notes)
• Minerals (p. 1210) (6 notes)
• Microcontrollers (p. 1211) (6 notes)
• Metrology (p. 1212) (6 notes)
• Magnesium (p. 1213) (6 notes)
• Instruction sets (p. 1214) (6 notes)
• Independence (p. 1215) (6 notes)
• GUIs (p. 1216) (6 notes)
• End user programming (p. 1217) (6 notes)
• Anisotropic fillers (p. 1218) (6 notes)
• Thermodynamics (p. 1219) (5 notes)
• The future (p. 1220) (5 notes)
• Syntax (p. 1221) (5 notes)
• Steel (p. 1222) (5 notes)
• Small things (p. 1223) (5 notes)
• Security (p. 1224) (5 notes)
• Refractory (p. 1225) (5 notes)
• Powder-bed 3-D printing processes (p. 1226) (5 notes)
• The Portable Document Format (PDF) (p. 1227) (5 notes)
• Parsing (p. 1228) (5 notes)
• Numerical modeling (p. 1229) (5 notes)
• Incentives (p. 1230) (5 notes)
• FORTH (p. 1231) (5 notes)
• Flexures (p. 1232) (5 notes)
• File formats (p. 1233) (5 notes)
• Copper (p. 1234) (5 notes)
• Cements (p. 1235) (5 notes)
• Bytecode (p. 1236) (5 notes)
• Aluminum foil (p. 1237) (5 notes)
• Vermiculite (p. 1238) (4 notes)
• Transactions (p. 1239) (4 notes)
• Sparks (p. 1240) (4 notes)
• Self-propagating high-temperature synthesis (SHS) (p. 1241) (4 notes)
• Scanning probe microscopy (p. 1242) (4 notes)
• Reverse Polish notation (RPN) (p. 1243) (4 notes)
• Reading (p. 1244) (4 notes)
• Poly(vinyl alcohol) (PVA) (p. 1245) (4 notes)
• Program calculator (p. 1246) (4 notes)
• Pascal (p. 1247) (4 notes)
• Operating systems (p. 1248) (4 notes)
• OCaml (p. 1249) (4 notes)
• Memory hardware (p. 1250) (4 notes)
• Life support (p. 1251) (4 notes)
• Input devices (p. 1252) (4 notes)
• Heating (p. 1253) (4 notes)
• Glass (p. 1254) (4 notes)
• Garbage collection (p. 1255) (4 notes)
• Encoding (p. 1256) (4 notes)

- Editors (p. 1257) (4 notes)
- Economics (p. 1258) (4 notes)
- Dynamic dispatch (p. 1259) (4 notes)
- Domain-specific languages (DSLs) (p. 1260) (4 notes)
- Displays (p. 1261) (4 notes)
- Control (cybernetics) (p. 1262) (4 notes)
- Compression (p. 1263) (4 notes)
- Communication (p. 1264) (4 notes)
- Ceramic-matrix composites (CMCs) (p. 1265) (4 notes)
- Caching (p. 1266) (4 notes)
- Weighing (p. 1267) (3 notes)
- Unix (p. 1268) (3 notes)
- Tiled graphics (p. 1269) (3 notes)
- Term rewriting (p. 1270) (3 notes)
- Sugar (p. 1271) (3 notes)
- Sorting (p. 1272) (3 notes)
- Solubility (p. 1273) (3 notes)
- Scheme (p. 1274) (3 notes)
- Roller screws (p. 1275) (3 notes)
- RISC-V (p. 1276) (3 notes)
- Reproducibility (p. 1277) (3 notes)
- Radio (p. 1278) (3 notes)
- Politics (p. 1279) (3 notes)
- Illinois PLATO (p. 1280) (3 notes)
- Poly(lactic acid) (PLA) (p. 1281) (3 notes)
- Patterning (p. 1282) (3 notes)
- Oscillators (p. 1283) (3 notes)
- Natural-language processing (p. 1284) (3 notes)
- Memory models (p. 1285) (3 notes)
- LEDs (p. 1286) (3 notes)
- Kleene algebras (p. 1287) (3 notes)
- Kingery, the father of modern ceramics (p. 1288) (3 notes)
- Keyboards (p. 1289) (3 notes)
- Insulation (p. 1290) (3 notes)
- Hypertext (p. 1291) (3 notes)
- Humor (p. 1292) (3 notes)
- Hashing (p. 1293) (3 notes)
- Glutaraldehyde (p. 1294) (3 notes)
- Forming (p. 1295) (3 notes)
- Flying (p. 1296) (3 notes)
- Fasteners (p. 1297) (3 notes)
- Emacs (p. 1298) (3 notes)
- Control flow (p. 1299) (3 notes)
- COMFY-* (p. 1300) (3 notes)
- Cameras (p. 1301) (3 notes)
- Batteries (p. 1302) (3 notes)
- BASIC (p. 1303) (3 notes)
- Audio (p. 1304) (3 notes)
- ASCII art (p. 1305) (3 notes)
- Art (p. 1306) (3 notes)
- Artificial neural networks (p. 1307) (3 notes)
- Allocation performance (p. 1308) (3 notes)
- Alabaster (p. 1309) (3 notes)
- X rays (p. 1310) (2 notes)

- Wiki (p. 1311) (2 notes)
- Video (p. 1312) (2 notes)
- The Veskeno virtual machine (p. 1313) (2 notes)
- The United States of America (USA) (p. 1314) (2 notes)
- Unicode (p. 1315) (2 notes)
- Toxicology (p. 1316) (2 notes)
- Thixotropy (p. 1317) (2 notes)
- Tcl (p. 1318) (2 notes)
- Talc (p. 1319) (2 notes)
- Stack machines (p. 1320) (2 notes)
- Spreadtools (p. 1321) (2 notes)
- Speech synthesis (p. 1322) (2 notes)
- Space (p. 1323) (2 notes)
- Sonic screwdrivers (p. 1324) (2 notes)
- Snaps (p. 1325) (2 notes)
- Smalltalk (p. 1326) (2 notes)
- Spatial light modulators (SLMs) (p. 1327) (2 notes)
- Silver (p. 1328) (2 notes)
- Silicone (p. 1329) (2 notes)
- Steel Bank Common Lisp (p. 1330) (2 notes)
- Sapphire (p. 1331) (2 notes)
- Sandblasting (p. 1332) (2 notes)
- Ropes (p. 1333) (2 notes)
- Regenerators (p. 1334) (2 notes)
- Refining (p. 1335) (2 notes)
- Randomness (p. 1336) (2 notes)
- Qfitzah (p. 1337) (2 notes)
- Prefix sums (p. 1338) (2 notes)
- Plasma (p. 1339) (2 notes)
- Piezoelectrics (p. 1340) (2 notes)
- Photoemission (p. 1341) (2 notes)
- Perl (p. 1342) (2 notes)
- Parsing expression grammars (PEGs) (p. 1343) (2 notes)
- Passwords (p. 1344) (2 notes)
- The Paeth predictor (p. 1345) (2 notes)
- Memory ownership (p. 1346) (2 notes)
- Overstrike (p. 1347) (2 notes)
- Mathematical optimization (p. 1348) (2 notes)
- Oogoo (p. 1349) (2 notes)
- Ontology (p. 1350) (2 notes)
- Namespaces (p. 1351) (2 notes)
- m4 (p. 1352) (2 notes)
- LuaJIT (p. 1353) (2 notes)
- Lua (p. 1354) (2 notes)
- LiDAR (p. 1355) (2 notes)
- Length (p. 1356) (2 notes)
- Layout (p. 1357) (2 notes)
- Latency (p. 1358) (2 notes)
- The JS programming language (p. 1359) (2 notes)
- JLCPCB (JiaLiChuang) (p. 1360) (2 notes)
- Interrupts (p. 1361) (2 notes)
- Incremental search (p. 1362) (2 notes)
- Household (p. 1363) (2 notes)
- Gradient descent (p. 1364) (2 notes)

- Gears (p. 1365) (2 notes)
- Generative adversarial networks (GANs) (p. 1366) (2 notes)
- Galileo (p. 1367) (2 notes)
- Fiction (p. 1368) (2 notes)
- Enthalpy (p. 1369) (2 notes)
- Employment (p. 1370) (2 notes)
- Electropolishing (p. 1371) (2 notes)
- Electroforming (p. 1372) (2 notes)
- Dreaming (p. 1373) (2 notes)
- Decentralization (p. 1374) (2 notes)
- Debugging (p. 1375) (2 notes)
- Databases (p. 1376) (2 notes)
- Cross linking (p. 1377) (2 notes)
- Command-line interfaces (CLI) (p. 1378) (2 notes)
- China (p. 1379) (2 notes)
- Ccn (p. 1380) (2 notes)
- Carborundum (p. 1381) (2 notes)
- Call by name (p. 1382) (2 notes)
- Block arguments (p. 1383) (2 notes)
- Bicicleta (p. 1384) (2 notes)
- Barcodes (p. 1385) (2 notes)
- Azane (p. 1386) (2 notes)
- AVR8 microcontrollers (p. 1387) (2 notes)
- Arduino (p. 1388) (2 notes)
- Archival (p. 1389) (2 notes)
- Apl (p. 1390) (2 notes)
- Ambiq (p. 1391) (2 notes)

liabilities/LICENSE.ETBook

```
[This is the copyright notice from the ET Book font Dercuano uses.]

Copyright (c) 2015 Dmitry Krasny, Bonnie Scranton, Edward Tufte.


Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:


The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.


THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

liabilities/dejavu-copyright

Format: http://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
Upstream-Name: DejaVu fonts
Upstream-Author: Stepan Roh <src@users.sourceforge.net> (original author),
see /usr/share/doc/ttf-dejavu/AUTHORS for full list
Source: http://dejavu-fonts.org/

Files: *
Copyright: Copyright (c) 2003 by Bitstream, Inc. All Rights Reserved.
Bitstream Vera is a trademark of Bitstream, Inc.
DejaVu changes are in public domain.
License:
Permission is hereby granted, free of charge, to any person obtaining a copy of the fonts accompanying this license ("Fonts") and associated documentation files (the "Font Software"), to reproduce and distribute the Font Software, including without limitation the rights to use, copy, merge, publish, distribute, and/or sell copies of the Font Software, and to permit persons to whom the Font Software is furnished to do so, subject to the following conditions:
.
The above copyright and trademark notices and this permission notice shall be included in all copies of one or more of the Font Software typefaces.
.
The Font Software may be modified, altered, or added to, and in particular the designs of glyphs or characters in the Fonts may be modified and additional glyphs or characters may be added to the Fonts, only if the fonts are renamed to names not containing either the words "Bitstream" or the word "Vera".
.
This License becomes null and void to the extent applicable to Fonts or Font Software that has been modified and is distributed under the "Bitstream Vera" names.
.
The Font Software may be sold as part of a larger software package but no copy of one or more of the Font Software typefaces may be sold by itself.
.
THE FONT SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF COPYRIGHT, PATENT, TRADEMARK, OR OTHER RIGHT. IN NO EVENT SHALL BITSTREAM OR THE GNOME FOUNDATION BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, INCLUDING ANY GENERAL, SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF THE USE OR INABILITY TO USE THE FONT SOFTWARE OR FROM OTHER DEALINGS IN THE FONT SOFTWARE.
.
Except as contained in this notice, the names of Gnome, the Gnome Foundation, and Bitstream Inc., shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Font Software without prior written authorization from the Gnome Foundation or Bitstream Inc., respectively. For further information, contact: fonts at gnome dot

org.

Files: debian/*
Copyright: (C) 2005-2006 Peter Cernak <pce@users.sourceforge.net>
(C) 2006-2011 Davide Viti <zinosat@tiscali.it>
(C) 2011-2013 Christian Perrier <bubulle@debian.org>
(C) 2013 Fabian Greffrath <fabian+debian@greffrath.com>
License: GPL-2+

This program is free software; you can redistribute it
and/or modify it under the terms of the GNU General Public
License as published by the Free Software Foundation; either
version 2 of the License, or (at your option) any later
version.

.

This program is distributed in the hope that it will be
useful, but WITHOUT ANY WARRANTY; without even the implied
warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE. See the GNU General Public License for more
details.

.

You should have received a copy of the GNU General Public
License along with this package; if not, write to the Free
Software Foundation, Inc., 51 Franklin St, Fifth Floor,
Boston, MA 02110-1301 USA

.

On Debian systems, the full text of the GNU General Public
License version 2 can be found in the file
/usr/share/common-licenses/GPL-2'.

Fan noise would be less annoying if intermittent

Kragen Javier Sitaker, 02021-01-03 (updated 02021-01-04) (1 minute)

Fan noise is painfully irritating in part because it's constant and relatively spectrally pure — it's not a sinusoid, but it's almost all harmonics of a particular frequency, so the sonic energy is concentrated with high Q in a small fraction of all possible frequencies.

In experimenting with bytebeat I've found that a bit of envelope modulation at 0.5–2 Hz can go a long way toward reducing the pain level of tones. Could you do that with a fan? For example, closing the fan with a baffle for 500 ms every 2 seconds would nearly silence it.

Topics

- Contrivances (p. 1143) (45 notes)
- Household (p. 1363) (2 notes)

Principled APL redux

Kragen Javier Sitaker, 02021-01-03 (updated 02021-12-31)
(12 minutes)

Writing rpneact I realized that there are some straightforward ways to extend such a system toward Bicicleta and the principled variant of APL I was thinking about, as well as to do performance work.

One is that inheritance is relatively straightforward:

```
class Var(namedtuple('Var', ('name',)), Expr):
    def eval(self, vars):
        return vars[self.name].eval(vars)
```

If the second `vars` were a different object, we'd have the equivalent of invoking an inherited method — any vars referred to by the formula would come from the different object. (And this is exactly how inheritance works in Bicicleta and in the ς-calculus it's based on.) And of course in Python such inheritance is straightforward to implement in the `vars` object by having its `__getattr__` delegate to one or more different mappings.

There's code in rpneact for the solver that evaluates a loss formula at various points along a design variable; it works as follows:

```
var = design_var.name
original = vars[var]
...

try:
    ...
            vars[var] = Const(b)
            b_obj = objective.eval(vars)
    ...
finally:
    vars[var] = original
```

This could be done in a purely functional way instead by the inheritance mechanism described above.

Let's call these "vars" namespace objects that might inherit from one another "worlds".

Bitmask caching

For caching property values in such a system, you'd like to carry over the cached value of a property to any child worlds that hadn't overridden any other property used during its computation. Moreover, if that value didn't get cached in the parent but did get computed in the child entirely from properties unchanged from the parent, you'd like to propagate those values back to the parent, so that other children can use them.

I'm not sure what's an actually scalable way to implement this, but an approach that's reasonable up to a certain extent, with single

inheritance, is to maintain a bitmask of overridden fields in each world. Each field is assigned an index, starting from the number of fields in the base class. Each cached result is stored at that index in a cached result vector associated with each world; each field definition is stored at that index in a definition vector; and in the bitmask of overridden fields for an world, the bit is set if the world has its own private definition of a field, rather than using the parent world's. And each cached result is associated with a bitmask of all the fields consulted during its computation.

(For the moment I'm going to ignore dependencies on things outside of the world itself, which were common in Bicicleta but sort of impossible in the principled-APL approach.)

These bitmasks can be 64 bits for worlds with 64 fields or less, so we can operate bitwise on them in a single instruction on a 64-bit CPU, or 128 bits for worlds with 128 fields or less. In the limit of a large number of fields, of course, such explicit sets will tend to incur a large space and time cost.

To know whether the parent world's value of a variable can be safely reused, we can AND the current world's overridden-field bitmask with the dependency bitmask for the parent world's variable; if the intersection is null, we can just reuse the parent's value (and perhaps also cache it locally). Similarly, if we compute a field value locally, we also compute its dependency bitmask, so we can check in the same way to see whether it involved any locally overridden values (although you could imagine that maybe it would be more efficient to just set a flag during the computation, you have to compute the dependency bitmask anyway) and then do the same check. This lets you see how far up the inheritance hierarchy you can push the cached value.

When a cached field is consulted during the computation of a field value, we must OR its dependency bitmask into the dependency bitmask of the computation, as well as set its bit.

Compilation and duplicate suppression

In a system compiled without runtime reflection, all of this bitmap consultation can happen at compile time; only the consultation to see if a field is already computed, and if so where, and what its value is, must be deferred to runtime. This in some sense involves compiling a separate version of each world for each callsite, depending on the particular overrides provided at that callsite, but in many cases these worlds will use precisely the same code and can thus be merged, as with C++ templates. (And runtime reflection can be supported in such a system by invoking the compiler at runtime.)

If an override method doesn't use any other fields of the object it's being stuck onto, it could be implemented as a simple delegation to a thunk. Then all the worlds that differ only by the contents of that thunk can be merged. For example, in Bicicleta:

```
prog.if(foo, then={all kinds of stuff}, else={more stuff})
```

can be merged with all the other invocations of `prog.if` that override only `arg0`, `then`, and `else`.

Field order

The fields within a world can be topologically sorted to emit a single sequence of code that computes all of them in dependency order. In cases where no cached results are possible, for example when a program starts, if there's no risk to termination behavior, we can run all the fields peremptorily in sequence, one after the other. In other cases, it may be sensible to do a slightly modified version of the same thing: concatenate the code and put a conditional jump past the computation of each field. The jump can be based on, for example, a bitmap of fields whose computation is desired: the abjunction of the dependencies of a desired-fields bitmap and an already-computed-fields bitmap. In many cases, we can jump directly to the first not-yet-computed field, and run straight-line code from there, perhaps even without such conditional skips.

Inlining the computations of other worlds' fields may be worthwhile, particularly when no references to those other worlds themselves escape from the computation. The `if` case above is paradigmatic: no reference to `if` world survives, just its output.

General inlining in a system as late-bound as the original Bicicleta design would require type specialization using hidden classes. Maybe a "Triciclo" design could omit the ability to override standard operators like arithetic, but you could probably get to better performance than CPython or similar systems by the Ur-Scheme approach of inlining the implementation of native operators (integer addition, etc.) with a conditional jump to an exception handler for cases where an operator is overridden. But this is to some extent orthogonal to the question of the caching strategy!

Simple compilation

In simple cases, the strategy is:

- Topologically sort the variables.
- Assign each variable a memory location, not necessarily in sequence.

- Generate the code snippet to compute each variable from the other variables, fetching from memory and storing into memory as necessary.
- Divide the topologically-sorted sequence into "basic blocks" approximating some fixed granularity, say, 256 clock cycles.
- Assign each basic block a memory location for a flag that indicates whether its results are valid (up-to-date) or not.
- For each basic block, emit a conditional checking to see whether its results are valid, and if not, executing the concatenated code snippets for its variables, then setting the flag to indicate that its results are indeed valid.
- Peephole-optimize each basic block.
- Eliminate memory allocations no longer read after peephole-optimization, unless they're final outputs.
- Concatenate the basic blocks.

So for a full computation from scratch, you clear all the validity flags, set the inputs, and jump to the beginning of the first basic block. In cases where you've changed some input, you clear some of the flags

(following the transitive closure of dependency relations between the blocks) and jump to the beginning of the first one whose validity flag isn't set there's a granularity tradeoff where making the blocks bigger also increases the amount of redundant computation they do: you may recompute a variable that didn't need recomputing because it was in the same basic block as one that did using larger basic blocks privileges worst-case computation time (where nothing is cached), while using smaller ones privileges best-case incremental update time.

There's no particular reason that this honking chunk of compiled code can't accept a base address argument off which to find its variables and validity flags, nor, for that matter, two different base addresses off which to find variables and validity flags that vary differently. And these "base addresses" can as easily be array indices. For example, you might have some set of variables that vary by day, and other variables that are global across all days. The basic blocks for the per-day variables would use the day index to index into an array of validity flags, one flag per day per basic block, as well as arrays of variable values to compute; the basic blocks for the non-per-day variables would not.

In the simplest case, the per-day variables are all computed from the non-per-day variables and not vice versa, but once reductions, quantifiers, and indexing are involved, the picture gets more complicated: you might have non-per-day variables computed from some of the per-day variables, more per-day variables computed from those, and so on. This affects how you can bundle them into basic blocks efficiently.

(Is Datalog-style stratified inference applicable to this problem?)

Transactional arrayification

Suppose you populate memory with some indepdendent-variable values and run a hunk of machine code in some kind of valgrind-like monitoring mode that monitors what memory locations it reads and writes. After it completes, you can check to see which of your input values it read; you can then infer that the locations that it *wrote* depend at least on the independent variables that it *read*. For example, if it read x and y but not z, and wrote a and b, you can conclude that a and b are dependent on x and y, but maybe not z.

If you have a set of possibilities in mind for x and y, you can write each of the possibilities into the x and y memory locations and rerun the code, thus deriving a and b values for each (x, y) pair. At some point you may or may not observe a dependency on z as well; if so, you might have to try all the z values as well.

You may be able to provide x, y, z, a, and b as closures rather than memory locations, in which case efficiently noting that they are being accessed, without slowing down the rest of the computation, becomes easy. This is the approach taken by, for example, Meteor and Adapton. Moreover, depending on the degree to which local memory writes can be controlled, you may be able to use such thunks to save backtracking state — if x is consulted some time before y, it may be worthwhile to just roll back the computation to the first consultation of y to try different y values.

Topics

- Programming (p. 1141) (49 notes)
- Performance (p. 1155) (22 notes)
- Safe programming languages (p. 1172) (11 notes)
- Compilers (p. 1178) (10 notes)
- Programming languages (p. 1192) (8 notes)
- Transactions (p. 1239) (4 notes)
- Caching (p. 1266) (4 notes)
- Bicicleta (p. 1384) (2 notes)
- Apl (p. 1390) (2 notes)
- Datalog
- Arrays

First class locations

Kragen Javier Sitaker, 02021-01-04 (3 minutes)

As mentioned in Layout typescript (p. 29) one of the deficiencies of the Lisp object-graph memory model is that *places* aren't first-class: object slots (instance variables or record fields), local variables, global variables, array elements, and so on, can't be referred to in the language except indirectly. In that note, I was pondering the implications for LispM-style "presentations" in a typescript, which I'd like to make interactively explorable by default, but I've also written about a related question in terms of IMGUI libraries.

In an IMGUI library, it's very convenient to be able to say something like (from [Dear ImGui]):

```
ImGui::SliderFloat("this field is called f", &f, 0.0f, 1.0f);
```

This works in C because we can pass a pointer to the global variable `f` to the slider; similarly we can pass pointers to record (struct) fields. Golang and C++ are similarly empowered. But this is potentially unsafe, in the sense that we can also pass pointers to local variables, the callee can save the pointer somewhere, and those local variables can then go out of scope, leaving a dangling pointer ready to cause mischief.

In languages like Pascal, a safe version of this facility is available as "var parameters": a parameter which, rather than being a local variable with a copy of the passed-in value, is an alias to a location provided by the caller. This preserves the freedom for the compiler to manage activation records with a stack discipline, while also providing the ability for SliderFloat or whatever to get the reference it wants. The callee can pass this reference to other subroutines but cannot reseat the reference or save it elsewhere. C++'s reference type works similarly.

On a modern machine, we could imagine saving activation records on the heap and capturing references to them inside of, for example, textual or graphical output, enabling a debugger to go back and trace the "why" of a given value — as in Bret Victor's tree demo from *Inventing on Principle*, for example. In an IMGUI context we can fake it by recreating the XXX

Topics

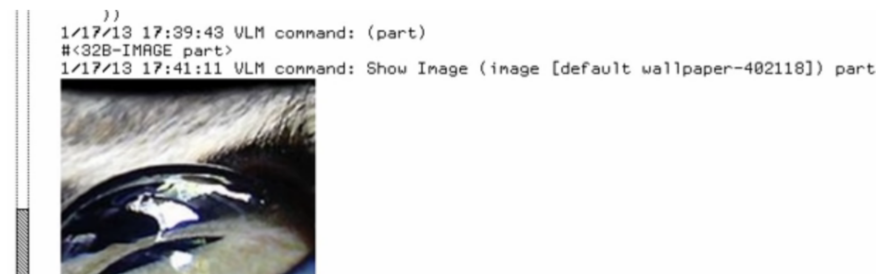
- Programming (p. 1141) (49 notes)
- Human-computer interaction (p. 1156) (22 notes)
- Safe programming languages (p. 1172) (11 notes)
- Programming languages (p. 1192) (8 notes)
- C (p. 1194) (8 notes)
- Terminals (p. 1202) (6 notes)
- Post-teletype terminal design (p. 1207) (6 notes)
- End user programming (p. 1217) (6 notes)

- Pascal (p. 1247) (4 notes)
- Memory ownership (p. 1346) (2 notes)
- Immediate-mode GUIs

Layout typescript

Kragen Javier Sitaker, 02021-01-04 (5 minutes)

I was watching Kalman Reti demo an (emulated) LispM linked from a thread on the orange website about REPL-driven development. Aside from enjoying the printed-book typography of the user interface, I thought it might be worthwhile to jot down some notes about the interaction model.



The Documents as User Interfaces demo of Cedar Tioga on a SPARCStation is a related promising direction, one that's viable today to some extent inside Emacs and web browsers: "Because buttons are represented just like other document properties, they can easily be generated by programs. For instance, we created a directory listing program to generate a button for each file listed. Clicking on such a button opens the named file." (4'26")

I thought the concept of linking text in a typescript ("presentations of objects I've already printed on my typescript") to live objects to make it "mouse sensitive" was interesting. What if your "typescript" consisted entirely of such "live objects"? Is that a viable mode of interaction?

That is, what if you could still `print(foo)` for whatever object `foo`, but the result was not merely appending some dumb characters to a buffer, but rather appending the object denoted by `foo` to the layout of a scrollable document capable of interaction? `foo` could, for example, be another such typescript, visually nested within the larger one.

To some extent I think the Lisp object-graph model (used by, say, Python) may not be ideally suited to this: if you print out, for example, an integer, an interaction you might want to support is *change* that integer. In the Lisp model the best you can do is to do print an integer-cell object which supports interactions to change its value; the problem with this is that if what you're printing is, say, the `x` field of some object, it's unlikely you would have previously thought ahead to mmake that field an integer-cell object.

Basically the problem is that in the Lisp memory model values like integers are first-class, but storage locations such as integer slots are not first-class. (Common Lisp patches over this to some extent with the "generalized places" concept of `setf`, but you still have to write `(defun (setf foo)` ... all the time.) See First class locations (p. 27) for more notes on this.

But in a situation like that, where you're changing some value you maybe used later in a calculation or to display something, you'd maybe sort of like to redo those calculations or that display. So there's

a whole question of spreadsheet-like incremental recomputation involved. (A key point about Reti's demo is that at one point he modifies some image data, but because the pixels on the screen are cached copies of a previous version of the image, they don't update until he rebuilds the screen from scratch.)

The LispM sort of doesn't exploit the full potential of a graphical terminal; in one of Reti's examples, at 5'7", he clicks on the "Read Image File" command from the help and starts typing "linux", and the resulting text is

```
1/17/13 17:34:16 VLM command: Read Image File (from file(s) [default WILSON:>reti
>demo.lisp.newest]) linux
```

Which, with the advantage of 40 years of UI research, I strongly suspect would work better by putting the default filename into the place where you're typing, rather than to the left of it. And I suspect it would be better to show all the arguments for a multi-argument command at once rather than one at a time.

(Incidentally, in this particular example, the similarity to Emacs is very strong; not only does Emacs prompt with the same style, but he's also accessing a remote filesystem in the same way as ange-ftp and TRAMP.)

The LispM interaction model is somewhat modal; as Reti points out at 6'14", none of the previously-mouse-sensitive things are mouse-sensitive at the point that he's being prompted for an image to load, so it's still essentially a command-line interface — it isn't so much that the various bits of text in the typescript support separate interactions themselves as that they are (or are not) viable arguments for the single interaction going on at the moment. Some form of such modality is apparently unavoidable (consider xmag, xkill, "Inspect Element", and screenshots) but it would be nice to keep it quasimodal rather than fully modal; with modern multitouch displays, this might be less challenging than previously.

A potentially interesting thing about building up a typescript as a layout of objects is that it could start very small on your display and grow as needed before starting to scroll.

Topics

- Programming (p. 1141) (49 notes)
- History (p. 1153) (24 notes)
- Human-computer interaction (p. 1156) (22 notes)
- Lisp (p. 1174) (11 notes)
- Terminals (p. 1202) (6 notes)
- Post-teletype terminal design (p. 1207) (6 notes)
- Operating systems (p. 1248) (4 notes)
- Emacs (p. 1298) (3 notes)
- Video (p. 1312) (2 notes)
- Layout (p. 1357) (2 notes)

Fibonacci scan

Kragen Javier Sitaker, 02021-01-10 (updated 02021-01-15) (1 minute)

The Fibonacci sequence is $F(0) = 1$, $F(1) = 1$, $F(n>1) = F(n-1) + F(n-2)$, giving 1 1 2 3 5 8 13 21 34 55 89… and a property I just noticed is that $\Sigma_n F(n)$ for $0 \le n \le m$ is just $F(m + 2) - 1$.

We can observe that this property is true for $m = 0$: the sum is 1, $m + 2 = 2$, $F(2) = 2$, so $F(2) - 1 = 1$. When we advance the sum from m_0 to $m_0 + 1$, we are adding $F(m_0 + 1)$ to it. $F(m_0 + 3) = F(m_0 + 2) + F(m_0 + 1)$, by definition, so if this property is true for some m_0, it is also true for $m_0 + 1$. So by induction it is always true.

In some sense we should expect something like this to be true, since the Fibonacci sequence grows exponentially, but it was still a bit of a surprise to me to observe that 1+1+2+3+5+8+13+21+34 = 89-1.

Topics

- Math (p. 1173) (11 notes)
- Prefix sums (p. 1338) (2 notes)

Relayout with heaps

Kragen Javier Sitaker, 02021-01-10 (updated 02021-01-15)
(6 minutes)

I was just writing a table editor in Python. Modern machines are fast, so simple brute force was sufficient to instantly recalculate the typewriter width of each column in the redraw:

```
def redraw(self, output):
    widths = [max(1, max(len(s) for s in col)) for col in self.contents]
    output.write('\033[2J\033[0H') # clear screen, home
    ...
```

Until I loaded a two-megabyte file with 100k rows and 400k cells into it, anyway, and redraw started taking a noticeable fraction of a second.

There are a lot of ways you can change the problem to make it easier. You can divide your table into pages with independently varying column widths. You can compute column widths with a 512-row window around the cursor that moves in big jumps with a little hysteresis. You can require the user to set the column widths manually. You can render the display immediately with the outdated column width, recomputing the correct column width in the background, and update the display again when you're done. You can cache the column widths, only recomputing them after a change.

But most changes — the vast majority, in fact — don't change a column width, so you can do better than just recomputing the column width after a change. If you make a cell narrower, it only changes the column width if it was previously of the maximal width in the column, and no other cell is still that maximal width. If you make it wider, it only changes the column width if it becomes wider than the previous column width, and then its new width is the new column width.

So, even though the column width depends on every cell in the column (100k cells in this example) a custom cache invalidation strategy can eliminate most column-width recomputations and make many others trivial: the new column width is the cell's new width. But there's still a case where you need to do a potentially expensive computation: where you're making the cell so narrow that it's narrower than the column. You need to figure out what the new narrow cell is.

Previously I've written about computing such semilattice reductions incrementally using the standard parallel or incremental prefix-sum algorithm, which gives you a logarithmic-time way of finding the new maximum. In this case, for example, you could have a 17-level perfect binary tree of maximal cell widths, each node annotated with the larger cell width from its two children, with the overall maximal width at the base; whenever you increase or reduce a cell width, you propagate that reduction up the tree, potentially all the way to the root, requiring only 17 pairwise max operations in the worst case, but only about two in the average case.

But it occurred to me that a possibly different approach is to build a max-heap over the cell widths. If a binary heap (rather than, for example, a Fibonacci heap), this is also a perfect binary tree, but because it the widths it keeps in internal nodes are not duplicated in leaves, it's half as big, and is thus only 16 levels deep. It also requires about two operations in the average case, this time comparisons and conditional swaps.

One tricky bit is that it isn't sufficient to just keep the widths themselves in the heap; we need a bit more metadata in order to be able to update them. There are a couple of different approaches. First, we can make the heap items into (width, cellindex) pairs, inserting a new one every time a cell changes width; and, when we consult the one at the top of the heap, we we check to see if its width is still up to date, and if not, we discard it and check the next one. Second, we can mutate the items within the heap, maintaining an index into the heap in the cell structure itself, with which to find its corresponding heap item and sift it up or down as appropriate; this index must be updated whenever the heap item is shouldered aside by a larger cell sifting up, or promoted by a smaller cell sifting down, so the heap item also needs a pointer back to the cell structure in this case, so it still needs to contain the cellindex, though it no longer needs to contain the width itself.

Note that, unless the cellindex is smaller to store than the width, this eliminates the apparent twofold size advantage of the bin-heap!

The heap approach also requires a total ordering on its keys, so it's *less general* than the reduction prefix-sum tree, which can work on more general semilattices as well as other monoids and even I think semigroups.

Reasoning in the opposite direction, the semilattice-reduction prefix-sum tree would seem to provide a viable logarithmic-time implementation of a priority queue, which will be cheaper if the items in the priority queue need to be addressable by something other than priority — process ID, for example, or table cell index. If you have a hash table of scheduled events, for example, you should be able to maintain a perfect binary tree of earliest scheduled events in various 2^n-sized subsets of the hash buckets. This will obviously be costly if most of the hash buckets are empty, but with modern hashing techniques like cuckoo hashing, the number of empty buckets can be kept very low.

So, is there ever a reason to use a binary heap? Heapsort is an in-place worst-case linearithmic comparison sort, and I don't think you can do that with this prefix-sum tree thing. But I think the usual priority-queue problems can be solved just as well with the prefix-sum-tree approach.

Topics

- Programming (p. 1141) (49 notes)
- Performance (p. 1155) (22 notes)
- Human-computer interaction (p. 1156) (22 notes)
- Algorithms (p. 1163) (14 notes)

- Python (p. 1166) (12 notes)
- Real time (p. 1195) (7 notes)
- Caching (p. 1266) (4 notes)
- Prefix sums (p. 1338) (2 notes)
- Layout (p. 1357) (2 notes)

Transactional editor

Kragen Javier Sitaker, 02021-01-14 (updated 02021-01-15)
(73 minutes)

I was thinking about the transaction-per-call notes in Derctuo, and it occurred to me that an Emacs clone might be a fun testbed for some of the ideas in there, in particular the use of transactions to guarantee UI responsiveness without presenting a complicated programming model to ad-hoc editing scripts, and providing easier error recovery. If updating the screen, syntax-highlighting text, auto-indenting, and handling keystroke commands are each done in separate transactions, it should be easy to guarantee rapid screen updates, and even in some cases rapid keystroke commands.

The transaction write logging can maybe also be repurposed for undo, which is a pretty essential feature. Emacs's local undo is one of the nicest things about Emacs.

What would I actually need to implement to get an editor I'd use?

Here is some of my view-lossage from Emacs while writing this note, to try to get a handle on which commands I use most and would therefore miss most if they were missing:

```
r e p a i n t i n g SPC t h e SPC s c r e e n M-q SPC
a n d SPC C-a M-d u p d a t i n g M-f M-f , SPC s y
<backspace> n <backspace> y n t a x - h i g h l i t
h <backspace> <backspace> g h t i n g SPC t e x t ,
C-x C-s C-e h a n d l i n g SPC k e y s t r o k e s
SPC a r e SPC d o n e SPC i n SPC s e p a r a t e M-q
SPC t r a n s a c t i o n s , SPC C-x C-s C-p C-e <backspace>
SPC c o m m n d s <backspace> <backspace> <backspace>
a n d s M-q M-f SPC e a c h C-x C-s C-e SPC t h e n
SPC C-x C-s <M-backspace> i t SPC s h o u l d SPC b
e SPC e a s y SPC t o SPC g u a r a n t e e M-q SPC
r a p i d SPC s c r e e n SPC u p d a t e s , SPC a
n d SPC p o s s i b l y SPC e v e n SPC r a p i d SPC
k e y s t r o k e SPC c o m m a n d M-b M-b M-b M-b
<M-backspace> M-f SPC i n SPC s o m e SPC c a s e s
C-e s . M-q C-x C-s M-> <return> <return> C-x C-s C-h
l  C-x o C-SPC
M-< M-w C-x o C-y C-x C-x C-> C-g C-l C-o C-o C-x 1
H e r e SPC i s SPC s o m e SPC o f SPC m y SPC v i
e w - l o s s a g e SPC f r o m SPC E m a c s SPC w
h i l e SPC r u n <M-backspace> w r i t i n g SPC t
h i s SPC d o c u m e n <M-backspace> p a g e <M-backspace>
n o t e : C-x C-s C-h l C-x o C-p
C-M-v C-p C-p C-p C-p C-p C-p C-p C-p C-p C-p C-p
C-p C-s c - h SPC l C-r M-f C-f C-f C-f C-f C-SPC C-n
C-n C-n C-n C-n C-n M-w C-x o M-> SPC C-y M-^ M-^ M-^
M-^ M-^ M-^ C-/ C-/ C-/ C-/ C-/ C-/ C-f <return> C-x
C-x C-e C-f C-> C-x C-s C-x 1 M-v C-g M-{ M-{ C-n C-p
```

C-o <return> C-x C-s T h e SPC t r a n s a c t i o
n SPC w r i t e SPC l o g g i n g SPC c a n SPC a l
s o SPC b e SPC r e p u r p o s e d SPC t <backspace>
f o r SPC C-x C-s C-x b s h <return> C-d g r e p SPC
- i SPC q e m a c s SPC . . / d e r c u <tab> m a r
<tab> * <return> C-x C-f M-p M-p M-n <M-backspace>
<M-backspace> <M-backspace> <M-backspace> <M-backspace>
d e r c u <tab> m a r <tab> q e <tab> m <backspace>
a <tab> <return> C-x 3 C-x b C-s <return> C-h l
p e n d i n g SPC p r o p e r l y , SPC M - q SPC l
e a v i n g SPC y o u SPC i n p <backspace> SPC p l
a c e , M-q SPC r e i <backspace> d i s p l a y SPC
t h a t SPC i s n ' t SPC v i s i l <backspace> b l
y SPC s l o w SPC (!) , SPC M - / , SPC c o n t r
l - b a c k s p a c e , M-q SPC M-b M-b M-f <backspace>
o l C-e c o m m a n d SPC <backspace> - g r a n u l
a r i t y SPC u n d o , M-q SPC p r e f i x SPC a r
g u m e n t s , SPC a n d SPC <M-backspace> <backspace>
<backspace> . M-b M-b a n d SPC C-e SPC S-SPC l s <backspace>
<backspace> A l s o SPC s o m e SPC t h i n g s SPC
I S-SPC u s e d SPC t h a t SPC d i c SPC <backspace>
<backspace> d SPC w o r k : M-q SPC C C-h k C-x C-e
C-x o q C-x o C-x o <backspace> g o t o - l i n e ,
SPC y a n k - p o p . M-b M-b <backspace> <backspace>
SPC a n d SPC C-e <backspace> SPC , s a y <M-backspace>
<backspace> , <backspace> <backspace> , SPC s a y .
M-q C-x C-s C-x 1 C-h l
C-x o C-x o C-x o M-< C-s C - h SPC l C-SPC
M-< M-w C-x o M-{ C-y C-x C-x C-> C-x C-s C-x 1 C-x
C-x C-o C-x C-s M-v M-{ M-{ M-{ M-{ M-{ M-} M-} C-p
M-f M-f M-f M-f M-f SPC m a y b e C-e u n d o , SPC
w h i c h SPC i s SPC p a <backspace> <backspace> a
SPC p r e t t y SPC e s s e n t i a l SPC f e a t u
r e . M-q SPC S-SPC E m a c ' <backspace> s ' s SPC
l o c a l SPC u n d o SPC i s SPC o n e SPC o f SPC
t h e SPC n e <backspace> i c e s t SPC t h i n g s
SPC a b o u t SPC E m a c s . M-' M-q C-x C-s <next>
<next> <prior> <prior> <prior> M-{ M-{ M-{ M-{ M-}
C-p C-x b t r a n C-g C-x C-f <M-backspace> <M-backspace>
d e r t <backspace> c t u <tab> m a r <tab> r a n <M-backspace>
t r a n s <tab> <return> C-x C-= C-= C-= <next> M-x
M-p <return> c d SPC . . / d e r c t u <tab> <return>
l s <return> f i r e f o <tab> SPC d e r c t <tab>
- 0 2 <tab> <tab> 1 / n o t <tab> t r a n s <tab> &
<return> C-x 1 C-x b <return> C-x b C-s <return> C-h
l
C-n C-n C-n C-n C-n C-n C-n C-n C-n C-n C-n C-n C-n
M-w C-x o M-> M-{ M-{ M-} C-y C-x C-x C-> C-g C-d C-x
C-s C-x 1 C-v M-} M-{ M-} C-p C-p C-p C-p C-p C-o M-{
C-p C-e <backspace> , SPC t o SPC t r y SPC t o SPC
g e t SPC a SPC h a n d l e SPC o n SPC w h i c h SPC
c o m m a n d SPC <backspace> s SPC I S-SPC u s e M-q
SPC m o s t SPC a n d SPC w o u l d SPC t h e r e f
o r e SPC m i s s SPC m o s t SPC i f SPC t h e y SPC
w e r e SPC m i s s i n g : M-q C-x C-s C-x 2 C-x C-f

<M-backspace> <M-backspace> d e v 3 / d e c o d e -
l <backspace> <backspace> l o s s a g e . p y <return>
! / u s r / b i n / p y t h o n 3 <return> " " <backspace>
<backspace> " " " D e c o d e SPC e m a c s SPC C-x
C-s C-h k C-h l C-x o v i e w - l o s s a g e . C-x
o C-x o C-n C-n C-n C-n C-n C-n C-n C-n C-e C-b C-b
C-b <return> C-x o C-x o C-v C-v C-v C-v M-x o p e
n - d r i <tab> <return> l o s s a g e . <backspace>
M-b t m p . <return> C-h l
C-x o C-r C-r C-r M-f M-f C-f C-f C-SPC C-n C-n C-g
C-r C-r C-r M-< C-SPC M-> C-p C-p M-w C-x o C-x o M->
M-{ C-y C-x C-x C-> C-x C-x C-g C-x C-f t m p . l <tab>
<return> C-n C-n C-x k <return> C-x o q C-x o <backspace>
SPC o u t p u t . " " " <return> C-x C-s M-b M-b M-f
M-f SPC w i t h SPC a n SPC a s s u m e d SPC k e y
m a p C-x C-s M-x d e s c r i b - e <backspace> <backspace>
e - k e y <tab> m <tab> <backspace> <backspace> <backspace>
<backspace> c u r <tab> k e y <tab> m <backspace> <backspace>
<backspace> <backspace> m <M-backspace> <M-backspace>
m <tab> o d <tab> <return> C-x o C-x o C-x o C-v C-v
C-v M-v C-v C-v C-v C-v M-v M-v M-v q C-h l
M-< C-s C - h SPC l C-s C-x C-x C-g
C-x C-x C-g C-r C - h SPC <backspace> C-p C-p C-p C-p
C-p C-p C-p C-p C-p C-a C-p C-p C-f C-f C-SPC C-n C-n
C-n C-n C-n C-n C-n C-n C-n C-n C-n C-e C-f M-w C-x
o C-x o C-y C-x C-x C-> C-x C-s C-x o q C-x o C-x o
C-x o C-g M-v M-v M-v M-v M-v M-v C-x o C-e <return>
<return> w o r <M-backspace> d e f SPC w o r d s M-(
i n f i l e C-e : <return> <tab> f o r SPC l i n e
SPC i n SPC i n f M-/ : <return> <tab> f o r SPC w
o r d SPC i n SPC l i n e . s p l i t ( ) : <return>
<tab> y i e l d SPC w o r d <return> C-x C-s <return>
<return> i f SPC _ _ n a m e _ _ SPC = = SPC ' _ _
m a i n _ _ ' : <return> <tab> p r i n t (l i s t
(w o r d s (s y s . s t d i n))) M-< M-} C-o C-o
i m p r o <backspace> <backspace> o r t SPC s y s C-x
C-s C-h l
C-x o C-x o C-r C-e C-g C-x o C-r
C - h SPC l C-r M-f M-f C-f C-f C-SPC M-> C-p C-p M-w
C-x o C-x o M-} C-y C-x C-x C-> C-g C-x C-s C-d C-x
C-s C-x o q C-x o C-x b C-g M-x M-p C-g M-o C-g M-x
s h e l l <return> c d SPC . . / d e v 3 <return> p
y t h o n SPC d e c o d e l <tab> <return> C-x o C-x
b <return> M-{ C-SPC M-} M-w C-x o C-y C-d <return>
C-d <return> C-c C-d <return> C-d C-d C-c C-p C-v C-v
C-v C-v C-v C-v M-v C-h l
C-x o
C-x o C-x o C-r C-r C-r M-f M-f M-f M-f M-b C-SPC C-n
C-n C-n C-e C-n C-n C-n C-n C-n M-w C-x o C-p C-n C-y
C-x C-x C-> C-l C-x C-s C-x 1 C-x C-x C-o C-f C-x C-s
C-x 2 C-x b C-s C-s <return> M-} M-} C-o <return> p
r e f i x e s SPC = SPC (' C - x ' M-b C-b C-b C-b
(C-e ,) M-b C-b C-b C-b C-b <backspace> [C-e SPC
(' C - c ' ,) , SPC C-x C-s C-x o M-v <prior> <prior>
<prior> <prior> <prior> <prior> <prior> <prior> <next>

<next> C-x o (' E S S <S-backspace> C ') <backspace>
, ' <backspace>)) <backspace>] C-f C-s C-g C-x C-s
C-n C-n C-n C-o C-k C-o <tab> m a i n (s y s s <backspace>
. s t d i n) C-f C-k C-k C-p C-p C-p C-o C-o d e f
SPC m a i n (i n p u t) <backspace> <backspace> <backspace>
<backspace> f i l e) : <return> C-y C-k C-x C-s M-x
c o m p C-g C-x o M-v M-v C-v C-n C-SPC C-v C-n C-n
C-n C-n C-n M-w C-x o M-x M-p <return> M-> C-p M->
M-p M-p <return> C-y C-/ C-x b <return> C-r) SPC C-f
, C-x C-s C-x g q C-h l
C-r C-r <return> M-> C-r C
- h SPC l C-r C-f M-f M-f C-SPC M-> C-p C-e C-p C-e
C-p C-e M-w C-x o C-x o M-} C-y C-x C-x C-> C-d C-x
C-x C-o C-x C-s C-x o q C-x b <return> M-p <return>
C-x o C-f C-SPC M-2 M-0 C-p M-w C-x o C-y <return>
C-d C-d C-x o C-h l
C-x o C-x o C-x o C-r
C-r C-r M-f M-f M-f M-b M-f C-SPC C-n C-n C-n C-n C-n
M-w C-x o C-x o M-} C-y C-x C-x C-> C-d C-x C-s C-x
o C-x o C-x o C-x b C-g C-n C-n C-n C-n C-n C-n C-o
C-x C-s C-x o C-x o C-x 3 C-x b <return> C-n C-n <return>
w <backspace> <tab> f o r SPC w o r d SPC i n SPC M-d
M-d C-d C-e <backspace> <backspace> : <return> C-p
C-o <tab> p r e f i x SPC = SPC ( ) C-n C-n C-e <tab>
C-x C-s p r e f i x SPC = SPC p r f <backspace> e f
i x SPC + S-SPC <M-backspace> <backspace> <backspace>
+ = SPC (w o r d ,) <return> <tab> i f SPC p r e
f i x SPC n o t SPC i n SPC p r e f i x e : <backspace>
s : <return> <tab> p r i n t (p r e f i x) M-b '
SPC ' . j o i n (C-e) <return> <tab> p r e f i x
SPC = SPC ( ) C-x C-s C-h l
C-x o C-x o C-x o C-r C-r C-r M-f M-f
M-f C-SPC M-} M-w C-x o C-x o C-x o C-y C-x C-x C->
C-d C-x C-s C-v C-x o C-x o C-f C-p C-p C-p C-n C-n
C-n C-o <tab> <backspace> <backspace> p r i n t ('
SPC ' . j o i n (p r e f i x)) C-a C-o C-x C-s C-x
o M-p M-p <return> M-p M-p <return> C-d C-d M-v M-v
M-v M-v M-v M-v M-v M-v M-v C-x o C-x o C-x o C-r [
C-d { C-e <backspace> } C-x C-s <return> <return> c
m d s SPC = SPC { <backspace> <backspace> <backspace>
<backspace> t r SPC SPC <backspace> = SPC ' ' ' <return>
C - v SPC C-h k C-v s c r o l l - u p <return> M -
v SPC s c r o l l - d o w n C-h f <return> C-p C-p
C-p C-p C-e C-b , SPC (' C - h , <backspace> ' ,)
C-x C-s C-x o C-x o C-x o C-x o C-n C-n C-n C-n <return>
C - h SPC l SPC v i e w - l o s s a g e <return> C-x
C-s C - x SPC o SPC o t h e r - w i n d o w C-h f <return>
<return> C - r SPC i s e a r c - b a <backspace> <backspace>
<backspace> h - b a c k w a r d C-h f <return> <return>
' ' ' C-x C-s C-h l
C-h f <return> C-h k M-w <M-backspace> <M-backspace>
<M-backspace> <M-backspace> k i l l - r i n g - s a
v e C-x C-s <return> C - p SPC p r e v i o u s - l
n e C-x C-s <backspace> <backspace> i n e C-x C-s <return>
C - y SPC y a n k C-h f <return> <return> C - x SPC

C - x SPC C-h k C-x C-x e x c h a M-/ C-x C-s C-h k
C-x 1 <return> d e <backspace> <backspace> C - x SPC
1 SPC d e l e t e - o t M-/ - w M-/ <M-backspace> w
i n d o w s C-x C-s C-n <return> c m d s SPC = SPC
{ w o r d [<backspace> s [M-b ' SPC ' . j o i n (
C-e : - 1]) : S-SPC w o r d s [- 1] <return> <tab>
f o r SPC l i n e SPC i n SPC C-p C-a C-o C-n C-n C-e
c m d s M-/ . s p l i t (\ <backspace> ' \ n ' (<backspace>
) <return> <tab> f o r SPC w o r d s SPC 9 <backspace>
i n SPC [l i n e . s p l i t ( )] <return> <tab>
f o r SPC <M-backspace> i f SPC C-x o M-> p y t h o
n 3 <return> ' \ n ' . s p l i t ( ) <return> M-p M-b
M-b C-b SPC <return> C-x o w C-g C-/ C-x o C-x o w
o r d s } C-x C-s C-h l
C-x o C-x
o C-x o C-r C-r C-r M-< C-f C-f C-SPC M-> C-p C-p M-w
C-x o C-x o C-x o C-y C-x C-x C-> C-g C-x C-s C-x C-x
C-g C-x o C-x o C-f C-v C-n C-l C-n C-n C-o <tab> c
m d SPC = SPC M-d C-d C-e <backspace> <return> C-k
C-n C-o <tab> i f SPC <M-backspace> p r i n t (p r
e f i x , SPC c m d s . g e t (p r e f i x , SPC '
s e l f - i n s e r t - c o m m a n d) <backspace>
')) C-x C-s C-x o C-d M-p M-p M-p M-p M-p <return>
M-p M-p M-p M-p M-p <return> C-d C-c C-p C-v C-x o
C-x o C-x o C-a M-f M-f <M-backspace> c m d M-f M-f
M-f <M-backspace> c m d s <backspace> C-x C-s C-x o
M-> C-d M-p M-p <return> M-p M-p <return> C-d C-c C-c
C-c C-p C-v <next> <down> <down> <down> <down> <down>
<down> <down> <down> <down> <down> <down> <down> <down>
<down> <down> <down> <down> <down> <down> C-x o C-x
o C-h l
C-r C-r C-r M-f M-f M-f C-SPC M-> C-p
C-p M-w C-x o C-x o C-x o C-y C-x C-x C-> C-d C-x C-s
C-x o C-x o M-{ C-x C-s C-p C-p C-p C-p C-n C-n C-n
C-n C-n C-n C-n C-n C-n C-n C-n C-n C-n C-n C-p C-p
C-e M-b M-b M-b M-d M-d M-d ? ? ? C-x C-s C-x o M->
M-p M-p <return> M-p M-p <return> C-d C-d C-c C-p C-v
C-v C-v C-l C-x o C-x o C-x o C-h k C-> M-v C-l C-n
C-n C-n C-o C - > SPC i n d e n t i <backspace> - r
i g i d l y - 4 <return> C - l SPC C-h k C-l r e c
e n t e r - t o p - b o t t o m C-x C-s <return> C
- x SPC C s - <backspace> <backspace> - s SPC C-h k
C-x C-s s a v e - b u f f e r C-x C-s <return> C -
o SPC C-h k C-o o p e n - l i n e <return> C - f SPC
C-h k C-f f o r w a r d - c h a r C-x C-s C-h k C-x
2 <return> C - x SPC 2 SPC s p l i t - w i n d M-/
<backspace> - M-/ M-/ <M-backspace> v e r M-/ <backspace>
t i c a l l y C-x C-s C-h l
C-x o C-x
o C-x o C-r C-r C-r M-f M-f M-f C-SPC M-> M-p C-p C-p
M-w C-x o C-x o C-x o M-} C-y C-x C-x C-> C-d C-x C-s
C-x o C-x o <return> C - s SPC i e <backspace> s e
a r c h - f o r w a r d C-h k C-s C-h k <return> <return>
< r e t u r n > S-SPC n e w l i n e C-x C-s C-h k M-}
<return> f o r w a r d - p a r a g r a p h C-a M -

} S-SPC C-x C-s C-M-v C-v C-n C-n C-n C-n C-n C-n C-n
C-o <return> f o r SPC c h r SPC i n SPC <backspace>
<backspace> <backspace> <backspace> <backspace> <backspace>
<backspace> _ c SPC i n SPC r a n g e (3 3 , SPC 1
2 7) : <return> <tab> c m d s [c] SPC = SPC ' s
e l f - i n s e r t - c o m m a n d ' C-x C-s C-h
l C-x o C-x o C-x o C-r C-r M-f
M-f M-f C-SPC M-} M-w C-x o C-x o C-x o M-} C-y C-x
C-x C-d C-x C-x C-> C-f C-g C-x C-s C-x o C-x o C-x
o M-> M-p M-p <return> M-p C-x o C-x o C-x o C-a M-f
C-f _ C-x C-s C-x o <return> M-p M-p M-p <return> C-d
C-c C-c C-x o C-x o C-x o C-b c h r (C-f C-f) C-x
C-s C-x o M-p M-p <return> M-p M-p <return> C-d C-c
C-c C-c C-p C-v C-v <next> C-h k C-x b C-x o C-x o
C-x o M-< C-v M-} C-p C-o C - x SPC b SPC i s w i t
c h - b u <backspace> <backspace> <backspace> b - b
u f f e r C-x C-s <return> S P C SPC C-h k SPC s e
l f - i n s e r t - c o m m a n d C-x C-s C-M-v <return>
C - b SPC b a c k w a r d - c h a r C-x C-s C-h k C-v
C-h k C-b C-h l C-x
o C-x o C-x o C-r C-r C-r M-f M-f M-f C-SPC M-> C-p
C-p M-w C-x o C-x o C-x o C-p C-y C-x C-x C-d M-^ C-n
C-a C-SPC M-} C-> C-g C-x C-s M-v C-x o C-x o C-M-v
<return> < p r i o r > S-SPC s c r o l l - u p C-h
k <prior> <backspace> <backspace> d o w n C-x C-s <return>
< n e x t > S-SPC s c r o l l - u p C-h k <next> C-M-v
C-h k <S-backspace> <return> < S - b a k c p s <backspace>
<backspace> <backspace> <backspace> c k s p a c e >
SPC p <backspace> C-x o C-h k <S-backspace> C-x o C-h
k <S-backspace> C-x o C-x o d e l e t e - b a c k w
a r d - c h a r C-x C-s <return> C - g SPC q u i t
C-h k C-g M-b k e y b o a d <backspace> r d - C-x C-s
C-e <return> H <backspace> C-h k C-k C - k SPC k i
l l - l i n e C-x C-s <return> C-h k <tab> < t a b
> S-SPC i n d e n t - f o r - t a b - c o m m a n C-h
C-g d C-x C-s C-h l C-x o C-x o C-x o C-r C-r C-r M-f M-f M-f C-SPC
M-} M-w C-x o C-x o C-x o M-} C-p C-e SPC C-y C-x C-x
C-d C-x C-x C-x C-x C-n C-e C-a C-> C-x C-s C-v M-}
C-x o C-x o C-M-v C-h k <backspace> C-x o C-h k <backspace>
C-x o C-x o C-x o C-p C-p C-p C-a C-k C-k C-y C-y C-p

M-f M-f M-b <M-backspace> C-x C-s C-h l C-x o C-x o C-x o C-r C-r C-r M-f M-f M-f
C-SPC
C-SPC M-} C-x C-x M-w C-x o C-x o C-x o C-p C-e C-y
C-x C-x C-n C-a C-n C-> C-x C-s C-n C-g C-x o C-x o
C-M-v C-M-v C-h k M-x C-n C-n C-n C-e <return> M -
x SPC e x e c u t e - e x t e n d e d - c o m m a n
d C-x C-s C-M-v C-h k M-p C-x o C-h k M-p C-x o C-x
o C-x o <return> M - p SPC c o m i n t - p r e v i
o u s - i n p u t C-x C-s <return> C - / SPC u n d
o C-h l C-h k M-/ C-h k C-/ <return> M - / SPC d a
b b r e v - e x p a n d C-x C-s C-h l
C-x o C-x o C-x o C-r C-r C-r C-r M-f C-f C-f C-f C-f
C-SPC M-} M-w C-x o C-x o C-x o C-n C-n C-n C-n C-n
C-e C-y C-x C-x C-x C-x C-x C-x C-n C-f C-> C-g C-x

C-s C-x C-x C-x C-x C-g C-u C-SPC C-u C-SPC C-u C-SPC
C-u C-SPC C-x o C-x o C-h k C-x g <return> C - x SPC
g SPC m a g i t - s t a t u s C-x C-s C-M-v C-x o M->
M-p M-p <return> M-p M-p <return> C-d C-c C-c C-h k
M-> C-x o C-x o C-x o <return> M - > S-SPC e n d -
o f - b u f f e r C-x C-s C-x o C-r ? ? C-r C-r M->
C-x o C-x o C-h l
C-r C-r <return> C-r c - h SPC l C-r C-n
C-a C-SPC M-} M-w C-x o C-x o C-x o M-} C-y C-x C-x
C-> C-g C-x C-s C-x o C-x o C-v C-n C-n C-n C-n C-n
C-n C-n C-n C-n C-p C-p C-p C-p C-p C-o <tab> f r e
q s SPC = SPC { } C-f C-n C-x C-s C-n C-p C-p C-o C-n
C-x C-s C-n C-n C-n C-p C-n C-n C-n C-n C-o <tab> M-{
C-o <tab> u n k o n w n s <backspace> <M-backspace>
u n k n o w n s SPC = SPC s e t ( ) M-} M-} C-p C-o
<tab> C-a C-k C-n C-o <tab> i f SPC c m d SPC i n SPC
c m d s : <return> <tab> f r e q <M-backspace> n a
m e SPC = SPC c m d C-p C-p C-p C-n C-n C-n C-p C-p
M-b C-M-S-SPC C-M-S-SPC C-M-S-SPC C-w n a m e C-p <return>
<tab> n a m e SPC = SPC C-y C-n C-n C-a k <tab> i M-b
C-k i f SPC n a m e SPC n o t SPC i n SPC f r e q s
: <return> <tab> f r e q s [n a m e] SPC = SPC 0
<return> <tab> <backspace> f r e q s [n a m e] S-SPC
+ = SPC 1 C-f C-k <return> <tab> i f SPC n a m e SPC
= <M-backspace> c m d SPC n o t SPC i n SPC c m d s
: <return> <tab> u n k n o w n s . a d d (c m d)
C-h l
C-x o C-x o C-x o C-r C-r C-r M-f M-f
M-f C-SPC M-> C-p C-p M-w C-x o C-x o C-x o M-} C-y
C-x C-x C-> C-d C-x C-s C-x o C-x o C-f C-k C-x C-s
C-n C-n C-o <return> <tab> f o r SPC n a m e SPC i
n SPC s o r t e d (f r e q s . k e y s ( ) , SPC k
e y = f r e q s . g e t M-b C-e , SPC r e v e r s e
= T r e u e) <backspace> <backspace> <backspace> <backspace>
u e) : <return> <tab> p r i n t (n a m e M-b ' %
8 d SPC % s ' SPC % S-SPC (f r e q s [n a m e , SPC
<backspace> <backspace>] , SPC C-e)) <return> <return>
<tab> p <backspace> <backspace> p r i n t (" u n k
n o w n s : " " <backspace> , SPC " , SPC " , <backspace>
. j o i n (u n k M-/)) C-x C-s C-h l
C-x o C-x o C-x o C-r C-r C-r M-f M-f M-f
C-SPC M-} M-w C-x o C-x o C-x o M-} C-y C-x C-x C->
C-d C-x C-s C-x o C-x o C-x o M-p M-p <return> M-p
M-p <return> C-d C-d M-v C-x o C-x o C-x o C-r r e
v M-d M-d <backspace> <backspace> C-x C-s C-x o C-x
o M-} C-SPC M-{ M-w C-x o C-x o C-x o M-> M-p M-n C-x
o C-x o C-x M-< C-g C-x o M-< C-n C-n C-o f r o m SPC
_ - f u t u r <M-backspace> <backspace> _ f u r u t
<backspace> <backspace> <backspace> t u r e _ _ S-SPC
i m p o r t SPC p r i n t _ f u n c t i o n C-x C-s
C-x o M-p M-p <return> C-x o C-SPC M-} M-w C-x o C-x
o C-x o C-y <return> C-d C-d M-v C-v C-h l
k C-d C-x o C-x o C-x o C-x o C-x o C-x o C - d SPC
d e l e t e - c h a r C-h k C-a <return> C - a SPC
m o v e - b e g M-/ - M-/ o f - l i n e C-x C-s C-h

k S-SPC <return> S - S P C SPC s e l f M-/ <return>
C - w SPC k i l l - r e g i o n C-h f <return> C-x
C-s <return> C-h k C-x C-f C - x SPC C - f SPC f i
n d - f i l e <return> C - x SPC C - e SPC C-h k C-z
C-h k C-x C-r C-h k C-x C-e e v a l - l a s t - s e
x p <return> C-x C-s C - u SPC C-h k C-u u n i v e
f s <backspace> <backspace> r s a l - a r g u m e n
t C-x C-s C-h k C-c C-d C-x o C-h k C-c C-d C-x o C-x
o C-x o <return> C - c SPC C - d SPC c o m i n t -
s e n d - e o f <return> C-x C-s M - < S-SPC C-h k
M-< b e g i n M-/ - f M-/ <M-backspace> o M-/ - M-/
M-/ C-x C-s <return> C-h k C-c C-c C-x o C-h k C-c
C-c C-x o C-x o C-x o C - c SPC C - c SPC c o m i n
t - i n t e r r u p t - s u b j o b C-h l
C-x C-s C-x o C-x o C-r C-r C-r M-< C-SPC M-} M-w C-x
o M-} C-y C-x C-x C-> C-g C-x C-s C-x o <return> C-h
k M-0 M - 0 SPC d i g i t - a r g u m e n t <return>
<f3> M - <f4> <backspace> <backspace> <backspace> <f3>
<return> M - <f3> SPC d i g i t - a r g u m e n t <f4>
<f4> <f4> <f4> <f4> <f4> <f4> <f4> <f4> <f4> C-a C-p
C-p C-p C-p C-p C-p C-p C-p C-p C-p C-k C-k C-SPC M-}
C-p M-w C-y C-x C-x M-% M - <return> C - <return> !
C-x C-s C-h l
s - p r o m p t <return> C-x C-s C-h k M-(M - (SPC
i n s e r t - p a r e n t h e s e s C-x C-s <return>
C - x SPC k C-h k C-x k SPC k i l l - b u f f e C-x
C-s <return> <backspace> r <return> C - x SPC C = <backspace>
- = SPC C-h k C-x C-= t e x t - s c a l e - a d j u
s t C-x C-s <return> M - ^ SPC C-h k M-^ d e l e t
e - i n d e n t a t i o n C-x C-s C-h k <down> <return>
< d o w n > S-SPC n e x t - l i n e C-x C-s <return>
C-h k C-x 3 C - x SPC 3 SPC s p l i t - w i n d M-/
- h o r i z o n t a l l y C-x C-s C-h k C-= C-x o C-x
o C-x o M-} C-r C - = C-h k C-M-v C-g C-h k C-M-v C-x
o <return> C - M - v SPC s c r o l l - o t h e r -
w i n d o w C-x C-s C-h k M-' <return> M - ' SPC s
m a r t - a p o s t r o p h e C-x C-s <return> < M
- B <backspace> b a c k s p a c e S-SPC <backspace>
> S-SPC C-h k <M-backspace> b a c k w M-/ - M-/ M-/
M-/ <return> C-h k M-{ C-h l
C-M-v C-g C-h k C-M-v C-x o <return> C - M - v SPC
s c r o l l - o t h e r - w i n d o w C-x C-s C-h k
M-' <return> M - ' SPC s m a r t - a p o s t r o p
h e C-x C-s <return> < M - B <backspace> b a c k s
p a c e S-SPC <backspace> > S-SPC C-h k <M-backspace>
b a c k w M-/ - M-/ M-/ M-/ <return> C-h k M-{ C-h
l C-x o C-x o C-SPC C-r C-r C-g C-SPC C-g C-r c - h
SPC l C-r M-{ C-SPC M-} M-w C-x o C-y C-x C-x C-> C-g
C-x o C-h k M-{ M - { S-SPC b a c k w a r d - p a r
a g r a p h C-x C-s <return> C - h SPC f SPC d e s
c r i b e - f u n c t i o n <return> C - h SPC k SPC
d e s c r i b e - k e y <return> M - q SPC f i l l
- p a r a g r a p h C-h k M-q C-x C-s <return> M -
o SPC C-h k M-o C-g C-g s h e l l C-x C-s <return>
C-h k M-n C-x o C-h k M-n C-x o C-x o C-x o M - n SPC

c o m i n t - n e x t - i n p u t <return> M - d C-h
k M-d SPC k i l l - w o r d C-x C-s C-h k C-g C-h
l
C-x o C-x o C-r C-r <return> C-r c - SPC h <backspace>
<backspace> h SPC l l l <backspace> <backspace> M-<
C-SPC M-} M-w C-x o M-} C-y C-x C-x C-> C-g C-x C-s
C-x o C-x o C-x o C-x o M-} C-SPC M-{ M-w C-x o C-x
o M-> M-p M-p <return> C-y <return> C-d C-d C-x o C-x
o C-x o C-h k C-h C-g C-h k C-x M-< C-h k C-x C-r C-h
k <f3> <return> < f 3 > S-SPC k m a c r o - s t a r
M-/ <backspace> <backspace> - M-/ m a c r o - o r -
i n s e r t - c o u n t e r <return> M - S-SPC <backspace>
% C-h k M-% SPC q u e r y - r e p l a c e <return>
< f 4 > S-SPC C-h k <f4> k m a c r o - e n d - o r
- c a l l - m a c r o C-x C-s C-h l
t a l , SPC C-x C-s M-x M-p <return> M-> M-p M-p <return>
M-p M-p <return> C-d C-d M-v C-x 2 C-x b C-s <return>
M-} <return> S o m e SPC n o t e s SPC o n SPC f r
e q u e n c i e s SPC o f SPC c o m m a M-b M-b M-b
m o s t SPC c o m m o n C-k SPC c o m m a n d SPC f
r e q u e n c i e s : <return> <return> C-x C-s C-x
o M-v C-p C-p C-p C-SPC C-n C-n C-n C-n C-n C-n C-n
C-n C-n C-n C-v C-v C-v M-v M-v C-p M-v C-n C-n C-n
C-n C-n C-n C-n M-w C-x o C-y C-x C-x C-> C-g C-x C-s
C-v C-x 1 C-l M-{ C-p C-e <backspace> . SPC S-SPC O
u t SPC o f SPC C-x C-s C-x b <return> M-> C-x b <return>
5 8 8 2 SPC c o m a <backspace> m a n d s SPC r e c
o r d e d SPC a b o v e SPC a n d SPC <backspace> SPC
s u c c s s <backspace> <backspace> e s s f u l l y
M-q d e c o d <backspace> <backspace> <backspace> <backspace>
<backspace> SPC d e o c <backspace> <backspace> c o
d e d , SPC t h e s e SPC C-f C-n C-SPC M-} M-= C-g
M-{ C-p C-e 2 4 SPC c a <backspace> <backspace> a c
c o u n t SPC f o r SPC C-v C-l M-(M-: M-(* S-SPC
5 8 8 2 SPC . 9 <return> C-/ C-h l
s SPC o b v i o u s l y SPC i s n ' t SPC e v e r y
t h i n g SPC e s s e n t i a l ; SPC M-' C-x C-s i
t ' s SPC m i s s i n g , SPC a m o n g SPC t h e r
SPC t h i n g M-b M-b o C-e s , SPC f i n d - f i l
e SPC a n d M-q SPC C-x C-s <M-backspace> <backspace>
, SPC C-r k i l l C-g C-x b C-s <return> M-v C-l C-x
b <return> y a n k , SPC k i l l - r i n g - s a v
e SPC M-(t h e SPC n e w SPC c o p y - r e g M-/ i
o n - a s k <backspace> - k i l l C-e . C-x C-s C-x
b <return> C-x b <return> <backspace> , SPC C-h k M-:
e v a l - e x p r e s s i o n , M-q SPC C-x C-s C-x
1 M-v <down> <down> C-n C-n C-n C-n C-n C-l C-n C-n
C-n C-n C-n C-n C-n C-n C-n C-e M-{ M-{ M-{ C-n C-n
M-f M-f M-f M-f M-b (<backspace> M-f M-f M-f SPC b
y SPC a SPC j a n k y SPC e r r o <M-backspace> u n
r e l i a b l e SPC c <backspace> s c r i p t M-q M-f
<backspace> <backspace> SPC f l l o w i n g <M-backspace>
f l <M-backspace> <backspace> s e C-x C-s C-h l
C-x o C-r C - h SPC l
M-< C-SPC M-} M-w C-x o M-{ C-y C-x C-x C-> C-g C-v

M-} C-n C-n M-b M-d C-d M-f SPC o f SPC t h e M-q C-x
C-s C-x 1 C-v C-l C-x b C-s <return> <down> <up> <up>
<up> <up> C-x b <return> C-x b <return> <down> <down>
<down> <down> <down> <down> <S-down> <S-down> <S-up>
C-g C-x 2 C-x b <return> M-} C-n C-p C-n M-f M-f M-f
M-f M-f M-f M-f M-f M-f C-o C-o SPC s u c h SPC b a
s i c s SPC a s SPC f o r w a r d - c h a r , M-d M-d
M-d C-d M-q C-n C-x o C-s e x t C-g C-x o C-e SPC n
d SPC e M-b M-b a C-e x e c u t e - e x t e n d e d
- c o m m a n d . C-r M - n C-r C-r C-r C-x C-f C-h
k M-n q C-g C-x b <return> M-v M-v C-r M - n M-f C-f
C-f C-f C-k n e x t - h i s t o r y - e l e m e n t
C-r M - p M-f C-f C-f C-f C-k C-x C-f C-h k M-p q C-g
p r e v i o u s - h i s t o r y - e l e m e n t C-x
C-s C-x b <return> C-x 1 C-v C-v C-v C-h l
C-x o C-r C-r M-> C-r c - h SPC l C-r M-f M-f
M-f C-SPC M-} M-w C-x o M-} C-y C-x C-x C-> C-d C-l
C-v C-x C-s M-} M-} M-} C-x C-x C-g C-x o C-x b <return>
C-u C-SPC C-n C-n C-n C-n C-n C-n C-n C-n C-n C-n C-n
C-n C-v C-l C-p C-p C-p C-p C-p C-p C-o <tab> u s e
d _ k e y s SPC = SPC { } <return> <tab> C-a C-k C-x
C-s C-n C-n C-n C-n C-n C-n C-n C-n C-n C-n C-n C-n
C-n C-n C-o C-o <tab> <backspace> C-a C-k C-o <tab>
<backspace> i f SPC c h d <backspace> <backspace> m
d SPC n o t SPC i n SPC u s M-/ : C-a C-o C-n C-e <return>
<tab> C-p M-b M-d n a m e C-n <tab> u s e M-/ [n a
m e] S-SPC = SPC c m d C-f C-k C-x C-s C-n C-n C-n
C-n C-n C-n C-e C-a M-f M-f M-f M-f SPC (e . g . ,
SPC % s) C-e C-b C-b , SPC u s M-/ [n a m e] C-e
C-x C-s C-h l
C-x o C-r C-r C-r
M-f M-f C-f C-f C-SPC M-} M-w C-x b <return> C-y C-x
C-x C-d C-x C-x C-> C-g C-x C-s C-SPC M-{ M-w M-x M-p
<return> M-> M-p M-p <return> C-y <return> C-d C-d
C-h k <S-up> C-x o C-x b C-s <return> M-v C-l C-n C-n
C-n C-n C-n C-n C-n C-o M - : S-SPC e v a l C-h k M-:
- e x p r e s s i o n C-x o C-x 2 C-x b C-s <return>
C-x o C-x o <return> < S - d o w n > S-SPC C-a C-k
C-k C-p C-p C-p C-p C-p C-p C-p C-p C-p C-p C-p C-p
C-p C-p C-p C-p C-y C-p C-p C-e C-n n e x t - l n e
<backspace> <backspace> i n e C-x C-s C-r u p C-a C-r
C-r C-v C-s C-s C-n C-n C-n C-n C-n C-y C-p C-e <M-backspace>
u p S-SPC <S-backspace> > SPC p r e v M-/ - M-/ M-/
<M-backspace> <M-backspace> C-h k <up> p r e v i o
u s - l i n e <return> < u p > SPC p r e v M-/ - M-/
C-x C-s C-h l
t SPC C-x C-s i f SPC y o u SPC h a v e SPC C-a C-k
T h i s SPC i s SPC p r o b a b l y SPC m o s t SPC
o f SPC w h a t SPC C-a C-k S o SPC w i t h SPC t h
o s e SPC c o m m a n d s , SPC y o u ' d SPC h a v
e SPC <M-backspace> <M-backspace> <M-backspace> i f
SPC m y SPC c u r r e n t SPC e d i t i n g SPC s e
s s i o n SPC w e r e SPC t y p i c a l SPC M-(w h
i c h SPC i t SPC p r o b a b l y SPC i s n ' t M-'
C-e , SPC M-q SPC y o u ' d SPC <M-backspace> <M-backspace>

I ' d SPC r u n SPC M-' i n t o SPC a C-p C-a M-f M-d
SPC i f SPC y o u SPC h a d M-f M-f SPC i m p l e m
e n t e d C-a M-f M-f M-d SPC I M-q C-n C-e C-x s SPC
C-p M-b C-n C-e SPC m i s s i n g SPC s t a i r SPC
o <backspace> a b o u t SPC o n c e SPC o u t SPC o
f SPC e v e r y SPC 1 0 0 SPC c h a r a c t e r <M-backspace>
k e y s t r o k e s . M-q C-x C-s C-h l
m a j o r i t y SPC o f SPC m y SPC k y <backspace>
e y s t r o k e s , SPC M-l M-q C-x C-s C-l C-l C-l
<next> <prior> <down> C-e M-b M-b M-b M-b a n d SPC
h a v i n g SPC h <backspace> j u s t SPC t h e s e
SPC i m p l e m e n t e d SPC o u <M-backspace> w o
u l d SPC m a k e SPC s o m e t h i n g SPC " f e e
l SPC l i k e SPC a n SPC E m a c s " , SPC M-" M-"
M-q C-x C-s <next> <next> <up> <next> <prior> <prior>
<prior> M-> <backspace> , SPC b u t SPC i t SPC s e
e m s SPC l i k e SPC i f SPC I S-SPC i m p l e m e
n t e d SPC * m y * S-SPC M-q SPC i <backspace> s t
a n <M-backspace> m o s t SPC u s e d SPC E m a c s
SPC c o m m a n d s SPC i n s t e a d SPC o f SPC B
e l l a r d ' s , SPC M-' t h e SPC s e t SPC w o u
l d SPC b e SPC e v e n SPC s m a l l e r . M-b M-b
M-b M-b t o SPC g e t SPC t o SPC c o m f o r t SPC
M-q C-x C-s <next> <prior> <prior> <prior> C-h l
C-n C-h f <return> C-n C-h f <return> C-n C-n C-n C-n
M-f M-f M-f C-h f <return> C-n C-h f <return> C-n C-h
f <return> C-p C-h f <return> C-n C-h f <return> C-x
0 a n d SPC d o d <backspace> w n c a s e - w o r d
SPC (! ?) . M-q C-x C-s C-x 1 C-l C-p C-p C-p C-p
C-p M-b M-b <M-backspace> <M-backspace> m o s t SPC
M-q C-x C-s <prior> <next> C-n C-n C-n C-n C-n C-n
C-n C-n C-n C-n C-n C-n M-f M-f M-f M-f SPC E m a <M-backspace>
G N U S-SPC E m a c s SPC i m p l e m e n t s SPC m
o s t M-f M-f M-f M-d M-d M-d C-e M-f M-f C-a M-d M-d
M-d <backspace> M-q M-> C-x C-s C-h f r e c e n t e
r <return> C-x 1 <backspace> S-SPC S-SPC S o m e SPC
o f SPC t h e m SPC <backspace> , SPC l i k e SPC r
e c e n M-/ , M-q SPC a r e SPC t h i n SPC L i s p
SPC v e n e e r s SPC o n SPC t o p SPC o f SPC b u
i l t - <M-backspace> p r i m i t i v e SPC f u n c
t i o n s SPC l i k e SPC r e c e n t e r . M-b M-b
M-d s u c h SPC a s M-q M-> C-x C-s <prior> <prior>
<prior> <prior> <prior> <prior> <prior> <prior> <prior>
<prior> <prior> <prior> <prior> <prior> <prior> C-h
l C-p C-p C-p C-p C-p C-p C-p C-p C-p C-p C-n C-a M-f
, SPC c o n f l i c t i n g SPC w i t h SPC C-x b <return>
M-> <return> | a <tab> b <M-right> <M-S-right> x <S-up>
<S-right> <S-left> <S-up> C-a C-p C-p C-k C-k C-k C-k
C-k C-x C-s C-x b <return> s h i f t - s e l e c t
i o n SPC a n d SPC s o m e SPC o r g - m o d e SPC
b i n d i n g s M-q C-x C-s C-x C-s C-n C-n C-n C-n
C-n M-} C-o <return> T h e r e ' s SPC a SPC t h i
n g SPC c a l l e d SPC [A c e SPC J u m p] [2]
<return> <return> [2] : SPC C-y C-p C-p C-e SPC w
h i c h SPC l e t s SPC y o u SPC j u m p SPC t o SPC

t e x t SPC b y SPC s e a <M-backspace> t y p i n g
SPC i t , SPC e v e n SPC i f SPC i t ' s SPC i n SPC
a n o t h e r SPC w i n d o w M-' . M-q C-x C-s <backspace>
, SPC s i m M-' <M-backspace> s o r t SPC o f SPC l
i k e SPC m o v i n g SPC b y SPC i n c r e m e n t
a l SPC s e a r c h . M-q C-x C-s C-h l
i g n SPC w i t h SPC <M-backspace> w i t h SPC t h
e SPC d i s p l a y SPC d i v i d e d SPC i n t o M-q
SPC v e r t i c a l SPC " t r a c k " S-SPC e a c <M-backspace>
<backspace> <backspace> s " S-SPC e a c h SPC d i v
i d e d SPC i n t o SPC h o r i z o n t a l SPC " w
i n d o w s " M-q M-" M-" M-" M-" SPC m i <backspace>
<backspace> <backspace> C-e SPC m i g h t SPC b e SPC
a SPC C-x C-s <M-backspace> <M-backspace> p r o v i
d e SPC s i m p l e r SPC a n d SPC m o r e SPC p r
e d i c t a b l e SPC b e a v <backspace> <backspace>
h a v i o r . M-q C-x C-s <backspace> SPC t h a n SPC
t h e SPC t r a d <M-backspace> m <backspace> S m t
a l l t <M-backspace> S m a l l t a l k - s t y l e
SPC o v e r a l <backspace> <backspace> l a p p i n
g SPC w i n d o w s . C-x C-s M-> C-p C-p C-p M-b M-b
<M-backspace> <M-backspace> w o n d e r SPC i f SPC
M-q M-> C-x C-s M-v M-v M-v M-v M-v M-v M-v M-v M-v
M-v M-v <next> <next> <next> <next> <next> <next> <next>
<next> <prior> <prior> <down> C-h l C-x
o C-r C - h SPC l M-> C-SPC C-p C-p C-SPC M-< M-w C-x
o M-{ M-{ C-y C-x C-x C-> C-g C-x C-s M-{ C-x 1 M-{
M-{ M-{ M-{ M-{ M-{ <next> <next> <next> <next> <next>
<next> <next> <next> <next> <next> <next> <next> <next>
<next> <next> <next> <next> <next> <next> <next> <next>
<next> <prior> <prior> <next> <next> <next> <next>
<prior> <prior> <prior> <prior> <next> C-x b p y <return>
C-f C-SPC M-v M-v M-{ C-n C-n M-= C-g C-x 2 C-x b <return>
M-> M-v M-v C-v M-v M-v M-v M-} C-o <return> I SPC
u s e d SPC t h i s SPC t a b l e SPC t o SPC g e n
e r a t e SPC t h e SPC b o <M-backspace> <M-backspace>
t h e SPC a b o v e : M-b M-b M-b M-b o f SPC 9 9 SPC
k e y b i n d i n g s SPC C-x C-s C-x o C-SPC C-v C-v
C-v C-v C-n C-n C-n C-n C-n C-n C-n C-n C-n C-n C-n
C-n M-w C-x o C-e <return> <return> C-y C-x C-x C->
C-g C-x C-s C-x C-x C-k C-x C-x M-| s o r t <return>
C-x C-x C-u M-| M-p <return> C-h l
C-x C-> C-x C-x C-o C-r C - h SPC l C-l C-l C-l C-x
C-x C-r C-r C-p C-p C-p C-p C-p C-p C-p C-p C-g C-p
C-p C-p C-p C-p C-p C-p C-p C-p M-f M-f C-f C-f C-SPC
C-p C-p C-p C-p C-p C-p M-b M-b M-b C-w C-x C-s C-h
k M-| M-} C-x o C-x b C-s <return> C-o M - | SPC s
h e l l - c o m M-/ <backspace> <backspace> <backspace>
m M-/ - M-/ <M-backspace> o n - r e g i n <backspace>
o n C-h f <return> C-x o C-x b <return> C-SPC M-{ C-n
M-| ~ / d e v e l / d e v 3 / d e c o d e l <tab> <return>
C-x b C-s <return> C-x C-s C-x b <return> C-x C-x M-|
M-p <return> C-x o M-> C-x o C-x b <return> <return>
C - x SPC 0 SPC C-h C-g C-x 2 C-h k C-x 0 C-x o C-x
o d e l e t e - i <backspace> w i n d o w C-x C-s C-h

```
k C-x M-< C-h k C-x s C-h k <M-right> <return> < M
- R i g h t > S-SPC f o r w r <backspace> a r d - w
o r d <return> < M - S - R i g h t > C-h k <M-S-right>
SPC f o r w a r d - w o r d C-x C-s C-x b C-g C-x o
C-x o C-r C-g C-p C-p C-p C-p C-p C-SPC C-p C-SPC C-p
M-w C-x o C-x o C-x b <return> C-v C-v C-v C-v C-v
C-v C-v C-v C-v M-v C-p C-p C-y C-p C-> C-x C-s C-h
l
<down> <right> <down> <down> <down> <down> <down> <down>
<down> <down> <down> <down> <down> <down> <down> <down>
<down> <down> <down> <down> <down> <down> <down> <down>
<down> <down> <down> <down> <down> <down> <down> <down>
<down> C-p M-v C-v C-n C-n C-n C-n C-n C-n C-n C-n
C-n C-n C-n C-n C-n C-n C-n C-n C-n C-p C-SPC C-n C-n
C-n C-n C-n C-n C-n C-n C-n C-n C-n C-n C-n C-n C-n
C-n C-n C-n C-n C-n C-n C-e C-x r k C-p C-p C-p C-p
C-p C-p C-p C-p C-p C-p C-p C-p C-p C-p C-p C-p C-p
C-p C-p C-p C-p C-p C-p C-p C-p C-p C-p C-p C-p C-p
C-p C-l C-p M-{ C-n C-e SPC SPC SPC SPC SPC SPC SPC
SPC SPC SPC SPC SPC SPC SPC SPC SPC C-x r y C-x C--
M-{ C-l C-n C-n C-n C-n C-n C-n C-l C-/ C-p SPC SPC
SPC SPC SPC SPC SPC SPC SPC SPC SPC SPC SPC SPC SPC
SPC SPC SPC SPC SPC SPC SPC SPC SPC SPC SPC SPC SPC
SPC SPC SPC SPC SPC SPC SPC SPC SPC SPC SPC SPC SPC
SPC SPC SPC SPC SPC SPC SPC SPC SPC SPC SPC SPC SPC
SPC SPC SPC SPC SPC SPC SPC SPC SPC SPC SPC C-x r y
M-b M-b M-b M-b M-b M-b M-b C-s C-w C-w M-b M-b C-b
C-b C-b C-b C-b C-b C-b C-b C-b C-b C-b C-b C-b C-b
C-b C-b C-b C-b C-b C-b C-b C-b C-b C-b C-k C-p C-k
C-p C-k C-p C-k C-p C-k C-p C-k C-p C-k C-p C-k C-/
C-/ C-/ C-/ C-/ C-/ C-/ C-/ C-/ C-/ C-/ C-/ C-/ C-/
C-/ C-/ C-/ C-/ C-p C-/ C-v C-n C-n C-n C-n C-n C-n
M-} C-SPC M-} C-w C-h l
```

Some notes on most common command frequencies. Out of 9014 of the commands recorded above and successfully decoded by a janky unreliable script, these 21 account for 90% of the commands:

```
5632     5632 self-insert-command (e.g., d)
 321     5953 next-line (e.g., C-n)
 297     6250 other-window (e.g., C-x o)
 242     6492 newline (e.g., <return>)
 216     6708 delete-backward-char (e.g., <backspace>)
 210     6918 previous-line (e.g., C-p)
 162     7080 save-buffer (e.g., C-x C-s)
 112     7192 scroll-up (e.g., C-v)
 112     7304 forward-word (e.g., M-f)
 101     7405 scroll-down (e.g., M-v)
  98     7503 isearch-backward (e.g., C-r)
  90     7593 describe-key (e.g., C-h k)
  71     7664 backward-kill-word (e.g., <M-backspace>)
  68     7732 previous-history-element (e.g., M-p)
  67     7799 indent-for-tab-command (e.g., <tab>)
  62     7861 backward-word (e.g., M-b)
  59     7920 end-of-line (e.g., C-e)
```

```
56      7976 forward-paragraph (e.g., M-})
56      8032 exchange-point-and-mark (e.g., C-x C-x)
55      8087 set-mark-command (e.g., C-SPC)
54      8141 keyboard-quit (e.g., C-g)
```

Although this is the majority of my keystrokes, and having just these implemented would make something "feel like an Emacs", this obviously isn't everything essential; it's missing such basics as forward-char, find-file, yank, kill-ring-save (the new copy-region-as-kill), eval-expression, and execute-extended-command. The next 9% (90% of the remainder) are accounted for by the following 29 commands:

```
49      8190 forward-char (e.g., C-f)
48      8238 delete-char (e.g., C-d)
46      8284 yank (e.g., C-y)
45      8329 open-line (e.g., C-o)
45      8374 view-lossage (e.g., C-h l)
44      8418 backward-paragraph (e.g., M-{)
43      8461 dabbrev-expand (e.g., M-/)
41      8502 iswitchb-buffer (e.g., C-x b)
41      8543 kill-ring-save (e.g., M-w)
40      8583 fill-paragraph (e.g., M-q)
37      8620 end-of-buffer (e.g., M->)
33      8653 indent-rigidly-4 (e.g., C->)
30      8683 kill-line (e.g., C-k)
29      8712 move-beginning-of-line (e.g., C-a)
24      8736 kill-word (e.g., M-d)
21      8757 ??? (e.g., <M-S-right>)
21      8778 recenter-top-bottom (e.g., C-l)
20      8798 isearch-forward (e.g., C-s)
16      8814 backward-char (e.g., C-b)
15      8829 describe-function (e.g., C-h f)
15      8844 delete-other-windows (e.g., C-x 1)
14      8858 scroll-other-window (e.g., C-M-v)
14      8872 beginning-of-buffer (e.g., M-<)
12      8884 kmacro-end-or-call-macro (e.g., <f4>)
10      8894 execute-extended-command (e.g., M-x)
10      8904 undo (e.g., C-/)
 9      8913 smart-apostrophe (e.g., M-')
 8      8921 split-window-vertically (e.g., C-x 2)
 8      8929 delete-indentation (e.g., M-^)
```

So if I had those commands implemented, if my current editing session were typical (which it probably isn't), I would run into a missing stair about once out of every 100 keystrokes, probably a couple of times a minute; still enough to break the spell of suspension of disbelief, but close to usable. The remaining 24 commands, though, include some very significant ones:

```
 7      8936 universal-argument (e.g., C-u)
 7      8943 find-file (e.g., C-x C-f)
 6      8949 comint-interrupt-subjob (e.g., C-c C-c)
 6      8955 smartquote (e.g., M-")
```

```
6     8961 insert-parentheses (e.g., M-()
5     8966 comint-previous-prompt (e.g., C-c C-p)
5     8971 shell-command-on-region (e.g., M-|)
5     8976 next-history-element (e.g., M-n)
4     8980 kmacro-start-macro-or-insert-counter (e.g., <f3>)
3     8983 eval-expression (e.g., M-:)
3     8986 comint-send-eof (e.g., C-c C-d)
3     8989 mark-sexp (e.g., C-M-S-SPC)
3     8992 digit-argument (e.g., M-0)
3     8995 split-window-horizontally (e.g., C-x 3)
2     8997 magit-status (e.g., C-x g)
2     8999 delete-window (e.g., C-x 0)
2     9001 query-replace (e.g., M-%)
2     9003 eval-last-sexp (e.g., C-x C-e)
2     9005 kill-buffer (e.g., C-x k)
2     9007 kill-region (e.g., C-w)
2     9009 shell (e.g., M-o)
2     9011 text-scale-adjust (e.g., C-x C-=)
2     9013 count-lines-region (e.g., M-=)
1     9014 downcase-word (e.g., M-l)
```

My notes on QEmacs from Dercuano noted the things I missed from Emacs in QEmacs: M-^, M-;, C-k appending properly, M-q leaving you in place, redisplay that isn't visibly slow (!), M-/, control-backspace, command-granularity undo, and prefix arguments. Also some things I used that did work: goto-line and yank-pop, say. I was astonished that with only 88 commands it managed to be pretty usable, but it seems like if I implemented *my* most used Emacs commands instead of Bellard's, the set to get to comfort would be even smaller, less than 75.

I used this table of 99 keybindings to generate the above:

```
<M-Right> forward-word
<M-S-Right> forward-word
<M-backspace> backward-kill-word
<S-backspace> delete-backward-char
<S-down> next-line
<S-up> previous-line
<backspace> delete-backward-char
<down> next-line
<f3> kmacro-start-macro-or-insert-counter
<f4> kmacro-end-or-call-macro
<next> scroll-up
<prior> scroll-down
<return> newline
<tab> indent-for-tab-command
<up> previous-line
C-/ undo
C-0 digit-argument
C-1 digit-argument
C-2 digit-argument
C-3 digit-argument
C-4 digit-argument
C-5 digit-argument
```

C-6 digit-argument
C-7 digit-argument
C-8 digit-argument
C-9 digit-argument
C-> indent-rigidly-4
C-M-S-SPC mark-sexp
C-M-SPC mark-sexp
C-M-v scroll-other-window
C-SPC set-mark-command
C-a move-beginning-of-line
C-b backward-char
C-c C-c comint-interrupt-subjob
C-c C-d comint-send-eof
C-c C-p comint-previous-prompt
C-d delete-char
C-e end-of-line
C-f forward-char
C-g keyboard-quit
C-h f describe-function
C-h k describe-key
C-h l view-lossage
C-k kill-line
C-l recenter-top-bottom
C-n next-line
C-o open-line
C-p previous-line
C-r isearch-backward
C-s isearch-forward
C-u universal-argument
C-v scroll-up
C-w kill-region
C-x 0 delete-window
C-x 1 delete-other-windows
C-x 2 split-window-vertically
C-x 3 split-window-horizontally
C-x C-= text-scale-adjust
C-x C-e eval-last-sexp
C-x C-f find-file
C-x C-s save-buffer
C-x C-x exchange-point-and-mark
C-x b iswitchb-buffer
C-x g magit-status
C-x k kill-buffer
C-x o other-window
C-y yank
M-" smartquote
M-% query-replace
M-' smart-apostrophe
M-(insert-parentheses
M-/ dabbrev-expand
M-0 digit-argument
M-1 digit-argument
M-2 digit-argument
M-3 digit-argument
M-4 digit-argument

```
M-5 digit-argument
M-6 digit-argument
M-7 digit-argument
M-8 digit-argument
M-9 digit-argument
M-: eval-expression
M-< beginning-of-buffer
M-= count-lines-region
M-> end-of-buffer
M-^ delete-indentation
M-b backward-word
M-d kill-word
M-f forward-word
M-l downcase-word
M-n next-history-element
M-o shell
M-p previous-history-element
M-q fill-paragraph
M-v scroll-down
M-w kill-ring-save
M-x execute-extended-command
M-{ backward-paragraph
M-| shell-command-on-region
M-} forward-paragraph
S-SPC self-insert-command
SPC self-insert-command
```

Notice that 20 of these 99 are just digit-argument. I supplemented these with self-insert-command bindings for printable ASCII.

indent-rigidly-4, smartquote, and smart-apostrophe are little commands I wrote that I often find useful.

Worth noting is that GNU Emacs implements most of these commands in Elisp, with a few exceptions: self-insert-command, other-window, delete-backward-char, forward-word, scroll-up, end-of-line, scroll-down, delete-char, forward-char, backward-char, scroll-other-window (!?), delete-other-windows, execute-exptended-command, kill-buffer, and downcase-word (!?). Some of them, like recenter-top-bottom, are thin Lisp veneers on top of primitive functions such as recenter.

Window management

The Emacs way of handling windows from the keyboard is totally broken. With two windows on the screen it's fine. Three is awkward. Four is unusable. In the above lossage there are strings of up to six other-window commands in a row. Any number of better alternatives for window *switching* have been invented: Win16 LRU alt-tab; screen/tmux numbering of windows; irssi numbering of windows with alt-1/alt-2/alt-3 etc. to switch between them and alt-a to jump to the window that's demanding attention; modifier keys which make the arrow keys move you between windows (Hovav Shacham's windmove from 1998, inspired by Julian Assange's change-windows-intuitively, uses shift by default, conflicting with cua-mode shift-selection and some org-mode bindings);

iswitchb's/icomplete's/ido's LRU list of buffers with typeahead filtering; Win16 MDI ctrl-tab/ctrl-f4; etc. Emacs itself has accreted ace-window (apparently this involves assigning numbers to all the windows so you can type C-x o 3, and there are other characters for things like deleting windows, splitting windows, maximizing windows, etc.), dimitri/switch-window (same), etc.

Spacemacs uses SPC 1, SPC 2, etc., to switch to windows 1, 2, etc, and has window-manipulation commands on SPC w.

There's a thing called Ace Jump which lets you jump to text by typing it, even if it's in another window, sort of like moving by incremental search.

A different problem is that many built-in Emacs commands, being designed for two windows or less, happily replace your window contents with their own, although in some cases you can appease them by offering them a sacrificial window. There's a whole ecosystem around the `display-buffer` command that attempts to make this less frequently annoying.

I think this is a symptom of a broken user interface design. I'm not sure what the right design is; maybe something where opening new things opens new windows by default, which you can then maximize and unmaximize, or close, or resize. I wonder if the Cedar/Oberon design with the display divided into vertical "tracks" each divided into horizontal "windows" might provide simpler and more predictable behavior than the Smalltalk-style overlapping windows.

Topics

- Programming (p. 1141) (49 notes)
- Human-computer interaction (p. 1156) (22 notes)
- Experiment report (p. 1162) (14 notes)
- Safe programming languages (p. 1172) (11 notes)
- Lisp (p. 1174) (11 notes)
- Transactions (p. 1239) (4 notes)
- Editors (p. 1257) (4 notes)
- Emacs (p. 1298) (3 notes)
- Incremental search (p. 1362) (2 notes)
- Qemacs

Trie PEGs

Kragen Javier Sitaker, 02021-01-15 (4 minutes)

Reading some hackers talking about Packrat parsers, it occurred to me that perhaps you could get a huge speedup in Packrat by a different memoizing approach.

Normally in Packrat a parsing grammar node (or perhaps a nonterminal) is invoked at a given position of a given stream and returns a result, either a parse tree and a new position, or a failure, and in either case the result is memoized with the (position, grammar node) pair as the memo table key, or (if the memo table is global or associated with the grammar rather than the input stream) the (input stream, position, grammar node) triple. The memo table ensures that we only ever invoke each parse at most once for each input position.

To facilitate incremental parsing, the parsers in Darius Bacon's parsing sketch additionally return a "far" value: how far ahead they read in the input stream. This makes it possible to invalidate memo table entries that depend on changed parts of the text.

It occurred to me, though, that a much more powerful approach is possible: instead of using the (position, grammar node) pair as the key, we can use the *actual text examined* by the parsing process. Then, if we try to parse the same text again later in the input stream with the same production, we can reuse the same memo-table result. So, for example, when parsing a Python program, every time you find an argument list in a function declaration that looks like `(self):`, you can simply return the same argument list after a memo-table lookup. Every identifier in a program would only be parsed for real only once for each character following it; all other attempts to parse them would find hits in the memo table. So, for example, all the occurrences of `weight(`, would have a single memo-table entry, but `weight` and `weight\n` would get their own entries. Every occurrence of `stack.pop()\n` or `for word in words:\n` or `if tree is None:\n` or `i+1]` would share the same memo-table entry.

This might sound like an unreasonable thing to do that would result in a huge and slow memo table, and it might be, but also it might not be. The memo table itself can be stored as a trie rather than a hash table, for example with Patricia, enabling it to be traversed relatively quickly and use a relatively manageable amount of space. And the great advantage it would have over the usual memo-table approach is that the majority of memo-table entries would be shared among several different locations in the source text.

It might also be possible to use this approach to automatically get incremental reparsing by using the same memo table for more than one parse. To the extent that the second parse shares text with the first parse, even in a different order, the old AST nodes will just flop ready-made out of the memo table.

The big drawback of this approach is that it loses Packrat's linear-time guarantee, because now the process of testing for a memo table hit potentially involves examining all the subsequent characters in the input stream. That means the overall parse potentially takes

quadratic time rather than linear time.

Topics

- Programming (p. 1141) (49 notes)
- Performance (p. 1155) (22 notes)
- Algorithms (p. 1163) (14 notes)
- Compilers (p. 1178) (10 notes)
- Parsing (p. 1228) (5 notes)
- Caching (p. 1266) (4 notes)
- Parsing expression grammars (PEGs) (p. 1343) (2 notes)
- Tries
- Packrat

Chat over a content-centric network

Kragen Javier Sitaker, 02021-01-15 (updated 02021-01-16) (3 minutes)

What would the simplest usable chat program look like? In 1999 IN wrote a chat server in C in a .signature:

```
char a[99]="  KJ",d[999][16];main(){int s=socket(2,1,0),n=0,z,l,i;*(short*)a=2;
if(!bind(s,a,16))for(;;){z=16;if((l=recvfrom(s,a,99,0,d[n],&z))>0){for(i=0;i<n;
i++){z=(memcmp(d[i],d[n],8))?z:0;while(sendto(s,a,l,0,d[i],16)<0);}z?n++:0;}}}
```

The client is four lines of C:

```
char a[99]="  KJ";main(int c,char**v){int s=socket(2,1,0);char*p,*t=strchr(*++v
,'@'),*o=a+4;*(short*)a=2;p=t;while(*p)(*p++&48)-48?*o++=atoi(p):0;connect(s,a,
16);strncpy(a,v[1],7);a[7]=':';a[8]=32;if(fork())while((c=read(0,a+9,90))>0)(
write(s,a,c+9)>0)||exit(0);else while((c=read(s,a,99))>0)write(1,a,c);}
```

This was for Solaris, where `SOCK_DGRAM` was 1, rather than 2 as in Linux `<bits/socket.h>`. Both the client and the server are vulnerable to buffer overflows, the server manifests a single chat channel, there are no private messages, there's no recovery from packet loss, and the server just keeps sending to disconnected clients forever. Nevertheless, under favorable circumstances, these two programs do manifest a usable text chat system over TCP/IP.

If you have a shared Unix filesystem you can use a three-line shell script for a client and need no server:

```
#!/bin/sh
: ${1?"usage: $0 nick [chan]"} ${chan=${2-/tmp/chat}}
sleep 1; tail -f "$chan" & pid=$?; trap "kill $pid" 0
while read t; do echo "<$1> $t"; done >> "$chan"
```

This inherits the Unix filesystem's permissions, the Unix terminal's line editing, and whatever networking your filesystem supports, and it should be reliable up to messages of `PIPE_BUF` size.

In 2008 I wrote an IRC client in 40 lines of shell script:

```
#!/bin/sh
# In the grim future of the Debian netinst disk, there is only nc.
# And dd and sh, of course.
: ${2?"Usage: $0 ircserver nickname"}
ircserver="$1"
nickname="$2"
grimdir="`dirname "$0"`"
case "$grimdir" in /*) ;; *) grimdir="../$grimdir" ;; esac

tmpdir=".tmp.grimirc.$$"
mkdir "$tmpdir"
cd "$tmpdir"
trap 'cd ..; rm -rf "$tmpdir"' 0
```

```
(echo user grimirc hostname "$ircserver" :grimirc user
echo nick "$nickname"
> grimirc-responses
tail -f grimirc-responses &
while read command
        do case "$command" in
        /join*) echo "joining">/dev/tty
                set $command; currentchan="$2"
                echo "$command" > .grimtmp
                dd bs=1 skip=1 < .grimtmp 2>/dev/null;;
        /*) echo "$command" > .grimtmp
                 dd bs=1 skip=1 < .grimtmp 2>/dev/null;;
        *) echo "PRIVMSG $currentchan :$command"
        esac
done) | "$grimdir/grimdebuglog" | nc "$ircserver" 6667 | while read response
        do echo "$response"
        case "$response" in
        "PING "*) echo "responding to ping"
                set $response
                shift
                echo "PONG $*" >> grimirc-responses
        esac
done
```

So, suppose we want to build a small program that implements something like Van Jacobson's CCN, with "interest" packets that get replied to with matching "data" packets or forwarded to where you think they might find those packets. That would make it pretty easy to implement a chat system, wouldn't it? One that really worked? How hard would that be to implement?

Topics

- Programming (p. 1141) (49 notes)
- Small is beautiful (p. 1190) (8 notes)
- C (p. 1194) (8 notes)
- Ccn (p. 1380) (2 notes)

- Irc
- Chat

Some notes on compiling and notations for grammars, starting from the inspiring RPN example in Parson

Kragen Javier Sitaker, 02021-01-15 (updated 02021-12-31)
(15 minutes)

Darius Bacon's Parson parsing library includes a beautiful little example compiler in 11 lines of code, "eg_calc_to_rpn.py"; quoted in full:

```
from parson import Grammar, alter

g = Grammar(r"""  stmt* :end.

stmt   :  ident '=' exp0 ';'  :assign.

exp0   :  exp1 ('+' exp1      :'add')*.
exp1   :  exp2 ('*' exp2      :'mul')*.

exp2   :  '(' exp0 ')'
       |  /(\d+)/
       |  ident               :'fetch'.

ident  :  /([A-Za-z]+)/.

FNORD ~:  /\s*/.
""")(assign=alter(lambda name, *rpn: rpn + (name, 'store')))

## print ' '.join(g('v = 42 * (5+3) + 2*2; v = v + 1;'))
#. 42 5 3 add mul 2 2 mul add v store v fetch 1 add v store
```

As can be seen from the Halp test at the bottom, this is all that's needed for Parson to compile an infix grammar to a sequence of stack-machine operations, although it will go wrong if you use "add", "mul", "fetch", or "store" as variable names.

Review of standard properties of Kleene algebras

A "Kleene algebra" is an idempotent semiring augmented with an unary "closure" operator "★" with certain properties. In the concrete case of regular languages, the + and · operators of the semiring are alternation and concatenation; that is, if a and b are two languages, $a + b$ is their union (and equal to $b + a$, because set union is commutative), and $a{\cdot}b$ is sort of their Cartesian product: the set of strings that can be produced by concatenating a string $\alpha \in a$ with a string $\beta \in b$ to form $a \mid\mid b$. (This is only *sort of* their Cartesian product because the string boundary is lost, and as a result this

product operator is associative: $(a \cdot b) \cdot c = a \cdot (b \cdot c)$, which is not true with most formulations of the Cartesian product.) In general $b \cdot a$ is a different language. We can observe that there is a language 0 containing no sentences that is the identity for +, and the language 1 containing only the empty sentence that is the (left and right) identity for ·. Multiplication by 0 produces 0: $0 \cdot a$ and $a \cdot 0$ contains no sentences for any a because there are no sentences to draw from 0 for the left substring (respectively the right substring) of the product.

We can also observe that multiplication distributes over addition: $a \cdot b + a \cdot c = a \cdot (b + c)$, because iff you have a string $\alpha \in a$ and a string $\gamma \in b + c$, then by definition $(\alpha \mid\mid \gamma) \in a \cdot (b + c)$. Let δ be $\alpha \mid\mid \gamma$. Also by definition, $\gamma \in b \vee \gamma \in c$. In the first case $\delta \in a \cdot b$ by the definition of ·, and therefore $\delta \in a \cdot b + a \cdot c$. In the second case $\delta \in a \cdot c$ and so again $\delta \in a \cdot b + a \cdot c$. The same chain of reasoning works in reverse and *mutatis mutandis* to show $b \cdot a + c \cdot a = (b + c) \cdot a$.

These are the semiring properties (+ a commutative monoid operation, · a monoid operation, distributivity, and annihilation) so we can conclude that languages are a semiring under such concatenation and alternation. Importantly, this conclusion is not limited to, for example, regular languages.

It happens that, with the union interpretation of addition, $a + a = a$, so languages form a so-called "idempotent semiring", which induces a partial ordering relation: $a \leq b$ iff $a + b = b$. In this case, this is simply the subset relation $\subseteq$. (I'm not sure what happens in non-idempotent semirings.)

(Henceforth I will write multiplication simply as juxtaposition.)

The usual Kleene closure operation ★ from regular expressions can be defined in terms of an equation:

```
x★ = 1 + x x★ (or equivalently 1 + x★ x)
```

But the definition used for Kleene algebras, invented not by Kleene but by Kozen in 1994, is axiomatic rather than equational:

- $1 + aa^\star \leq a^\star$
- $1 + a^\star a \leq a^\star$
- $ax \leq x \Rightarrow a^\star x \leq x$
- $xa \leq x \Rightarrow xa^\star \leq x$

A BNF grammar as a set of equations in a Kleene algebra

Let's consider the expression grammar above as a grammar over tokens, with the semantic actions and tokens stripped out:

```
g       : stmt*.

stmt    :  ident '=' exp0 ';'.

exp0    :  exp1 ('+' exp1)*.
exp1    :  exp2 ('*' exp2)*.
```

```
exp2   :  '(' exp0 ')'
       |  number
       |  ident.
```

Parson parses PEGs, but we can safely regard this as a CFG because it doesn't use any of the extra-CFG features of PEGs; for example, it doesn't use negation, and it has no ambiguity of the sort that PEGs resolve in a non-CFG way. So we can rewrite this as a system of equations in a Kleene algebra:

$$g = s\bigstar$$
$$s = i \text{ '=' } e_0 \text{ ';'}$$
$$e_0 = e_1 \text{ ('+' } e_1)\bigstar$$
$$e_1 = e_2 \text{ ('' } e_2)\bigstar$$
$$e_2 = \text{'(' } e_0 \text{ ')'} + n + i*$$

We could think of the problem of constructing a parser as the problem of finding the solution, or at least a least fixpoint, of these equations. What (infinite, non-regular) language g satisfies this system of equations? And a mostly satisfactory answer is "The language recognized by such-and-such a pushdown automaton," in general a nondeterministic one. We can always compute such an answer, and it's reasonable to think of the parser generation problem as the problem of solving such sets of equations to compute such an answer.

It's not an entirely satisfactory answer, though, because if the PDA in question isn't deterministic, its equivalence to another PDA is undecidable. (The 1997 proof that the problem *is* decidable in the deterministic case won Sénizergues the 2002 Gödel Prize, suggesting that it may be difficult in practice.)

We don't need the whole Kleene algebra to express the system of equations, though; its semiring operations are enough, and we don't even need parentheses in the notation, except as literal strings:

$$g = s + g\, s$$
$$s = i \text{ '=' } e_0 \text{ ';'}$$
$$e_0 = e_1 + e_1 \text{ '+' } e_0$$
$$e_1 = e_2 + e_2 \text{ '' } e_2$$
$$e_2 = \text{'(' } e_0 \text{ ')'} + n + i*$$

Note that there are three "nonlinear" terms in here, where two nonterminals (that is, from the equation-solving point of view, unknowns) are part of a single product. A CFG that cannot do this, where each concatenation is restricted to at most one nonterminal, cannot express arbitrarily branching derivations — each parse tree node can only have a single non-leaf child — but it can still describe some non-regular languages, for example:

$$p = \text{'<' } p \text{ '>'} + \text{'⚜'}$$

But it can't match parentheses in more than one place, so it can't express, for example:

$$q = \text{'\{' } q \text{ '\}'} + q\, q + \text{'☮'}$$

So these "linear" context-free grammars are a class of languages strictly larger than regular languages, but strictly smaller than CFGs. There's probably a well-known name for them.

Notational alternatives

Consider the original grammar again:

```
stmt* :end.

stmt   :  ident '=' exp0 ';'  :assign.

exp0   :  exp1 ('+' exp1      :'add')*.
exp1   :  exp2 ('*' exp2      :'mul')*.

exp2   :  '(' exp0 ')'
       |  /(\d+)/
       |  ident               :'fetch'.

ident  :  /([A-Za-z]+)/.

FNORD ~:  /\s*/.
```

Or the semantics-free version:

```
stmt*.

stmt   :  ident '=' exp0 ';'.

exp0   :  exp1 ('+' exp1)*.
exp1   :  exp2 ('*' exp2)*.

exp2   :  '(' exp0 ')'
       |  /(\d+)/
       |  ident.

ident  :  /([A-Za-z]+)/.

FNORD ~:  /\s*/.
```

If we were driven by a mad Tuftean urge to minimize ink on the page, we could start by shortening the identifiers as I did before:

```
s*.

s   :  i '=' e ';'.

e   :  f ('+' f)*.
f   :  g ('*' g)*.

g   :  '(' e ')'
    |  /(\d+)/
    |  i.

i  :  /([A-Za-z]+)/.

FNORD ~:  /\s*/.
```

Then we could omit the (strictly speaking, redundant) colons, and replace the | with something lighter, such as `,`, even though `;` would have a more accurate connotation from Prolog:

```
s*.

s  i '=' e ';'.

e  f ('+' f)*.
f  g ('*' g)*.

g  '(' e ')',
   /(\d+)/,
   i.

i  /([A-Za-z]+)/.

FNORD ~  /\s*/.
```

If we switch to `;` for rule terminators, we could use `.` to tag nonterminals, thus eliminating the need for *most* quotes:

```
.s*;
s  .i = .e ';';
e  .f (+ .f)*;
f  .g ('*' .g)*;
g  '(' .e ')', /(\d+)/, .i;
i  /([A-Za-z]+)/;
FNORD ~  /\s*/;
```

Hmm, that didn't pay off quite as well as I was hoping; I could instead tag the literal tokens with a lighter-weight `:`, perhaps, unless they contain whitespace:

```
s*.
s  i := e :; .
e  f (:+ f)*.
f  g (:* g)*.
g  :( e :) , /(\d+)/, i.
i  /([A-Za-z]+)/.
FNORD ~  /\s*/.
```

We can replace Kleene's `*` with a "join" operator, for which we can use `;`, vaguely connected to Perl's `$;`; `( number; :, )` would mean "one or more numbers separated by commas", for example, equivalent to the current `(number (:, number)*)` construct. Parson spells this `number ** ','` or, in the Python interface, `star(number, separator=',')`. The idea of `;` is that it's lower-noise than `**` and it binds *less* tightly than concatenation, or `,` alternation. This gives us:

```
, s ; .
```

```
s  i := e :; .
e  f ; :+ .
f  g ; :* .
g  :( e :) , /(\d+)/, i.
i  /([A-Za-z]+)/.
FNORD ~  /\s*/.
```

This eliminates the multiple references that gave rise to the necessity for naming f and g, so we can reduce this to:

```
, s ; .
s  i := e :; .
e  ((:( e :) , /(\d+)/, i) ; :*) ; :+ .
i  /([A-Za-z]+)/.
FNORD ~  /\s*/.
```

This is kind of unreadable but it’s also 97 characters. To crunch it *further* we can try defining nonterminals inline; instead of saying `i := e :;  .` and then only later defining `i`, we can define it right then and there by saying `<i /([A-Za-z]+)/> := e :;  .` If we want to refer to `i` again, and we do, it's probably more readable to refer to it as `<i>`, although that obviously does cost some strokes if we’re playing golf. This also means that the whole grammar can be inlined into one giant unreadable expression, so we no longer need any terminating periods. Meanwhile we can rename FNORD to _, as in a variable you’re ignoring in Prolog, ML, Python, or Erlang. Then we have *this*:

```
, <_ /\s*/> <i /([A-Za-z]+)/> := <e ((:( <e> :) , /(\d+)/, <i>); :* ); :+ > :; ;
```

That's 80 characters: an infix precedence partner for possibly-empty sequences of assignment statements in one line of code.

I don’t like the colon-delimited tokens, though. It’s too easy to remove the whitespace following them. So I'm going to add some noise back in:

```
,<_ /\s*/> <i /([A-Za-z]+)/> "=" <e (( "(" <e> ")", /(\d+)/, <i>); "*"); "+"> ";"
;
```

It’s 83 characters now, but I think enormously more readable.

So those are our means of *composition* or combination of grammars to make richer grammars. But it would also be useful to have a lot of canned *primitives* with common meanings, not just regular expressions and constant strings, for common kinds of tokens with common semantics. For example:

- $a one or more upper and lowercase letters
- $A one or more uppercase letters

• $w one or more digits, upper and lowercase ASCII letters, and underscores, starting with a non-digit
• $s zero or more spaces, tabs, carriage returns, or newlines (but not vertical tabs!)
• $S zero or more Unicode whitespace characters, including $s
• $u a single Unicode codepoint encoded in UTF-8
• $U one or more non-ASCII characters
• $d possibly signed decimal integer
• $n decimal integer with no sign
• $r either carriage return, linefeed, or carriage return and then linefeed
• $R the rest of the line: `/.*/ $r`
• $x hexadecimal digit
• $f possibly signed decimal floating-point number
• $c C-style double-quoted string with -escaping of doublequotes and \
• $e Elisp-style double-quoted string with -escaping and possible embedded newlines
• $q SQL-style apostrophe-quoted string with doubling of embedded apostrophes
• $# comment to end of line introduced with #
• $^ ASCII control characters in general, including carriage returns, linefees, tab, and delete, but not including the ISO-8859-1 control characters after delete, the other Unicode control characters like ZWNJ, or space (ASCII 32)

Such an arsenal of preloaded ammunition allows us to reduce the above expression grammar to 62 characters, about 25%:

```
,<_ $s> $a "=" <e (( "(" <e> ")", $n, $a); "*"); "+"> ";";
```

We could even make some reasonable extensions:

```
,<_ $s, $#> $w "=" <e (( "(" <e> ")", $n, $w); "*", "/"); "+", "-"> ";";
```

An accommodation especially for precedence parsers would be to declare that the ; operator associates to the left, allowing us to flatten the grammar considerably at, perhaps, some cost to readability:

```
,<_ $s, $#> $w "=" <e "(" <e> ")", $n, $w; "*", "/"; "+", "-"> ";";
```

As written, this can accidentally match the empty string an infinite number of times, which may or may not be a problem depending on your matching technology. If so, parens help:

```
,(<_ $s, $#> $w "=" <e "(" <e> ")", $n, $w; "*", "/"; "+", "-"> ";";)
```

Because `;x` matches zero or more `x`es, we can rewrite this without any parens at all:

```
; <_ $s, $#> $w "=" <e "(" <e> ")", $n, $w; "*", "/"; "+", "-"> ";"
```

To compile into Forth, we could maybe try to include snippets of literal output with {} and use {{}} to mark bits of the input to copy to the output:

```
; <_ $s, $#> {{$w}} "="
    <e "(" <e> ")", {{$n}}, {{$w}};
         "*" {"*"}, "/" {"/"}; "+" {"+"}, "-" {"-"}>
{"swap !"} ";"
```

The inability to reorder chunks of the input here is a clear weakness. In this case we not only need a `swap`, we're emitting the operators too early! We can fix this in the conventional way, though losing the advantage of ;:

```
; <_ $s, $#> {{$w}} "="
    <e <e2 <e3 "(" <e> ")", {{$n}}, {{$w}}>
            (; "*" <e3> {"*"}, "/" <e3> {"/"})>
        (; "+" <e2> {"+"}, "-" <e2> {"-"})>
{"swap !"} ";"
```

If we define a lowest-precedence `@@` operator which discards its right argument, we could write one of the above versions perhaps more readably as

```
,($w "=" <e> ";";)
@@
    <_ $s, $#>
    <e "(" <e> ")", $n, $w; "*", "/"; "+", "-">
```

Using apostrophes instead of doublequotes:

```
,($w '=' <e> ';';)
@@
    <_ $s, $#>
    <e '(' <e> ')', $n, $w; '*', '/'; '+', '-'>
```

JSON might be (from memory) something like

```
<v "[" <_ $s> (,(v; ",")) "]"
 , "{" (,(<s $c> ":" <v>; ",")) "}"
 , ("true", "false", "null") !!$s
 , $f
 >
```

Hmm, I just checked, and JSON has a slightly more complex string syntax, and doesn't actually require whitespace after the keywords as I claimed above. And my definition above of $s happens

to be perfect for JSON.

```
<v '[' <_ $s> (,(<v>; ',')) ']'
 , 'true', 'false', 'null'
 , <s '"'
    (;
        !'"' !'\\' !$^ $u,
        '\\' ('"', '\\', '/', 'b', 'f', 'n', 'r', 't', 'u' $x $x $x $x)
    ) '"'>
 , '{' (,(<s> ':' <v>; ',')) '}'
 , $f
 >
```

You might want to add more tags and regex captures to help with semantic actions:

```
,(<_ $s, $#> <s $w> "=" <e <t "(" <e> ")", <k $n>, <f $w>;
                          <m /([*/])/>; <a /([-+])/>>";";)
```

An alternative to `;` might be `|`, as in Haskell list comprehensions, though that looks awkward without extra whitespace:

```
,(<_ $s, $#> $w "=" <e "(" <e> ")", $n, $w | "*", "/" | "+", "-"> ";" | )
```

Or `:`, since we aren't using that for anything else and it's less jarring than `|` in the absence of whitespace:

```
,(<_ $s, $#> $w "=" <e "(" <e> ")", $n, $w: "*", "/": "+", "-"> ";":)
```

Or `*` of course, although that visually binds more tightly than ",":

```
,(<_ $s, $#> $w "=" <e "(" <e> ")", $n, $w * "*", "/" * "+", "-"> ";" *)
```

An alternative to would be `x: y`. This would reduce the `;` version to

```
,(<_ $s, $#> $w "=" e: "(" e ")", $n, $w; "*", "/"; "+", "-". ";";)
```

An alternative use of `.` would be for canned primitives:

```
,(<_ .s, .#> .w "=" e: "(" e ")", .n, .w; "*", "/"; "+", "-". ";";)
```

Topics

- Math (p. 1173) (11 notes)

- Compilers (p. 1178) (10 notes)
- Syntax (p. 1221) (5 notes)
- Parsing (p. 1228) (5 notes)
- Reverse Polish notation (RPN) (p. 1243) (4 notes)
- Domain-specific languages (DSLs) (p. 1260) (4 notes)
- Kleene algebras (p. 1287) (3 notes)

Can transactions solve the N+1 performance problem on web pages?

Kragen Javier Sitaker, 02021-01-16 (8 minutes)

Reading a note on Rust for web APIs I ran across the "n+1" problem, familiar to anyone who's done database-backed web sites:

> The n+1 problem is something that everyone building web applications should understand. The gist is: you have a page of photos (1 query). You want to show the author of each photo. How many queries do you end up with: 1, combining the photos & authors, or a query per photo to get the author after retrieving the photos? Or 2 queries, with the second having something like `user.id IN ids` to fetch all authors in a single pass and then reconnect them to their photos.

Now, the most straightforward kind of ORM to write will kind of push you to having the n+1 problem, but there are a variety of design approaches to writing an ORM (or other database layer) that make it possible to avoid it. In Django's ORM, for example, you'd say `photo.author_id.name` to get the field `name` from the related row in the authors table, but whether or not this results in an n+1 problem depends on how `photo` was fetched. If you did something like `for photo in Photo.objects.all()` then by default you have an n+1 problem, but you can specify `for photo in Photo.objects.all().select_related('author')` to do just a single query joining both tables. And Django has a janky special-purpose `author__publisher__country` syntax that allows following multiple levels here, and a `prefetch_related` that handles one-to-many relationships in one query per table. Other systems such as Rails have analogous facilities.

The trouble with this is that to some extent it violates DRY. You have to specify *twice* that you are going to access the photos' authors: once when you construct the query, and again later when you use it. If you don't so specify, your code will still run and produce the same results, but it may run two orders of magnitude slower (or more, if your database is fucked up enough!), which may be better or worse than just crashing, depending on your situation. When you modify the set of attributes used from each photo, you must also remember to modify the query accordingly, which is likely in a different file from the HTML template where the attributes are used.

One approach is to build up the `prefetch_related` set lazily: when an `author` object is fetched via one of the `photo` objects, we can notify the original query that the `photo` came from that it was inadequate, and it needs to add a `.prefetch_related('author')` and execute it forthwith.

Also, though, I've been enthusiastic about transactions. How can transactions help?

Maybe we can rerun the query

Closely related to the "build up `prefetch_related` lazily" approach is "abort the transaction whenever it tries to read data that isn't yet loaded, and retry it when the data arrives".

Maybe we can use type inference

Here's a wild non-transaction approach.

If we statically infer the set of attributes required on an argument passed in to the template, we can use that to build up a query (or finite number of queries) whose results have the right type. The "type" in this case is something like a solution to a set of equations describing a sort of graph:

```
photo = {url: str, author: α, ..}
α = {name: str, publisher: β, ..}
β = {country: str, ..}
```

Here `str` is a concrete type (an atomic constant of the universe of types) and the `{..}` syntax specifies a minimal set of fields (outgoing graph edge labels) that must be present at the specified node. In this case the equations are acyclic and so could be directly solved by substitution:

```
photo = {url: str, author: {name: str, publisher: {country: str, ..}, ..}, ..}
```

If we additionally have a `.` operator on field labels (analogous to Django's `__` mentioned above) perhaps we can use a distributive law to expand this:

```
photo = {url: str, author.name: str, author.publisher.country: str, ..}
```

This sort of distributivity suggests that perhaps we are looking at a semiring. You could however reasonably argue that the two statements above are not equivalent: in the second case, the photo might have two or more authors, one which has a string name and another of which has a publisher with a string country. But for the moment I will explore this possibly unsound path of reasoning to see if there's some way to chop off its feet to fit it into the semiring bed.

If we make the additional simplification of replacing "the concrete type str" (a node in the universe of types) with "a node with the field str" then really what we have here is

```
photo = url str + author α
α = name str + publisher β
β = country str

photo = url str + author (name str + publisher (country str))

photo = url str + author name str + author publisher country str
```

I'm not quite sure how to interpret the semantics here: are these really structural types, in the sense of the OCaml lower-type-bound syntax I'm aping? (Presumably `photo` here is a different sort of name than `url`, `author`, and `str`.) Are they relations, with multiplication/concatenation being interpreted as composition and addition being interpreted as relational product (and, if so, does it make sense to bring in Kleene closure, and do we get a Kleene algebra, and what about the inverse relation)? (Binary relations are

matrices indexed by the relation's domain and range over the Boolean semiring, and their matrix multiplication does correspond to composition, but their matrix addition is of course just simple union, not relational product; relational product is more closely allied to intersection.) Does this expression represent a query that could be evaluated, perhaps over a directed graph or a SQL database, and return a table with three columns? Is this just Binate again with different syntax?

Nevertheless, in some cases the solution is not quite so simple:

```
x = Guy.objects.get(id=id)
while x.manager_id is not None:
    x = x.manager_id
return x.name
```

Type inference here gives us something like:

```
guy = {manager: guy, name: α ..}
```

Which is to say:

```
guy = manager guy + name α
```

Does allowing the nose of Kleene closure into the tent allow us to solve this equation, with something like the following?

```
guy = manager* (name α)
```

See also Some notes on compiling and notations for grammars, starting from the inspiring RPN example in Parson (p. 57).

Maybe we can have stuff in RAM already

One of the claims I was making is that pervasive transactionality ought to make cache invalidation more straightforward and performant. So maybe if our database is more like an OODBMS like ObjectStore or Gemstone, the n+1 query problem stops being a problem at all: each traversal of an `author` link from a `photo` is just a read of a transactional variable, so it's not an outrageous cost and doesn't necessarily involve a context switch or network round trip.

Maybe we can pipeline

If all of the *n* individual fetches of the author of a photo can run concurrently, with their small queries being pipelined to the database server and back, then you still have n+1 queries, but only two round trips, and that might be acceptable.

Topics

- Programming (p. 1141) (49 notes)
- Performance (p. 1155) (22 notes)
- Transactions (p. 1239) (4 notes)

Notes on simulating a ZVS converter (Baxandall converter)

Kragen Javier Sitaker, 02021-01-16 (6 minutes)

I tried simulating a Baxandall converter ZVS driver circuit with circuit.js; here's the circuit:

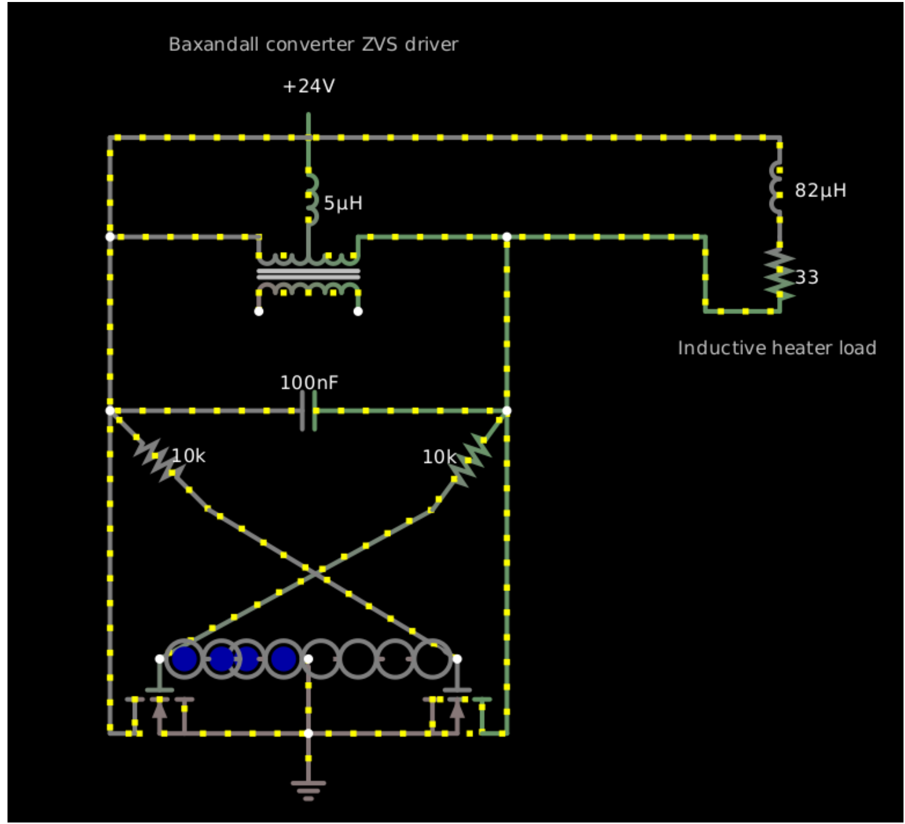
```
$ 1 1e-7 2.9224283781234943 50 100 43
l 256 64 256 112 0 0.0005 0.025188214463181376

169 224 160 224 112 0 0.000009999999999999999 1 8.881784197001252e-16 6.3171235710343775 6.342311785806956 0.99
c 128 224 384 224 0 1e-7 -40.777875024550454 0.001
w 128 224 128 112 0
w 128 112 224 112 0
w 288 112 384 112 0
w 384 224 384 432 0
R 256 64 256 16 0 0 40 24 0 0 0.5
f 352 384 352 432 32 1.5 0.02
f 160 384 160 432 40 1.5 0.02
w 176 432 256 432 0
w 256 432 336 432 0
g 256 432 256 464 0 0
w 368 432 384 432 0
w 384 224 384 112 0
w 144 432 128 432 0
w 128 432 128 224 0
x 166 -8 353 -5 4 12 Baxandall\sconverter\sZVS\sdriver
w 256 384 256 432 0
w 160 384 336 288 0
r 336 288 384 224 0 10000
r 128 224 192 288 0 10000
w 192 288 352 384 0
34 fwdrop\q3.2 1 9.32e-11 0.042 5.356678529866179 0 1
162 304 384 272 384 2 fwdrop\q3.2 0 0 1 0.1
34 fwdrop\q3.2-2 1 9.32e-11 0.042 5.356678529866179 0 1
162 208 384 224 384 2 fwdrop\q3.2-2 0 0 1 0.1
162 192 384 208 384 2 fwdrop\q3.2-2 0 0 1 0.1
162 320 384 304 384 2 fwdrop\q3.2-2 0 0 1 0.1
162 352 384 320 384 2 fwdrop\q3.2-2 0 0 1 0.1
162 272 384 256 384 2 fwdrop\q3.2 0 0 1 0.1
162 224 384 256 384 2 fwdrop\q3.2-2 0 0 1 0.1
162 160 384 192 384 2 fwdrop\q3.2-2 0 0 1 0.1
o 2 16 0 5123 320 51.2 0 2 2 3
```

By itself this produces a somewhat disappointing peak voltage multiplication of about 3×, or 6× peak to peak, with a resonant peak about 158 kHz on the FFT, which is high enough for inductive heating of even non-ferrous metals in some cases. The two strings of four 3.2-volt blue LEDs limit the gate voltages on the two N-channel MOSFETs to about 9.9 volts (2.4 volts per LED, which is a bit

unrealistic...) so they don't burn out, while the 10k resistors limit the current through the LEDs to about 5 mA.

The resonant frequency is of course $1/(2\pi\sqrt{(LC)})$, and in this case the relevant L is the 10 μH of the inductor, and the C is the 0.1 μF of the capacitor, 159.1 kHz, which is right. The peak current in the tank circuit is about 7.5 amps, 1500 times the current through the LEDs.

If I load the output (parallel to the capacitor) with a resistive load, then somewhere below about 180 Ω, the output collapses. For it to recover, the load evidently needs to be 330 Ω or more. It's possible to use an inductive load of 82 μH with 33 Ω equivalent series resistance and have it work:




```
$ 1 1e-7 2.3728258192205156 50 100 43
l 256 64 256 112 0 0.000004999999999999996 1.3758031682752978

169 224 160 224 112 0 0.000009999999999999999 1 0 -4.784632558596629 -3.4088293903213227 0.99
c 128 224 384 224 0 1e-7 33.040933721242446 0.001
w 128 224 128 112 0
w 128 112 224 112 0
w 288 112 384 112 0
w 384 224 384 432 0
R 256 64 256 16 0 0 40 24 0 0 0.5
f 352 384 352 432 32 1.5 0.02
f 160 384 160 432 40 1.5 0.02
w 176 432 256 432 0
w 256 432 336 432 0
g 256 432 256 464 0 0
w 368 432 384 432 0
w 384 224 384 112 0
w 144 432 128 432 0
```

```
w 128 432 128 224 0
x 166 -8 353 -5 4 12 Baxandall\sconverter\sZVS\sdriver
w 256 384 256 432 0
w 160 384 336 288 0
r 336 288 384 224 0 10000
r 128 224 192 288 0 10000
w 192 288 352 384 0
34 fwdrop\q3.2-2 1 9.32e-11 0.042 5.9498493010683156 0 0.1
162 208 384 224 384 2 fwdrop\q3.2-2 0 0 1 0.01
162 192 384 208 384 2 fwdrop\q3.2-2 0 0 1 0.01
162 320 384 304 384 2 fwdrop\q3.2-2 0 0 1 0.01
162 352 384 320 384 2 fwdrop\q3.2-2 0 0 1 0.01
162 224 384 256 384 2 fwdrop\q3.2-2 0 0 1 0.01
162 160 384 192 384 2 fwdrop\q3.2-2 0 0 1 0.01
162 304 384 272 384 2 fwdrop\q3.2-2 0 0 1 0.01
162 272 384 256 384 2 fwdrop\q3.2-2 0 0 1 0.01
w 128 112 128 48 0
w 128 48 560 48 0
w 384 112 512 112 0
r 560 112 560 160 0 33
w 512 112 512 160 0
w 512 160 560 160 0
l 560 112 560 48 0 0.000082 0.3059284467964981
x 494 188 623 191 4 12 Inductive\sheater\sload
o 2 16 0 5123 80 6.4 0 2 2 3
```

The original Baxandall converer circuit is by Peter Baxandall from 1959.

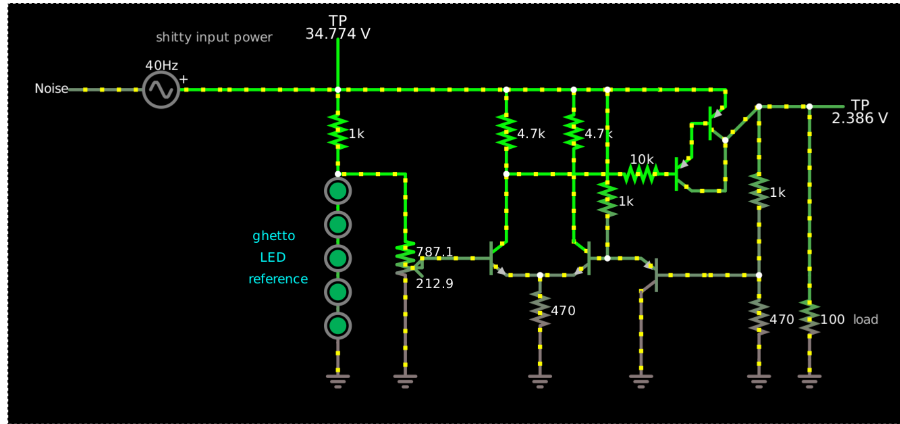
Topics

- Electronics (p. 1145) (39 notes)
- Ghettobotics (p. 1169) (12 notes)
- Power supplies (p. 1176) (10 notes)
- Falstad's circuit simulator (p. 1198) (7 notes)

A ghetto linear voltage regulator from discrete components

Kragen Javier Sitaker, 02021-01-21 (updated 02021-01-27)
(10 minutes)

I designed this ghetto linear regulator:



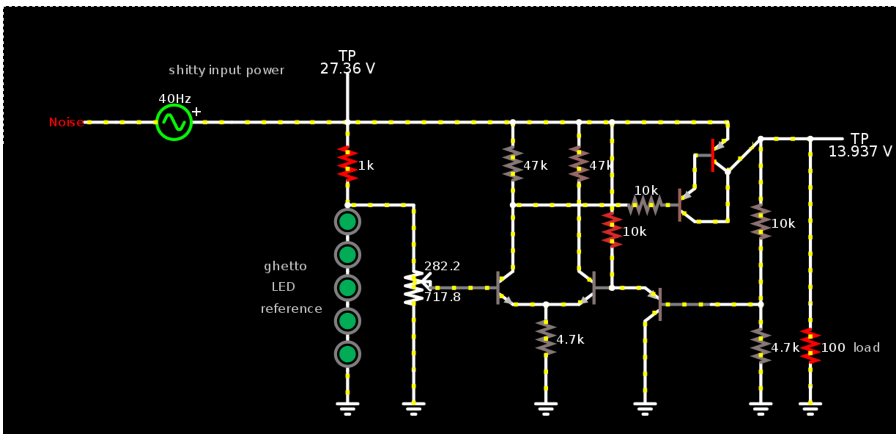

```
$ 1 0.0005 0.41233529972698213 50 5 43
34 fwdrop\q1.7 1 9.32e-11 0.042 2.8457354689914074 0 1
162 48 208 48 240 2 fwdrop\q1.7 0 1 0.5 0.1
g 48 368 48 400 0 0
r 48 208 48 128 0 1000
162 48 240 48 272 2 fwdrop\q1.7 0 1 0.5 0.1
162 48 272 48 304 2 fwdrop\q1.7 0 1 0.5 0.1
162 48 304 48 336 2 fwdrop\q1.7 0 1 0.5 0.1
162 48 336 48 368 2 fwdrop\q1.7 0 1 0.5 0.1
w -64 128 48 128 0
w 48 208 112 208 0
174 112 368 128 208 1 1000 0.21290000000000003 Resistance
g 112 368 112 400 0 0
t 128 288 208 288 0 1 -30.2681886282526 0.561821870800069 100
t 304 288 272 288 0 1 -23.47754064241586 0.6093418296786747 100
w 208 304 240 304 0
w 240 304 272 304 0
r 240 304 240 368 0 470
g 240 368 240 400 0 0
w 208 272 208 208 0
r 208 208 208 128 0 4700
w 208 128 48 128 0
r 272 208 272 128 0 4700
w 272 128 208 128 0
w 272 208 272 272 0
w 304 128 272 128 0
w 496 144 496 304 0
r 496 304 496 384 0 100
g 496 384 496 400 0 0
r 448 144 448 304 0 1000
r 448 304 448 384 0 470
g 448 384 448 400 0 0
```

```
w 304 288 304 256 0
w 448 144 496 144 0
368 496 144 544 144 0 0
368 48 128 48 64 0 0
w 208 208 320 208 0
t 384 160 416 160 0 -1 29.966570072596046 -0.6789517017645963 100
w 352 128 416 128 0
w 416 128 416 144 0
w 416 176 448 144 0
t 352 208 384 208 0 -1 29.406988463009824 -0.5595816095862212 100
w 384 192 384 160 0
r 352 208 320 208 0 10000
w 384 224 416 224 0
w 416 176 416 224 0
t 400 304 336 304 0 -1 0.8622748285616171 -0.684513215849966 100
w 304 128 304 208 0
r 304 208 304 256 0 1000
w 304 288 336 288 0
g 336 320 336 400 0 0
w 400 304 448 304 0
w 304 128 352 128 0
v -176 128 -64 128 0 1 40 5 32 0 0.5
R -176 128 -224 128 0 6 40 5 0 0 0.5
x 537 350 562 353 4 12 load
x -125 82 -14 85 4 12 shitty\sinput\spower
x -33 271 7 274 4 12 ghetto
x -26 291 -2 294 4 12 LED
x -37 312 19 315 4 12 reference
o 33 2 0 4355 80 51.2 0 2 33 3
o 32 2 0 4355 20 12.8 1 2 32 3
```

The LED string at the left provides a constant-voltage reference (though with a shitty temperature coefficient), the potentiometer to the right divides it down to get the reference voltage for the long-tailed pair in the middle which compares the voltage from the pot reference with a feedback voltage from a PNP emitter follower to the right, which buffers an unnecessarily hot voltage divider down from the output. One of the outputs of the differential pair couples through a 10k base resistor to a PNP darlington pass transistor that regulates the output as such.

In this simulation it works really well. The output is regulated very closely (about 40 dB PSRR). In real life it might not work that well.

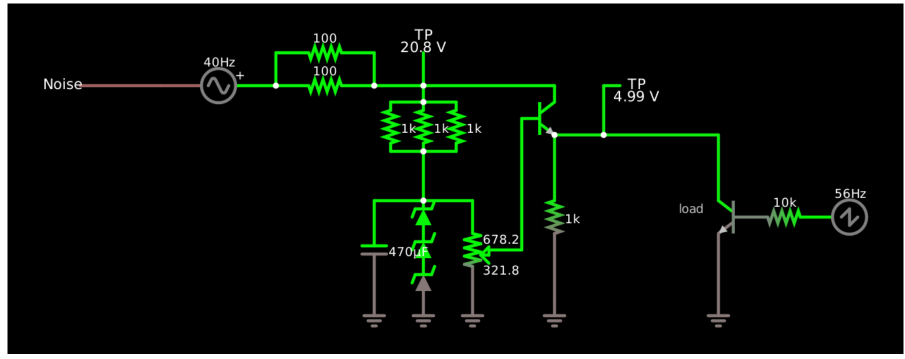
A slightly revised version runs the control loop at much lower current, which should lead to less heat problems, and works almost as well in simulation:




```
$ 13 0.0005 0.41233529972698213 50 5 43
34 fwdrop\q1.7 1 9.32e-11 0.042 2.8457354689914074 0 1
162 48 208 48 240 2 fwdrop\q1.7 0 1 0.5 0.1
g 48 368 48 400 0 0
r 48 208 48 128 0 1000
162 48 240 48 272 2 fwdrop\q1.7 0 1 0.5 0.1
162 48 272 48 304 2 fwdrop\q1.7 0 1 0.5 0.1
162 48 304 48 336 2 fwdrop\q1.7 0 1 0.5 0.1
162 48 336 48 368 2 fwdrop\q1.7 0 1 0.5 0.1
w -64 128 48 128 0
w 48 208 112 208 0
174 112 368 128 208 1 1000 0.7178000000000001 Resistance
g 112 368 112 400 0 0
t 128 288 208 288 0 1 -21.88050572375028 0.5151707707652369 100
t 304 288 272 288 0 1 0.5709371172593585 0.601924362557102 100
w 208 304 240 304 0
w 240 304 272 304 0
r 240 304 240 368 0 4700
g 240 368 240 400 0 0
w 208 272 208 208 0
r 208 208 208 128 0 47000
w 208 128 48 128 0
r 272 208 272 128 0 47000
w 272 128 208 128 0
w 272 208 272 272 0
w 304 128 272 128 0
w 496 144 496 304 0
r 496 304 496 384 0 100
g 496 384 496 400 0 0
r 448 144 448 304 0 10000
r 448 304 448 384 0 4700
g 448 384 448 400 0 0
w 304 288 304 256 0
w 448 144 496 144 0
368 496 144 544 144 0 0
368 48 128 48 64 0 0
w 208 208 320 208 0
t 384 160 416 160 0 -1 13.850113221994937 -0.7228391475035494 100
w 352 128 416 128 0
w 416 128 416 144 0
w 416 176 448 144 0
t 352 208 384 208 0 -1 13.246644166661381 -0.6034690553335551 100
```

```
w 384 192 384 160 0
r 352 208 320 208 0 10000
w 384 224 416 224 0
w 416 176 416 224 0
t 400 304 336 304 0 -1 4.458376683019007 -0.6122710669745723 100
w 304 128 304 208 0
r 304 208 304 256 0 10000
w 304 288 336 288 0
g 336 320 336 400 0 0
w 400 304 448 304 0
w 304 128 352 128 0
v -176 128 -64 128 0 1 40 5 32 0 0.5
R -176 128 -224 128 0 6 40 5 0 0 0.5
x 537 350 564 353 4 12 load
x -125 82 -13 85 4 12 shitty\sinput\spower
x -33 271 9 274 4 12 ghetto
x -26 291 -3 294 4 12 LED
x -37 312 24 315 4 12 reference
o 33 2 0 4355 80 51.2 0 2 33 3
o 32 2 0 4355 80 51.2 1 2 32 3
```

A caveman version using real zeners is this one which will probably burn out a 1k pot:




```
$ 1 0.0005 0.5754602676005731 50 5 43
34 zener-6.1 1 1.7143528192810002e-7 0 2.0000000000000084 6.1 1
z 48 304 48 272 2 zener-6.1
g 48 304 48 320 0 0
z 48 272 48 240 2 zener-6.1
z 48 240 48 208 2 zener-6.1
w 48 208 96 208 0
174 96 208 96 304 1 1000 0.6782 Voltage adjustment knob
g 96 304 96 320 0 0
w 112 256 144 256 0
w 144 256 144 128 0
t 144 128 176 128 0 1 -20.9259922632883 0.7095174649350575 100
r 176 144 176 304 0 1000
g 176 304 176 320 0 0
w 48 96 176 96 0
w 176 96 176 112 0
w 176 144 224 144 0
c 0 208 0 304 0 0.00047000000000000004 18.35536628033426 0.001
w 0 208 48 208 0
g 0 304 0 320 0 0
```

```
368 224 96 256 96 0 0
R -208 96 -304 96 0 6 40 5 0 0 0.5
v -208 96 -96 96 0 1 40 5 30 0 0.5
368 48 96 48 48 0 0
w 0 96 48 96 0
r 0 64 -96 64 0 100
w 224 96 224 144 0
w 224 144 336 144 0
t 384 224 336 224 0 1 -4.3103019007059675 0.7081563777938871 100
w 336 208 336 144 0
g 336 240 336 320 0 0
r 384 224 416 224 0 10000
R 416 224 464 224 0 4 56 5 5 0 0.5
x 297 220 322 223 4 12 load
w 48 96 48 112 0
w 16 112 48 112 0
w 48 112 80 112 0
r 16 112 16 160 0 1000
r 48 112 48 160 0 1000
r 80 112 80 160 0 1000
w 16 160 48 160 0
w 48 160 80 160 0
w 48 160 48 208 0
w -96 64 -96 96 0
w 0 96 0 64 0
r 0 96 -96 96 0 100
o 18 2 0 4355 20 12.8 0 2 18 3
o 21 2 0 4355 40 51.2 1 2 21 3
o 9 2 7 4098 40 0.1 2 1 5
o 26 2 3 4359 10 0.1 3 1
```

Topics

- Electronics (p. 1145) (39 notes)
- Ghettobotics (p. 1169) (12 notes)
- Power supplies (p. 1176) (10 notes)
- Falstad's circuit simulator (p. 1198) (7 notes)

Intel engineering positions considered as a dollar auction

Kragen Javier Sitaker, 02021-01-21 (updated 02021-01-27) (1 minute)

Reading the comment at https://news.ycombinator.com/item?id=25861762 I'm reminded of the grad student dollar auction and the up-and-out rule at top accounting firms: tenure is well paid, secure, and highly competitive, and the way you compete for it is by working for many years as a poorly paid and insecure grad student (and, in some fields, postgrad). And if you don't get a professorship or tenure you get kicked out, first of the university and then possibly out of the university system entirely, which people respond to by throwing good "money" (in the form of work) after bad. So in a sense for this guy Intel was grad school and the bank job was the professorship.

Dollar auctions are widely known for

In many countries, up-and-out is illegal, though with an exception for universities.

Topics

- Incentives (p. 1230) (5 notes)
- Economics (p. 1258) (4 notes)
- Employment (p. 1370) (2 notes)
- Intel

Duplicating Durham's Rock-Hard Putty

Kragen Javier Sitaker, 02021-01-22 (updated 02021-01-27) (1 minute)

Durham's Rock-Hard Putty is a popular home repair product in the US, converting into a cream with water and curing after several minutes to a rock-hard filler; it consists of 70–80% plaster, 5–15% talc, 5–9% dextrin, <1% quartz, and <1% ochre. As noted in Derctuo, alabaster costs US$8 per tonne and calcines to plaster at 100°–150°. Presumably the dextrin is a thickener and the talc is to make it more thixotropic, though maybe it has some chemical effect.

I can't find dextrin locally but food-grade maltodextrin costs US$5/kg at retail (AR$600) and carboxymethylcellulose is about US$9/kg (AR$1400).

Topics

- Materials (p. 1138) (59 notes)
- Pricing (p. 1147) (35 notes)
- Alabaster (p. 1309) (3 notes)
- Thixotropy (p. 1317) (2 notes)
- Talc (p. 1319) (2 notes)

iPhone replacement cameras as 6-μs streak cameras

Kragen Javier Sitaker, 02021-01-22 (updated 02021-12-30)
(2 minutes)

The iPhone 6s specs say it can record 720p video at 240 fps and 1080p video at 120 fps. A friend tells me he bought a replacement camera for US$8, including the flex cable, but it was the front camera, which is only 60fps for 720p.

iPhone cameras are well-known for their rolling-shutter feature, where they scan each line of the image out at a separate time during the frame, leading to visual distortions of rapidly moving objects rather than blur, at least when light is adequate. So rather than considering the back camera as a 240fps camera, it may be reasonable to consider it as a 172800-line-per-second camera, although the gating time (shutter speed) for each line might actually be longer than the 5.8 μs suggested there. Robert Elder reports success recording 660 frames per second on a US$6 Raspberry Pi camera and 1007 frames per second on the V2 camera by reading out the same lines over and over again (so you get a smaller frame at a higher frame rate) but alternatives include using a glass rod to defocus the camera in Y so that every line sees almost exactly the same image, or to use two parallel mirrors to kaleidoscopically replicate a small fraction of the field of view several times.

Streak cameras are an extremely important research tool for investigating all kinds of ultrafast phenomena, such as the time evolution of an arc in air.

Topics

- Contrivances (p. 1143) (45 notes)
- Metrology (p. 1212) (6 notes)
- Cameras (p. 1301) (3 notes)
- Ultrafast

Compiling machine-code loops to pipelined dataflow graphs

Kragen Javier Sitaker, 02021-01-23 (updated 02021-01-27)
(2 minutes)

Occurred to me that if you have a loop in machine code:

```
loop: add r0, r1, r2
      mul r3, r2, r4
      sd r3, foo(r0)
      subi r0, 1
      beq r0, zero, loop
```

you can reasonably build the loop body into a dataflow graph, then perhaps run it in a pipelined fashion at maybe as little as one loop iteration per clock cycle.

What conditions are necessary for this to work? You don't necessarily need a lot of ILP but you do need instructions that aren't inside the core critical path from one iteration of the loop to the next. If it requires a chain of five RTL operations to get the next state of the loop variables from the current state, you aren't going to be able to run the loop at less than five clock cycles per iteration.

The idea here is that you do a sort of "place & route" either upon entry to the loop or after figuring out that you're going to be in the loop for long enough to justify it. Each value you produce as you go through the loop gets assigned to a hardware register with a given ALU function attached to it, with the inputs routed from the inputs to the operation. All branches of a conditional are computed, though some care is needed here to prevent Spectre and also faults; the correct results are selected out with a mux.

What if the loop body doesn't fit in the hardware resources? What about non-innermost loops?

Topics

- Programming (p. 1141) (49 notes)
- Performance (p. 1155) (22 notes)
- Assembly-language programming (p. 1175) (11 notes)

Some preliminary notes on the amazing RISC-V architecture

Kragen Javier Sitaker, 02021-01-24 (updated 02021-07-27)
(29 minutes)

I'm looking at the RISC-V instruction set. It seems kind of boring, on purpose, in a good way, kind of like C and Golang. There's intentionally very little that's clever.

Overall it looks very pleasant, much simpler to learn than amd64, and maybe not even more verbose. You have 32 int registers (though x0 is a hardwired zero) and 32 float registers and a load-store architecture. There's a nascent assembly programmer's manual. You can run it at 400+ MHz in 809 LUTs on a Xilinx 7-series FPGA or even run it on a Lattice iCE40-HX8K (multiple different designs even)or in even fewer LUTs with a slower design, or even with Linux support, and there's a 1.4 GHz full-custom quad-core dev board, a software emulator on amd64 with only a 3× slowdown, and a 108MHz GigaDevice GD32VF103 microcontroller with a US$6 Seeed Studios devboard.

The 145 pages of the user-level instruction set manual plus the 91 pages of the privileged ISA manual, Volume II compare quite favorably to the 262 pages of the MOSTek MCS6500 family programming manual or the 332 pages of the Z80 User Manual.

There are some surprises, though. There's hardware threading ("harts", suggesting that perhaps a memory space should be called a "forest"), with a concurrency mechanism called "LR/SC" that's new to me, and no condition flags. (The opcode listing is not, as I first thought, omitted from the instruction set manual, but consigned to the six-page Chapter 19, plus seven privileged instructions in chapter 5 of Volume II.) The `lui` load-upper-immediate instruction has a 20-bit immediate, so the other immediate-load instructions have only 12-bit immediates. Some of the bit fields in the instruction encoding are scrambled. And it uses NaN-boxing to support multiple floating-point formats on the same chip.

ABI

RISC-V defines a standard ABI as well as the instruction set, because of their experience that more than just an ISA was needed to get the benefits of a large software ecosystem. The calling convention is that the return address is passed in x1, the stack pointer is x2, x3 and x4 are gp "global pointer" and tp "thread pointer" respectively, x5 to x7 are caller-saved temporary registers to to t2, x8 is the (callee-saved) frame pointer fp aka s0, x9 is callee-saved s1, x10 to x17 are argument/return registers a0 to a7, x18 to x27 are callee-saved registers s2 to s11, and x28 to x31 are caller-saved t3 to t6.

The weird split putting s2–11 after a0–7 is apparently to map the arguments into the 8 registers most broadly accessible from RVC compressed instructions (p. 70).

There are alternative conventions for processors with and without

floating-point registers.

It's mostly what you'd expect: C on RV64 is LP64, eight arguments in registers a0 to a7, everything else pushed on the stack, with the earliest non-register argument pushed last to facilitate C varargs. There are a few surprises: C `char` is unsigned. 32-bit unsigned ints are sign-extended to 64 bits on RV64. Alignment padding for stack arguments affects assignment of arguments to registers. Return values of more than two registers are returned in memory, with a pointer to the memory passed as an extra argument prepended to the list. The stack pointer must always be 16-byte aligned even on RV32 and RV64.

Flags and condition codes

I'm surprised that it has no condition-code flags. The authors explain that this was one of their reasons for not using the OpenRISC 1000 instruction set (p. 15, 24 of 145 of the instruction set spec):

> OpenRISC has condition codes and branch delay slots, which complicate higher performance implementations.

Instead there are `beq`, `bne`, `blt`, `bltu`, `bge`, and `bgeu` instructions, which compare two registers and conditionally jump by an immediate ±4KiB offset (p. 17), as done in PA-RISC and the ESP32's Xtensa (p. 18, where the alternatives are discussed).

This seems like it might complicate multi-precision arithmetic; the authors explain a workaround (p. 13, 25 of 145):

> We did not include special instruction set support for overflow checks on integer arithmetic operations in the base instruction set, as many overflow checks can be cheaply implemented using RISC-V branches. Overflow checking for unsigned addition requires only a single additional branch instruction after the addition: `add t0, t1, t2; bltu t0, t1, overflow`.
>
> For signed addition, if one operand's sign is known, overflow checking requires only a single branch after the addition: `addi t0, t1, +imm; blt t0, t1, overflow`. This covers the common case of addition with an immediate operand.
>
> For general signed addition, three additional instructions after the addition are required, leveraging the observation that the sum should be less than one of the operands if and only if the other operand is negative.
>
> ```
> add t0, t1, t2
> slti t3, t2, 0
> slt t4, t0, t1
> bne t3, t4, overflow
> ```

This would seem to imply that, on a straightforward in-order machine, addition and subtraction of multi-precision two's-complement numbers is almost an order of magnitude slower than on a conventional machine with condition codes. The MuP21's approach of having an extra carry bit in its internal CPU registers (21 bits in the MuP21 case, 33 or 65 bits in the RV32I or RV64I case) seems perhaps more reasonable; it would eliminate the concern about complicating higher-performance implementations.

Instruction word format

There are only a small number of instruction layouts (named with letters: **R**egister/register, **I**mmediate/register, **S**tore, **U**pper

immediate, **B**ranch, and **J**ump), which is refreshing, and the choice to reserve two bits in the fixed-width 32-bit format to indicate instruction length brilliantly avoids the complications of ARM Thumb “interworking” between Thumb code and non-Thumb code. I haven’t yet tried to compare the code density of the variable-length RV64IC or RV32IC instruction formats, but I’m optimistic that they will provide Thumb-2-like code density, which would be a unique advantage in the 64-bit world, now that Aarch64 has abandoned Thumb.

The S-type, B-type, and J-type instructions include immediate fields in a slightly weird permutation. In response to the observation that sign-extension was often a critical-path logic-design problem in modern CPUs, they always put the immediate sign bit in the MSB of the instruction word, so you can do sign-extension before instruction decoding is done, but this leads to the J-type format imm[20] || imm[10:1] || imm11 || imm[19:12] || rd || opcode, with a ±1MiB PC-relative range, and an only slightly-less-surprising B-type format, with a ±4KiB range.

They justify this by saying (p. 13):

> By rotating bits in the instruction encoding of B and J immediates instead of using dynamic hardware muxes to multiply the immediate by 2, we reduce instruction signal fanout and immediate mux costs by around a factor of 2. The scrambled immediate encoding will add negligible time to static or ahead-of-time compilation. For dynamic generation of instructions, there is some small additional overhead, but the most common short forward branches have straightforward immediate encodings.

U-type instructions, of which there are only two (`lui` and `auipc`), have a 20-bit immediate field, but I-type and S-type instructions (used for things like `addi` and `slti` and, notably, memory loads and stores) have only a 12-bit immediate field.

ALU instructions

Surprisingly, there are no bit rotates (they suggest on p. 85 that these might be added in the “B” extension) and no abjunction, and multiplication and division are optional extensions.

It supports floating-point, and it spends 22 pages on this, which I am going to comprehensively ignore.

Prologues and epilogues

The hardware calling convention stores the return address in a link register instead of on the stack, and the standard ABI defines quite a lot of callee-saved registers, and there’s no ARM-like store-multiple instruction (p. 72 explains how they considered and rejected this), so a typical prologue is relatively long, for example:

```
addi sp, sp, -12
sw ra, 0(sp)
sw a0, 4(sp)
sw s0, 8(sp)
```

And the epilogue is similar.

There’s an intriguing suggestion on p. 16 about factoring this out

with “millicode”:

The alternate link register supports calling millicode routines (e.g., those to save and restore registers in compressed code) while preserving the regular return address register. The register x5 was chosen as the alternate link register as it maps to a temporary in the standard calling convention, and has an encoding that is only one bit different than the regular link register.

This suggests you could replace the above with something like `jal prologue_a0_s0, x5` or `li t2, 1; li t3, 1; jal prologue_variadic, x5`, which would indeed reduce code size. You could implement ARM-like bitmap-driven “load multiple” and “store multiple” instructions that way.

The S-type encoding used for stores has the same number of bits of each type as the I-type encoding used for instructions such as `addi` and loads, but they are in different places, so that if there’s a destination register, it’s always indicated in bits 11 to 7, and if there are source registers, they are always indicated in bits 24 to 20 and 19 to 15. I guess the idea is to avoid a possible additional level of muxing.

The “RVC” or “C” extension described later also has a couple of instruction formats specifically designed to cut the size of these prologues and epilogues in half.

OS stuff

There’s a page on the OSDev Wiki with an introduction.

There’s a `scall` instruction (now `ecall`) for trapping into the kernel, and three privilege modes: **U**ser mode, **S**upervisor mode, and **M** achine mode. There are correspondingly three versions of `iret`: `uret` from any mode if you have the “N-extension” (not yet finished, p. 101) to enable user-mode trap handlers, `sret` from S-mode (or M-mode), and `mret` only from M-mode.

There are 4096 “CSR”s, control/status registers, 1024 per mode (including 1024 reserved for hypervisors, I guess? There used to be an “H” mode that has now been removed), and accessing one you aren’t allowed to will trap. These include things like floating-point rounding mode and exception state, the trap vector (I guess for setting up interrupt handlers and other trap handlers?), and cycle counters. The minimal set of CSRs is like 12 or 16 bits.

The virtual memory setup seems super simple.

The smallest protection unit is 4-KiB pages, though there’s some kind of large page support; there’s an `addr_space_id` field in the “SATP” CSR (“supervisor address translation and protection”) that specifies which address space you’re in so you can context-switch without flushing your TLB I guess. RV32 has 2 levels of page tables and 32-bit physical addresses, so I guess that’s 1024-way branching at each level (Vol. II, p. 68, §4.3.1); RV64 has 3 or 4 levels of 512-way branching and respectively 39 or 48 virtual address bits (“Sv39” and “Sv48”), and then there’s something called “RSVD” which may or may not be another name for RV128.

An Sv32 (Vol. II, p. 67, §4.3) page table entry is 32 bits; bits 31:20 (12 bits) are “PPN1” (“physical page number”), bits 19:10 (10 bits) are “PPN0”, there are two bits reserved, and then 8 bits DAGUXWRV. XWR are permissions; if all are 0, the PTE is a non-leaf PTE. G is 1 if the PTE is global to all address spaces, and U is 1 if the PTE is

accessible in U-mode. D and A are Dirty and Accessed bits, and "V" is a "valid" bit. The bottom N bits of the "SATP" CSR are the "page table root physical page number" for the current task (whose `addr_space_id` is more of those bits). This is documented in Vol. II, p. 63, §4.1.12.

Interestingly, U-mode pages are normally not readable or executable in supervisor mode, although there's an override bit to make them readable.

There's also a "physical memory attributes" thing (Vol. II, p. 43, §3.5) that sounds like MTRRs, and a "physical memory protection" thing that lets you irreversibly lock some memory regions at boot.

Trap handlers, including interrupts, page faults, and other exceptions, are set up with the "xtvec" CSR (for x in U, S, or M, I guess; vol. II, p. 27, §3.1.7; `stvec` in particular is documented in vol. II, p. 57, §4.1.4) pointing to the trap handler address, or optionally to a table of instructions (if the bottom 2 bits of `xtvec` are set to 01). There is an "xcause" CSR that tells you which interrupt it was, even without the table. And four more related "x*" CSRs. There are currently nine interrupts defined and 12 exceptions: misaligned instruction (0), instruction access fault (1), illegal instruction, (2), breakpoint (`ebreak`, previously `sbreak`) (3), load address misaligned (4), load access fault (5), store address misaligned (6), store access fault (7), environment call (8), instruction page fault (12), load page fault (13), and store page fault (15). The misaligned-address faults are present because, although misaligned fetches are architecturally allowed, you're also allowed to implement them with a trap handler instead of in hardware ☺. (And the same is true of page table traversal.) All the "store" faults may also be "AMO" faults, which I think is "atomic memory operation".

All this is, I think, specified in the RISC-V privileged-instructions spec, which is Volume II.

There's an S-mode `sfence.vma` instruction for, I guess, flushing TLBs — for the current "hart".

Counters, timing, and nondeterminism

There's a 64-bit timer counter (the `time` CSR, I think, 0xC01, p. 108) that can provide an interrupt at a predetermined wall-clock time; each hart has its own comparator, configured in M-mode. *Reading* the timer CSR is *privileged* and traps to M-mode (at least in U-mode), which means you can remove it as a source of nondeterminism from user processes. I'm not sure if the same is true of the performance counters like `instret`, the instructions-retired counter (0xC02), and `cycle` (0xC00); they say (p. 36):

> We mandate these basic counters be provided in all implementations as they are essential for basic performance analysis, adaptive and dynamic optimization, and to allow an application to work with real-time streams. Additional counters should be provided to help diagnose performance problems and these should be made accessible from user-level application code with low overhead.

However, on p. iv of "Volume II: RISC-V Privileged Architectures", they explain that one of the changes from version 1.9.1 to version 1.10 was so that "S-mode can control availability of counters to U-mode".

(I don't know how you can both have a 64-bit timer CSR *and* have only 12 or 16 bits of total CSRs... maybe the dude meant RV32E.)

Hmm, the timer in question actually seems to be the memory-mapped `mtime` register (vol. II, p. 32), coupled with `mtimecmp` which posts a timer interrupt when `mtime` exceeds it.

Aha, here's the poop on the counter enabling (vol. II, p. 34):

> The counter-enable registers `mcounteren` and scounteren are 32-bit registers that control the availability of the hardware performance-monitoring counters to the next-lowest privileged mode. ...
>
> When the CY, TM, IR, or HPMn bit in the `mcounteren` register is clear, attempts to read the `cycle`, `time`, `instret`, or `hpmcountern` register while executing in S-mode or U-mode will cause an illegal instruction exception. When one of these bits is set, access to the corresponding register is permitted in the next implemented privilege mode (S-mode if implemented, otherwise U-mode).
>
> [analogously for `scounteren`]
>
> Registers `mcounteren` and `scounteren` are WARL registers that must be implemented if U-mode and S-mode are implemented. ...
>
> The `cycle`, `instret`, and `hpmcountern` CSRs are read-only shadows of `mcycle`, `minstret`, and `mhpmcountern`, respectively. The `time` CSR is a read-only shadow of the memory-mapped `mtime` register.

So *yes*, the OS (or M-mode code) can hide timers from user code to deny it nondeterministic behavior.

Booting

Julian Stecklina says QEMU boots RISC-V with OpenSBI firmware and can load an ELF kernel with `qemu-system-riscv64 -M virt -bios default -device loader,file=kernel.elf`.

Hardware threading

I don't know what's going on here but there is a `fence` instruction for synchronization betweent threads, a `fence.i` instruction for JIT, but apparently no instructions to spawn or terminate "harts". There's some kind of "IPI" interprocess interrupt mechanism for nonconsensual IPC: you set the USIP bit (or maybe SSIP or MSIP) in another hart with memory-mapped I/O in M-mode, although this is tricky when an OS might be concurrently descheduling a hart.

I never did find anywhere that it says how to start or stop a hart. There's a CSR `mhartid` that tells you what the current hart ID is, so maybe all the harts are running all the time?

RV32E embedded

For embedded systems, like maybe soldering irons, 1024 bits of integer architectural registers might be a lot (??) so they defined a smaller profile with only 16 registers and no counters. I guess that means you don't get a6 and a7, nor s2–11 and t3–6.

The whole chapter about this is only a page and a half long.

RV64I

The 64-bit extension is similar to amd64 in that the normal instructions now work on 64-bit registers, and there are new instructions like `addiw` or `sltiw` which work on only the low 32 bits, and a `lwu` instruction that loads a 32-bit value and zero-extends it. `addiw rd, rs, 0`, to sign-extend a 32-bit value to 64 bits, has an alias `sext.w`.

I suspect that loading a 64-bit non-PC-relative constant will require using an ARM-style "constant pool" rather than the `lui`/`addi` pair needed for 32-bit constants.

This is reasonably compatible, but not totally; it might not be feasible to generate machine code that can do the same thing on either RV64I or RV32I, aside from having some sort of conditional jump. But with the exception of `slli` and `sari` and the like, most of the instructions can just ignore the upper 32 bits if you don't care about them.

Multiplication and division

The standard `mul` operation is only 32×32 → 32, but then there are `mulh`, `mulhu`, and `mulhsu` operations for various ways of computing the high 32 bits of the result. RV64 has a `mulw` as well. Analogously `div` has `rem`, `divu`, `remu`, `divw`, `divuw`, `remw`, and `remuw`, but it does not take a double-precision dividend.

Division by zero or divide overflow does not raise an exception.

Atomics

The atomic instruction set doesn't provide compare-and-swap, or even LL/SC, but rather something called "load-reserved/store-conditional" and an "atomic fetch-and-op" facility. This part seems to be in flux, related to something called "RCsc" or "release consistency".

LR "registers a reservation" on a memory address and reads a word from it; SC writes a word to a memory address, "provided a valid reservation still exists on that address" (p. 40, 52 of 145). I guess if someone else writes to the address, that demolishes your reservation, so your later SC will fail; this is an alternative to CAS that avoids A–B–A bugs, though they say it's more vulnerable to livelock in other designs that aren't RISC-V.

It's not clear whether *read* accesses from another hart to the reservation will cause it to fail.

They give the following implementation of CAS in terms of LR/SC (p. 42):

```
    # a0 holds address of memory location
    # a1 holds expected value
    # a2 holds desired value
    # a0 holds return value, 0 if successful, !0 otherwise
cas:
    lr.w t0, (a0)        # Load original value.
    bne t0, a1, fail     # Doesn't match, so fail.
    sc.w a0, a2, (a0)    # Try to update.
    jr ra                # Return.
fail:
    li a0, 1             # Set return to failure.
    jr ra                # Return.
```

(`jr rs` here is "jump register" (p. 110): `jalr x0, rs, 0`.)

The atomic "fetch-and-ops" are atomic swap, add, and, or, xor,

max, maxu, min, and minu (p. 43), which points out the curious fact that min and max are not provided as standard ALU operations.

They give a three-instruction example of a critical section using `amoswap.w.aq`, `bnez`, and `amoswap.w.rl`.

"RVC" compressed instructions

Chapter 12, p. 67 (79 of 145), explains a Thumb-2-like scheme, providing a 16-bit version of the instruction when:

- the immediate or address offset is small, or
- one of the registers is the zero register (x0), the ABI link register (x1), or the ABI stack pointer (x2), or
- the destination register and the first source register are identical, or
- the registers used are the 8 most popular ones.

It turns out, though, that the last two are actually an "and" rather than an "or", and the conditions are actually considerably more restrictive than the above implies.

There's an opcode map on pp. 81–83.

They point out that the Cray-1 also had 16-bit and 32-bit instruction lengths, following Stretch, the 360, the CDC 6600, and followed by not only ARM but also MIPS ("MIPS16" and "microMIPS") and PowerPC "VLE", and that RVC "fetches 25%-30% fewer instruction bits, which reduces instruction cache misses by 20%-25%, or roughly the same performance impact as doubling the instruction cache size." I'm not sure how that's possible.

There are eight compressed instruction formats.

Much to my surprise, the eight registers accessible by the three-bit register fields in the CIW (immediate wide), CL (load), CS (store), and CB (branch) formats are *not* the *first* eight registers, but the *second* eight registers, x8–15! These are callee-saved s0–1 and the first argument registers a0–5. The CR (register–register), CI (immediate), and CSS (stack store) formats have full-width five-bit register fields. (The CJ format doesn't refer to any registers.)

Complementing the stack-store format are stack-load instructions (p. 71) using the CI format with a 6-bit immediate offset, which is prescaled by the data size (4, 8, or 16 bits). These index only upward from the stack pointer, institutionalizing the otherwise-only-conventional downward stack growth. The immediate-offset field in the stack-store format is also 6 bits and treated in the same way. And there's a thing called `c.addi16sp` which adds a signed multiple of 16 to the stack pointer, that is, allocates or deallocates stack space.

So in a 16-bit instruction you can load or store any of the 32 integer registers to any of 64 stack slots (if you've allocated that many), and you can do a two-operand operation with either two registers or a register and an immediate. It's the more general load, store, and branch formats (CL, CS, CB) that limit you to the 8 "popular" registers and only permit 5-bit unsigned offsets (thus 32 slots indexed by those "popular" registers).

These general CL and CS formats effectively require the register to either be used as a base pointer to a struct or contain a memory address computed in a previous instruction, although you could

reasonably argue that the 12-bit immediate field in the uncompressed I-type and S-type instructions imposes a similar restriction — 2 KiB is not very much space for all of your array base addresses!

Additionally, on RV32C and RV64C, the CIW-format `c.addi4spn` loads a pointer to any of *256* 4-byte stack slots (specified in an immediate argument) into one of the 8 popular registers, which you can then use with a CL or CS instruction to access it.

Unconditional jumps and calls (to ±2KiB from PC) and branches on zeroness (to ±256 bytes from PC) are also encodable in 16 bits, using the CJ format. These are also restricted to the 8 popular registers. There's also `c.jr` and `c.jalr` indirect unconditional jumps and calls, which can use any of the 32 registers except, of course, x0.

There's a couple of compressed load-immediate instructions with a 6-bit immediate operand, of which the second (`c.lui`) seems entirely mysterious.

16-bit-encoded ALU instructions (subtract, `c.addw`, `c.subw`, copy, and, or, xor, and shifts) are all limited to the 8 popular registers, except for addition, which can use all 32 registers.

`ebreak` (into the debugger) is mapped into RVC, which is pretty important, but `ecall`/`scall` isn't.

There doesn't seem to be a reasonable way to load immediate memory addresses in 16-bit code except through the deprecated `c.jal .+2` approach, which leaves the current PC in `ra`, at which point you can add a signed 6-bit immediate to it with `c.addi`, thus generating an address of some constant (or maybe a variable, if your page is mapped XWR or you don't have memory protection!) within 32 bytes of where you are, but then it's still in x1 and not a popular register. There's no compressed version of the `auipc` instruction, for example. This is maybe not such a big deal like it would be in Thumb, since you can freely intersperse 32-bit instructions like that into your 16-bit code.

So in pure 16-bit instructions you can freely walk around pointer graphs, index into arrays, jump around, jump up, jump up, and get down, add, subtract, and do bitwise operations, but you can't invoke system calls or load addresses of global variables or constants.

So you could *almost* do a 16-bit-instruction RISC-V hardware core that emulates other instructions with traps but executes at full speed when running 16-bit instructions. You'd need to add a few additional 16-bit instructions for accessing CSRs, loading addresses, and handling traps.

MMX "P"

On p. 91 they talk about packed SIMD (as in the TX-2 and MMX) which they have decided to support by reusing the floating-point registers for integer vectors (as in MMX) and not support for floating point in favor of Cray-like variable-length vector registers (p. 93). But evidently the packed SIMD proposal is not ready.

2019 update of instruction set

All of the above was from looking at the 2017 2.2 spec. The current version of the user-level ISA spec is 20191213 and is about

another hundred pages, 238 pp. in total.

This answers a bunch of my questions above about hart initiation, why the RV64I *W instructions are the way they are, etc.

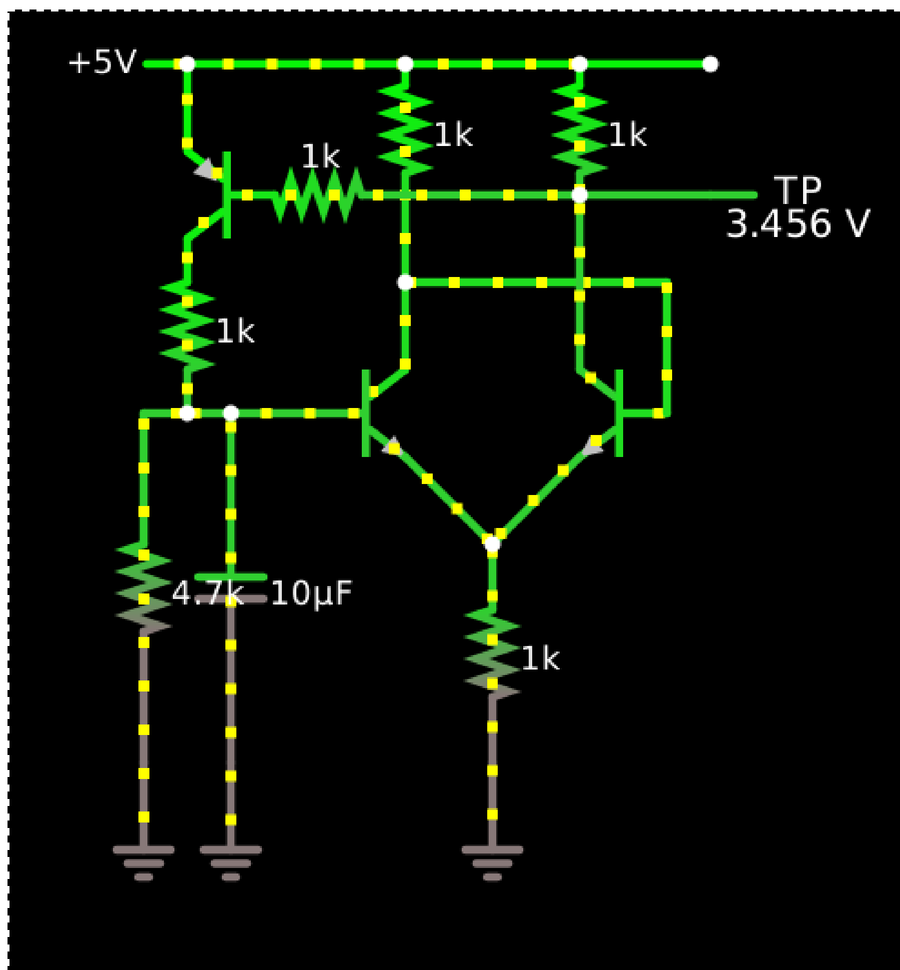
Topics

• Assembly-language programming (p. 1175) (11 notes)
• Instruction sets (p. 1214) (6 notes)
• RISC-V (p. 1276) (3 notes)

Trying to design a simple switchmode power supply using Schmitt-trigger relaxation oscillators

Kragen Javier Sitaker, 02021-01-26 (updated 02021-01-27) (32 minutes)

This Schmitt trigger is a relaxation oscillator:




```
$ 1 0.000005 10.20027730826997 50 5 43
t 128 176 176 176 0 1 -0.2938307156093387 0.33934606572308645 100
t 272 176 240 176 0 1 0.5930756260039916 0.6331767813324252 100
w 176 192 208 224 0
w 240 192 208 224 0
r 208 224 208 304 0 1000
g 208 304 208 336 0 0
w 176 160 176 128 0
w 176 128 272 128 0
w 272 128 272 176 0
w 240 160 240 96 0
w 176 128 176 96 0
r 176 96 176 48 0 1000
r 240 96 240 48 0 1000
```

```
w 176 48 128 48 0
w 176 48 240 48 0
w 240 48 288 48 0
R 96 48 64 48 0 0 40 5 0 0 0.5
w 240 96 288 96 0
368 288 96 320 96 0 0
w 240 96 160 96 0
t 128 96 96 96 0 -1 -0.5925597349297167 -0.6239301896520031 100
w 96 48 96 80 0
w 96 48 128 48 0
r 96 112 96 176 0 1000
c 96 176 96 304 0 0.00001 3.7550188699753013 0.001
g 96 304 96 336 0 0
w 96 176 128 176 0
r 160 96 128 96 0 1000
r 64 176 64 304 0 4700
g 64 304 64 336 0 0
w 64 176 96 176 0
o 18 64 0 36875 10 6.4 0 2 18 3
o 24 64 0 4355 5 0.003125 1 2 24 3
```

It oscillates at about 33 Hz with a 31% duty cycle.

The wire from the differential amplifier's negative output to its negative input provides the positive feedback that makes it a Schmitt trigger, a configuration which I got from the introductory transistor chapter of Horowitz & Hill, so the transistors in the pair are nearly always either in saturation or cutoff. When the left transistor (the positive input) is in saturation, the right one is in cutoff, and so the PNP switch at the top is also in saturation, so the capacitor discharges until the positive-input transistor goes into cutoff, the negative input goes into saturation, and so does the PNP switch, so the capacitor starts charging.

When the positive output is high, the voltage down from the positive power supply to the positive output is nearly 0; the simulation has up to about 50 pA of leakage current through that transistor, so up to about 50 nV of drop through its 1kΩ collector resistor. This keeps the PNP transistor firmly off, allowing the 10μF timing capacitor to discharge through the 4.7 kΩ resistor at a milliamp or two. (It's also bled by the input impedance of the positive input of the differential pair, but only about 30 μA goes that way.)

So we have a discharge time constant of about 47 ms, discharging down from about 4 V (where does this number come from?) toward zero to 2.9 V, at which point it's crossed the 1.1 volts of hysteresis and the Schmitt trigger fires. 2.9 ÷ 4.0 = 0.725, whose natural log is -0.3, so this should be about 0.3 time constants, or about 15 ms, but in the simulation it's more like 9 ms, so I must be analyzing something wrong. The discharge current drops from about 2.4 mA to about 0.6 mA during this time, which suggests it's about 1.4 time constants. So I'm analyzing both the asymptotic value and the time constant wrong.

When the output snaps low, down to about 3.5 volts (where does this number come from?), this is enough to draw about 0.9 milliamps

through the PNP transistor's base resistor, driving it into saturation and charging the capacitor from the 1kΩ–4.7kΩ divider. (We can ignore the 30–40 mV of saturated-transistor voltage drop.)

The Thévenin equivalent of the voltage divider is a source of 4.1 V ($5 \times 4.7/(1 + 4.7)$) with a short-circuit current of 5 mA, and thus a source impedance of about 800 Ω, the parallel combination of the 1 kΩ and 4.7 kΩ, so we'd expect the charging time constant to be about 8 ms. (The cutoff base contributes even less here, just -0.1 pA or so of diode leakage.) The 1.1 volts from 2.9 to 4.0 volts should be about 11/12 of the total distance, leaving 1/12, which should take about 2.5 time constants, or about 20 ms.

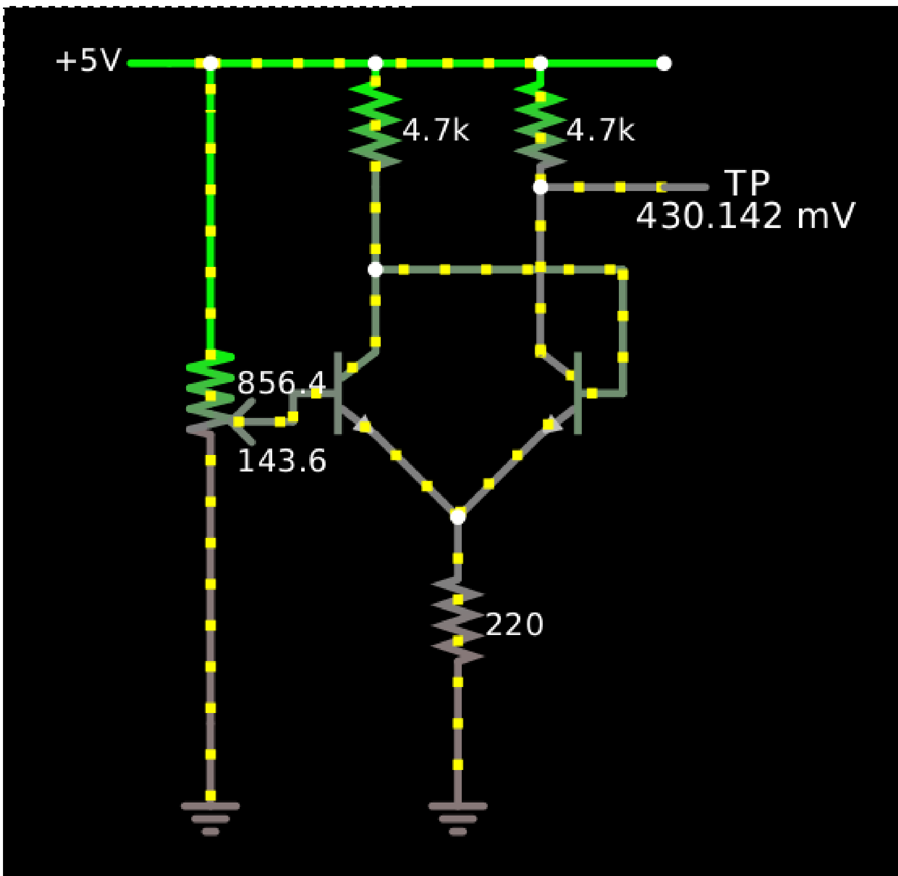
So I'm totally confused about everything. But the circuit does oscillate!

Where does the 3.5 volts come from? It must be 1.5 mA through the collector resistor, which also goes through the shared emitter resistor along with another 0.9 mA from the PNP switch's base current (1.5 V - 0.6 V of Vbe on a 1 kΩ resistor) and the positive-feedback current into the base of the negative-input transistor, which is about another 0.9 mA. This all adds up to about 3.4 mA and thus 3.4 V in the emitter resistor to ground, plus another 40 mV of Vce. So it's basically the voltage divider between the collector resistor and the emitter resistor, perturbed by the additional base currents.

The base current on the negative-input transistor is so large because it's only limited by the positive-input transistor's collector resistor. As long as $\beta > 2.5$ we could get by with a smaller base current. Similarly we only demand about $\beta > 2.2$ on the PNP switch and about $\beta > 1.2$ on the other transistor of the differential pair. We could introduce a 10-kΩ resistor on all three bases without changing the circuit's behavior radically, though it does speed up the oscillation and bring the duty cycle close to ½ and reduce the hysteresis from about 1.1 V to about 0.6 V; and the output is then close to 2.9–3.1 V instead of 3.5. It also increases the amount of time the positive-input transistor is in forward-active mode and consequently makes the circuit behavior dependent on its β, although I'm not sure why.

Trying to get higher gain out of the Schmitt trigger

If we use larger collector resistors, or a smaller emitter resistor, the output voltage swing will be larger, though at the expense of lower input impedance. For example, with 4.7-kΩ collector resistors and a 220-Ω emitter resistor, the output voltage can go down to 430 mV, though with an input impedance of just over 220 Ω, only about 200 mV of hysteresis, and a threshold around 800 mV. Then maybe we can get by without the PNP switch for the oscillator? The idea is that when the positive input transistor switches off, its collector gets pulled high, and the capacitor on its input starts to charge from there. Here's the Schmitt trigger driven from a pot:




```
$ 1 0.000005 1.3241202019156522 60 5 43
t 144 176 176 176 0 1 -0.30258918619292474 0.3178213755193918 100
t 272 176 240 176 0 1 0.5904473435832114 0.6204105617123166 100
w 176 192 208 224 0
w 240 192 208 224 0
r 208 224 208 304 0 220.00000000000003
g 208 304 208 336 0 0
w 176 128 272 128 0
w 240 160 240 96 0
w 176 128 176 96 0
r 176 96 176 48 0 4700
r 240 96 240 48 0 4700
w 176 48 112 48 0
w 176 48 240 48 0
w 240 48 288 48 0
R 96 48 64 48 0 0 40 5 0 0 0.5
w 240 96 288 96 0
368 288 96 320 96 0 0
w 96 48 112 48 0
g 112 304 112 336 0 0
w 272 128 272 176 0
w 176 128 176 160 0
174 112 304 144 64 1 1000 0.1436 Resistance
o 16 64 0 36875 10 12.8 0 2 16 3
o 4 64 0 4355 2.5 0.0125 1 2 4 3
```

At one point, the input voltage is 964 mV and Ib is 601 mA; at 1.113 V Ib is 1.254 mA, thus about 228 Ω for a change of 653 μA over 149 mV; and at 1.204 V Ib is 1.659 mA, so a further 91 mV gives a

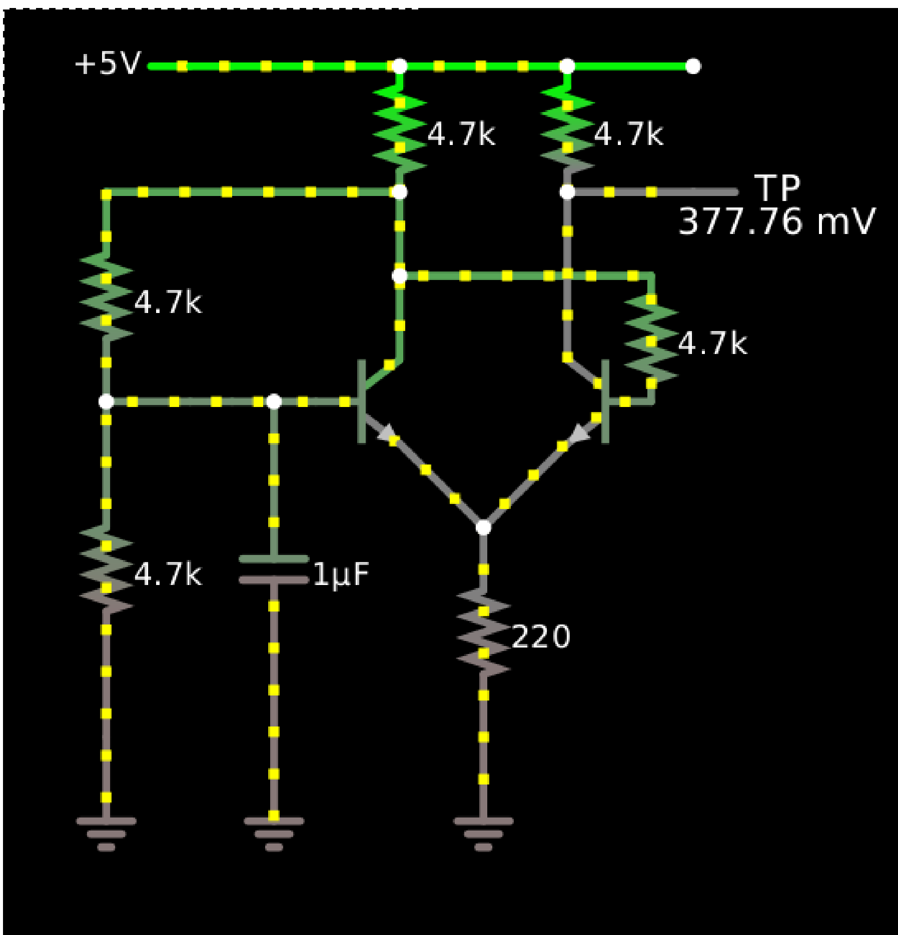
further 405 μA, giving 224 Ω. So the input impedance is just over that of the emitter resistor, because the majority of the emitter current here is base current, and the collector current is almost constant. Moving the input 100 mV higher will move the resistor's voltage almost 100 mV higher, and thus its current as well. So for example at 1090 mV input the base current is 1.149 mA, the emitter is at 464 mV, the collector is at 491 mV, and the emitter resistor is carrying 2.109 mA from those 464 mV, of which the other 0.960 mA come from the collector resistor at 4.509 V. But if I bump up the input to 1113 mV (23 mV higher), the base current goes up to 1.254 mA (as said above, and that's 0.105 mA more), the emitter voltage up to 486 mV (22 mV higher), and the collector voltage up to 512 mV (21 mV higher). This diminishes the current through the collector resistor to 0.955 mA (0.005 mA less), so now the emitter resistor is carrying 2.209 mA, 0.100 mA more.

Can we make the Schmitt trigger oscillate without the extra transistor switch?

So if we rig up an unbuffered capacitor on the input like before, it's going to be pulled down by this 220-Ω input impedance whenever the left-hand transistor is conducting, so it's going to burn down pretty fast. But the feedback output is only going to be able to supply a milliamp or so.

You might think to rely on just the 220-Ω input impedance to discharge the capacitor, then charging it through a simple resistive pullup, but that won't work — the impedance approaches infinity as you get close to the threshold, so the cap discharges close to the threshold but never crosses it, bringing the differential pair into perfect equilibrium.

Further efforts in this direction have not proven fruitful:



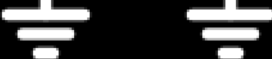

```
$ 1 0.000005 1.3241202019156522 60 5 43
t 144 176 176 176 0 1 -0.9040533288840188 0.5648973225760063 100
t 272 176 240 176 0 1 0.5498868952546583 0.6028520852394035 100
w 176 192 208 224 0
w 240 192 208 224 0
r 208 224 208 304 0 220.00000000000003
g 208 304 208 336 0 0
w 176 128 272 128 0
w 240 160 240 96 0
w 176 128 176 96 0
r 176 96 176 48 0 4700
r 240 96 240 48 0 4700
w 176 48 112 48 0
w 176 48 240 48 0
w 240 48 288 48 0
R 96 48 64 48 0 0 40 5 0 0 0.5
w 240 96 288 96 0
368 288 96 320 96 0 0
w 96 48 112 48 0
g 128 304 128 336 0 0
w 176 128 176 160 0
c 128 176 128 304 0 0.000001 0.889692304649781 0.001
w 128 176 144 176 0
w 128 176 112 176 0
w 112 176 96 176 0
r 64 176 64 304 0 4700
w 96 176 64 176 0
g 64 304 64 336 0 0
```

```
r 272 128 272 176 0 4700
w 176 96 64 96 0
r 64 96 64 176 0 4700
o 16 64 0 36875 2.5 1.6 0 2 16 3
o 4 64 0 4355 1.25 0.00625 1 2 4 3
o 20 64 0 4099 2.5 0.0015625 2 2 20 3
```

Tweaking this like a mindless monkey, I can't seem to get it to actually oscillate. Try as I might, the capacitor just finds a tranquil equilibrium point. I think what's happening is that a single RC circuit isn't sufficient to create enough phase shift to get this amplifier to oscillate: I need unity gain or better at 180°, and a single capacitor only gets me a 90° phase shift. But I don't understand that stuff at all, so I might be talking total nonsense here. You'd think that you could get 180° of phase shift just by using the negative output!

A linear constant-current source

I was thinking that I could make a constant-current source by connecting the output of a simple Schmitt-trigger oscillator like these to some kind of switching transistor, a sense resistor, and some kind of LC filter or something. A very simple non-switching constant-current source, based on what I think is a very widely used design, looks like this:



```
$ 13 0.000005 10.20027730826997 50 5 43
t 256 160 304 160 0 1 -26.012675061853923 0.65192506786543 100
t 304 240 256 240 0 1 -0.65192506786543 0.6059774622652498 100
w 256 160 256 224 0
w 304 176 304 240 0
w 304 240 304 272 0
r 304 272 304 336 0 68
w 256 256 256 336 0
g 256 336 256 352 0 0
g 304 336 304 352 0 0
```

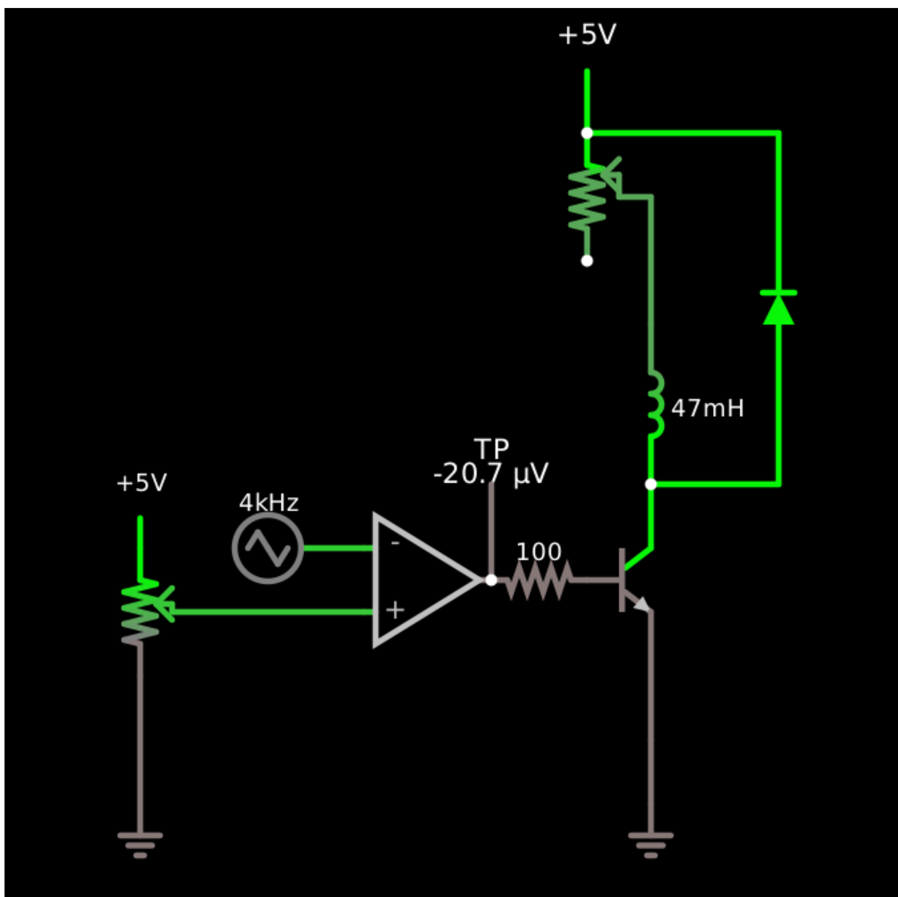
```
r 304 48 304 144 0 1000
R 304 48 304 -16 0 1 40 20 30 0 0.5
r 256 160 256 48 0 22000
w 256 48 304 48 0
368 304 144 368 144 0 0
368 304 48 384 48 0 0
368 304 240 352 240 0 0
o 9 64 0 4099 10 0.0125 0 2 9 3
o 14 64 0 4099 80 51.2 1 6 14 3 13 0 13 3 15 0 15 3
```

In this simulation, the input voltage across the 1-kΩ load varies between +10 V and +50 V. The load is grounded through the collector of the pass transistor, whose emitter has a 68-Ω sense resistor to ground, which in parallel with the base-emitter junction of a feedback transistor. The collector of the feedback transistor robs current from the base of the pass transistor, which is pulled up to the source with a 22-kΩ resistor. So the feedback transistor maintains the sense resistor at 600 mV or so (570–620 mV in the simulation) by reducing its collector current whenever its base voltage starts to drop, which allows the base voltage on the pass transistor to soar.

The upshot is that the load current is held constant within about ±5% despite 5× variations in the input voltage, but the pass transistor burns 2× as much power as the load on average, 4× at peak.

Regulating current efficiently with PWM

What I was thinking was that if you could feed the pass transistor a PWM signal, then filter its collector with an LC filter, you could regulate the load current without dissipating all that power. I can't figure out how to connect a constant-current LC filter to a transistor without either blowing up the transistor with a voltage spike when it turns off, or blowing it up with a current spike when it turns on, so for the moment I'm trying to make do with a diode:




```
$ 1 0.000005 0.28339363076941687 50 5 43
t 304 272 336 272 0 1 -5.2088354545247935 -0.000020699986477813047 100
w 336 288 336 352 0
w 336 352 336 384 0
g 336 384 336 400 0 0
a 192 272 256 272 8 5 0 1000000 3.3999999647789307 3.1930000000000005 100000
R 304 48 304 0 0 0 40 5 0 0 0.5
174 304 48 304 112 0 100 0.1337 load
w 320 80 336 80 0
w 336 80 336 144 0
l 336 144 336 224 0 0.047 0.21955359875043728
w 336 224 400 224 0
34 power-schottky 1 0.0001714352819281 0 1.1281915331325552 0 5
d 400 224 400 48 2 power-schottky
w 400 48 304 48 0
w 336 224 336 256 0
174 80 320 96 272 0 1000 0.6386000000000001 current setting
g 80 320 80 400 0 0
w 96 288 192 288 0
R 192 256 144 256 0 3 4000 2.5 2.5 0 0.5
R 80 256 80 224 0 0 40 5 0 0 0.5
r 256 272 304 272 0 100
368 256 272 256 208 0 0
o 9 2 0 4099 5 0.4 0 2 9 3
o 20 2 0 4098 10 6.4 1 2 20 3
```

The comparator in this circuit compares the current input setting to a triangle wave, thus generating a PWM waveform to use to switch a power transistor between saturation and cutoff. The collector of the

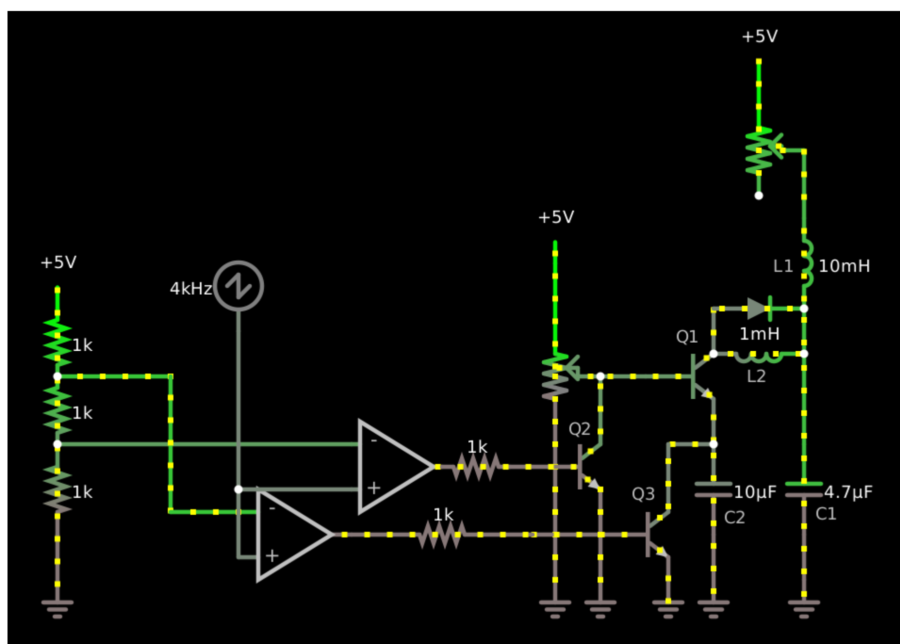
power transistor grounds one end of the inductor some of the time, while otherwise allowing it to return to the positive voltage supply through the Schottky diode.

Thus analyzed, this is just an ordinary buck converter drawn in a slightly weird way, and its variable-current output isn't really a current output at all; with a constant PWM duty cycle and supply voltage, it will give you a constant *voltage*, not a constant current. To avoid unboundedly increasing current, the inductor's time-averaged voltage must be zero (ignoring parasitic resistance) so the average voltages at its two ends must be equal. But that wasn't what we wanted; we wanted to set a *current* and have the sink produce that current.

A Rube Goldberg linear current sink

Now, we could achieve that with feedback, as suggested earlier: adjust the duty cycle of an oscillator according to a current for example as indicated by the voltage across a sense resistor. But is there an inherently regulated way?

Well, a capacitor does have the property that its time-averaged current is zero, but by itself that doesn't give you what you want. But you could reasonably charge a capacitor up to a desired voltage and then dump it to ground at a variable frequency in order to get a frequency-to-current converter, or charge it up to a variable voltage and then dump it to ground at a fixed frequency:



This circuit does not really work. The idea here is that L1 and C1 smooth the current from the load, while Q1 allows C2 to charge up to some variable voltage, set by the potentiometer. Then, at some fixed frequency (4 kHz here), Q2 pulls Q1's base to ground, turning it off, and then Q3 shorts C2 to ground. If the potentiometer is at, say, 900 mV, then Q1 will allow C2 to charge to 300 mV, thus accumulating 3 μC of charge, and so at 4 kHz this circuit should sink 12 mA for a wide range of load impedances. L2 and its flyback diode are intended to limit the current with which C2 charges, so that when Q2 turns off and Q1 turns back on, Q1's collector current isn't excessive. The two comparators, the sawtooth, and the voltage

divider handle the sequencing of Q2 and Q3.

And all of that sort of works in the simulation. The regulation isn't perfect: there's ripple, there's ringing, there's a lot of overshoot, and it can even reverse the voltage to the load (which would often be a fatal flaw in real life), but the real problem is what happens at Q3. Because it's trying to short out a charged capacitor, it has to dissipate all the energy in the capacitor — whatever base current you give it will never be enough for it to start out in saturation, because an ideal capacitor can supply infinite current.

So this is really just a Rube Goldberg variable linear current sink, similar to the two-transistor one given earlier, except that it's burning up the wasted energy in pulses instead of continuously. And that's the big sense in which it doesn't really work.

Essentially the problem is that, if this current sink is connected between the load and ground, and its load terminal is at +3 V dc from ground, and it's sinking an average of 12 mA dc, then it's receiving 48 mW of energy through that terminal. That energy has to go somewhere; it can't just disappear.

A slightly less broken Rube Goldberg switching current source?

This suggests a straightforward way to solve the problem: instead of trying to sink current from the load, let's source current to the load via Q3! This has the problem that, when we try to dump C2 into the load via Q3, we aren't dumping a known amount of charge; we're only discharging C2 down to the voltage of the load's input terminal. I tried some things here but I'm not confident that it works at all; I think it may just have fooled me:

```
$ 1 0.000005 18.278915558614752 40 5 43
R 272 144 272 96 0 0 40 15 0 0 0.5
174 496 272 496 336 0 100 0.8564 load
w 512 304 528 304 0
w 528 304 528 368 0
174 160 320 176 176 0 1000 0.08420000000000001 current setting
g 160 320 160 400 0 0
R 160 160 160 128 0 0 40 15 0 0 0.5
t 224 240 272 240 0 1 -14.954239010668287 -3.889054598302753 100
w 272 224 272 192 0
d 272 192 272 144 2 default
c 272 288 272 352 0 0.00001 3.9348155877184956 0.001
g 272 352 272 400 0 0
w 192 240 176 240 0
w 192 240 192 288 0
t 128 304 192 304 0 1 0.631950629878116 0.6777116192938586 100
w 224 240 192 240 0
w 272 256 272 288 0
w 272 288 240 288 0
w 240 288 240 336 0
t 144 352 240 352 0 1 -3.937246095539954 0.558406250396508 100
g 528 368 528 400 0 0
a 16 304 80 304 8 5 0 1000000 1.6666666666666667 2.8000000408304526 100000
a -64 352 16 352 8 15 0 1000000 3.3333333333333335 2.8000000408304526 100000
```

```
g 192 320 192 400 0 0
r 128 304 80 304 0 1000
r 144 352 16 352 0 1000
R -192 192 -192 160 0 0 40 5 0 0 0.5
r -192 192 -192 240 0 1000
r -192 240 -192 288 0 1000
r -192 288 -192 352 0 1000
g -192 352 -192 400 0 0
w -192 288 16 288 0
w -192 240 -112 240 0
w -112 240 -112 336 0
w -112 336 -64 336 0
R -64 208 -64 176 0 4 4000 2.5 2.5 0 0.5
w -64 208 -64 320 0
w -64 320 16 320 0
w -64 320 -64 368 0
l 496 208 496 272 0 0.1 0.00900069879531614
x 245 216 262 219 4 12 Q1
x 214 327 231 330 4 12 Q3
x 169 281 186 284 4 12 Q2
x 279 344 295 347 4 12 C2
x 266 166 281 169 4 12 L1
w 240 368 416 368 0
w 416 368 416 208 0
l 288 144 288 192 0 0.01 1.0027900861914286e-13
w 416 208 496 208 0
w 496 208 576 208 0
w 528 368 576 368 0
d 576 368 576 208 2 default
w 272 192 288 192 0
w 288 144 272 144 0
o 14 1 2 4099 5 0.1 0 3 19 2 7 2
o 3 2 0 4097 5 0.1 1 2 3 3
```

An approach that will definitely work:

PWM with feedback

Returning to the previous PWM circuit, it should be straightforward to make it regulate current rather than voltage by adding negative feedback.

Here’s a messy version:

```
$ 1 0.000004999999999999996 0.28339363076941687 50 15 43
t 304 272 336 272 0 1 0.6260511746778811 0.674755596023026 100
g 336 384 336 400 0 0
a 192 272 256 272 8 5 0 1000000 3.800000004678905 4.126251457841372 100000
R 304 48 304 0 0 0 40 15 0 0 0.5
174 304 48 304 112 0 1000 0.8267000000000001 load
w 320 80 336 80 0
w 336 80 336 144 0
w 336 224 336 256 0
w 96 288 192 288 0
R 192 256 144 256 0 3 4000 2.5 2.5 0 0.5
r 256 272 304 272 0 1000
368 256 272 256 208 0 0
r 336 320 336 384 0 47
w 336 288 336 320 0
a 96 336 0 336 9 5 0 1000000 0.8613930098335231 0.8614248813748869 100000
w 0 336 0 384 0
w 96 352 96 384 0
r 96 384 0 384 0 2700
r 96 384 96 432 0 1000
g 96 432 96 448 0 0
R -336 -32 -336 -48 0 0 40 5 0 0 0.5
174 -336 64 -304 -16 1 1000 0.5 Current setting
g -336 64 -336 80 0 0
w -304 16 -80 16 0
r 96 320 336 320 0 1000
r -80 16 96 16 0 1000
c 96 16 160 16 0 1e-7 2.500000000000011 0.001
g 160 16 160 48 0 0
w 304 48 384 48 0
d 384 224 384 48 2 default
w 384 224 336 224 0
l 336 144 336 224 0 0.22000000000000003 0.014864165387393801
d -64 336 -128 336 2 default
c -128 336 -128 432 0 0.000001 2.534996346391612 0.001
g -128 432 -128 448 0 0
w -128 336 -160 336 0
```

```
r -160 336 -160 432 0 1000
g -160 432 -160 448 0 0
r 0 336 -64 336 0 33
a -64 176 -208 176 9 5 0 1000000 2.534996346391612 2.500000000000011 100000
w 96 16 96 80 0
w 96 80 48 80 0
w 48 160 -64 160 0
w -128 336 -128 256 0
w -128 256 -64 256 0
w -64 256 -64 192 0
w 48 80 48 160 0
r -208 176 -208 272 0 1000
c -208 272 -208 320 0 0.000001 4.126251457841372 0.001
g -208 320 -208 352 0 0
w -208 272 96 272 0
w 96 272 96 288 0
o 11 2 0 36866 10 6.4 0 2 11 3
o 31 2 0 4099 20 0.1 1 2 31 3
o 15 2 0 4098 10 0.00625 2 2 15 3
o 33 2 0 4099 5 0.0125 3 2 33 3
```

I struggled a lot to get this to work, battling lots of unexplained convergence problems, and hopefully I can do a better version later.

When the switching transistor is on, it passes the current through the 47-ohm sense resistor, which converts each milliamp into 47 mV. The first op-amp multiplies that by 3.7 to get 173.9 mV/mA, and then a peak-detector diode lops about 520 mV off of that. So 18 mA gives you 3.1 volts at the output of the opamp, and 2.58 volts at the output of the diode. (The inrush-limiting 33-Ω resistor adds an additional error, about 130 mV in this case.) The peak-detecting capacitor is discharged to ground with a 1-ms time constant, so the 250-μs period of the 4-kHz PWM signal is enough for about a 22% decay in the worst case of a very low duty cycle, so on average the voltage here is about, say, 8% lower than the peak. So our 2.4 volts or so goes to the opamp above to be compared with the set point from the potentiometer, and this comparison is also filtered with a 1-ms-τ RC filter. The duty cycle with which the ripply input signal falls below the set point is then fairly directly used as the duty cycle for the comparator with the sawtooth.

This does overshoot and ring, and it has residual error, but it doesn't work nearly as badly as it sounds like it ought to. I wanted to wedge an error integrator in there somewhere in order to eliminate the residual error, but I kept getting convergence failures in the solver.

Another problem with this circuit is that the base current on the switching transistor is included in the current measurement. This is a big problem at low duty cycles; in the example circuit, with the load set at 114 Ω, we achieve about 7–9 mA through the load at about an 8% duty cycle. But when the switching transistor is turned on with a 5-V signal, there's another 3.8 mA of base current which gets added, so the sense resistor actually measures 11.5 mA. Turning the load up to 668 Ω, the duty cycle stabilizes around 30%, with 8.2 mA through the sense resistor, so we have about 29% residual error due to

proportional control — but the current through the actual load is only about 4 mA, more like a 50% error!

This base-current error could be corrected either by running the sense resistor on the opposite side of the load from the switching transistor, by putting the sense resistor between the load and the switching transistor (which would probably require differential measurement), or by reducing the base current when operating the load at low currents.

(And of course the ultimate objective is to simplify the whole circuit to about seven transistors by using the oscillators presented at the top of this note, feeding them some kind of signal to modify their duty cycle, produced through the kind of PI control I totally failed to achieve in this case.)

Topics

- Electronics (p. 1145) (39 notes)
- Power supplies (p. 1176) (10 notes)
- Falstad's circuit simulator (p. 1198) (7 notes)
- Oscillators (p. 1283) (3 notes)

Trying and failing to design an efficient index for folksonomy data based on BDDs

Kragen Javier Sitaker, 02021-01-26 (updated 02021-01-27) (7 minutes)

Suppose you have some tags and want to index them. My URLs file is only about 4500 URLs, so for an interactive query sequential search would be fine, but what if it wasn't, because you had a much larger database?

Each item has some set of tags. A tag query consists of some set of required tags and some set of forbidden tags, and the desired result is the set of items that have all the required tags and none of the forbidden tags; usually we want to be able to start iterating over the result set as soon as possible.

So far this sounds trivial. What makes it difficult is the Zipf distribution and non-independence of tags.

I have about 2200 tags in my 5000-item URLs file, 1100 of which are used only once; for example:

```
  1 #3D-XPoint
  1 #Alex-Jones
  1 #random-forest
  1 #dimensional-analysis
  1 #QR-codes
  1 #Data-General
  1 #CCC
  1 #digital
  1 #aurora
  1 #Steven-Universe
```

These are easy to handle: if they're in the required set, a simple inverted index directs you from the tag directly to its single hit, and if they're in the forbidden set, it's adequately efficient to check each tag in candidate items against a hash table of forbidden tags.

But there are some tags that are used frequently:

```
345 #hardware
254 #paper
217 #politics
174 #history
154 #USA
144 #materials
136 #PDF
133 #pdf
120 #security
118 #toread
112 #video
103 #algorithms
```

```
100 #ebook
 91 #human-rights
 91 #energy
 83 #performance
```

So, for example, #hardware has about 7% selectivity, and #paper about 5%. Worse, these tags aren't independent: only about 5% of all URLs are tagged #paper, but 72 of the 133 #pdf URLs are, over 50%, according to

```
grep '#pdf' urls | perl -lne 'print $1 while /\s(#[-\w]+)/g' |
    sort | uniq -c | sort -n
```

The set of all tags commonly grows almost linearly with corpus size, but my intuition is that the set of *popular* tags like #politics grows much more slowly. The first 2200 URLs contain 1366 distinct tags (4613 total), 0.62 per URL, while all 4578 URLs contain 2175 distinct tags (10269 total), 0.48 per URL.

Whether this is true depends on how you define "popular". If we take an arbitrary cutoff point of 100, the full set had the 13 "popular" tags above, while the first 2200 had only one "popular" tag (#paper). If instead we say that a "popular" tag is one with 1% selectivity or worse, then in the smaller set there were 25 "popular" tags, and in the larger set there were 30. If we draw the line at 0.5%, it's 86 and 79.

We can build a tree over just the "popular" tags that enables efficient enumeration of query results by a subdivision process. Each internal node has an associated tag, with one child for a subtree of items containing that tag, and another child for all other items. Several nodes may be associated with the same tag. Leaf nodes are associated with small sets of items.

The greedy approach

We build the tree as follows. First, if the set of items is too large for sequential search to be desirable, take the tag that produces the most even split between hits and misses, the one whose selectivity is closest to 50%, #hardware above, and split the items. Then repeat the process recursively on the resulting two subsets. So, for example, within #hardware we have the following popular tags:

```
345 #hardware
 26 #simulation
 22 #Arduino
 19 #PDF
 18 #reverse-engineering
 18 #history
 16 #display
 16 #cellphone
 16 #AVR
 15 #RISC-V
```

Here, #hardware no longer has useful selectivity (it's 100%) but now we have #simulation with 7.5% selectivity, so (if for the sake of argument 345 is not small enough yet) we subdivide according to

whether #simulation is present or absent.

In the non-#hardware group, the ordering is pretty much unchanged, except that the spuriously separate #PDF and #pdf tags have switched places, #video has moved down the list, and #Trump has moved up:

```
242 #paper
217 #politics
156 #history
154 #USA
141 #materials
124 #pdf
117 #PDF
111 #security
107 #toread
103 #algorithms
101 #video
 99 #ebook
 91 #human-rights
 89 #energy
 79 #Trump
 76 #performance
```

So we would subdivide with #paper; -hardware +paper is divided into -pdf and +pdf, while -hardware -paper is divided into -politics and +politics. -hardware -paper -politics is unsurprisingly divided into -history and +history, while +hardware +simulation is unsurprisingly divided into -SPICE and +SPICE. And so on.

The same tag can occur in more than one place; for example, both -hardware +paper +pdf and -hardware +paper -pdf +PDF are divided by the tag #toread.

To evaluate a query, we begin by querying the inverted index for all the required tags; if any of them have an adequately small result set for sequential search to be practical, we iterate over that result set. If not, then all of our required tags are "popular", so we begin to traverse the tree top-down. When the tag associated with a tree node is in the query, either as a required or a forbidden tag, we only visit one of its children; otherwise, we visit both.

Ultimately each of the leaf nodes of this tree contains individually a reasonable number of items to search, although this doesn't necessarily imply that the aggregate total of all the leafnodes to search will necessarily be reasonable.

Why the greedy approach fails

A difficulty with this approach is that in some cases we will end up reading through an entire subtree because a tag, though globally popular, is unpopular within that subtree. For example, the intersection of #hardware and the lamentable #Trump tag is empty, since Trump doesn't know about hardware, so for the simple query +Trump, the algorithm will have to sequentially examine all 345 #hardware items. Even if there were one or two #Trump items in there in a much larger set, it would never be a popular tag within that

context.

A non-greedy approach: BDD-like consistent choice order

An alternative is to use a consistent order of tags throughout the tree: divide the root by #hardware, its two children (if neither is small enough to be a leafnode) by #paper, their four children (except small ones) by #politics, and so on. This avoids the catastrophic worst case described above, but it probably means that the mechanism can't handle more than about 4–8 popular tags.

Topics

• Programming (p. 1141) (49 notes)
• Performance (p. 1155) (22 notes)
• Algorithms (p. 1163) (14 notes)
• Facepalm (p. 1199) (7 notes)
• Databases (p. 1376) (2 notes)
• Folksonomies

The use of silver in solar cells

Kragen Javier Sitaker, 02021-02-02 (updated 02021-09-11)
(8 minutes)

A significant fraction of global silver production is used for photovoltaic modules, and this accounts for something like 10% of their cost, and something like 10% of global silver production, at present, a number which is likely to grow. The USGS's silver report doesn't mention photovoltaics, but of course photovoltaics aren't being produced in the US now. It says apparent consumption in the US was 6500 tonnes in 02019 out of 27000 tonnes of global world production.

(As a side note, it says photography use in the US is down to 3% of total silver consumption, down from 28% in 01999, which is still almost 200 tonnes.)

Current situation

In PV panels, as of 02018, conductive silver paste is used to make electrical connections to the photocells at a loading of 130 mg Ag per 4.7-watt cell, down from 400 mg in 02007. The Silver Institute anticipates that cells will grow to 6 watts by 02030 and reduce silver usage to 65 mg per cell by 02028. (I'm assuming these numbers are peak watts because that's how cells are normally sold; the capacity factor varies wildly depending on where you install them, making it not only anticommercial but impractical to derate them for an expected capacity factor.)

In 02020 about 140 GWp of new PV capacity was installed, doubling about every three years, bringing the total to about 770 GWp. 130 mg / 4.7 Wp is 28 mg/Wp, so those 140 GWp amount to 3.9 billion grams or 3900 tonnes of silver consumption, 14% of world silver production. Perhaps some of the anticipated reduction from 02018 has already happened; apparently from 02007 to 02018 the trend of silver intensity was about a 4.3% decline per year, so if that had continued for two more years, in 02020 it would be down to 119 mg per cell.

This 3900 tonnes per year is almost double the 70-million-ounces number given in the article cited above: "CRU experts forecast silver demand for the PV industry of around 70 to 80 million ounces per year until a decline to between 50 and 55 million ounces in the mid-2020s. Only by 2030 is demand expected to recover, to approximately 66 million ounces per year." 70 million troy ounces is almost 2200 tonnes. They also cite historical figures:

> According to a report published by the Silver Institute in April, global industrial demand for silver grew around 4%, from 5,768 million ounces in 2016 to 5,990 million last year. This spurt was mainly due to the record growth of the PV industry, which pushed demand for silver as a component of silver pastes for solar cells, from 79.3 million ounces in 2016, to 94.1 million ounces in 2017 – year-on-year growth of around 19%.

In SI units, 5768 million troy ounces is 179400 tonnes, almost seven times the number the USGS gives for global annual silver production, including that sold to investors and that used for non-industrial uses

such as jewelry; 5990 million troy ounces is 186300 tonnes; 79.3 million troy ounces is 2470 tonnes; and 94.1 million troy ounces is 2930 tonnes. (If we were to go further into SI units, 2930 tonnes per year is 92.7 grams per second.)

Clearly the article is in error about global industrial demand for silver, both because it's an order of magnitude too high and because an increase of 222 million ounces per year cannot be "mainly due to" an increase of 15 million ounces per year.

2930 tonnes ÷ (130 mg / 4.7 Wp) gives an estimate of 106 GWp of PV cells manufactured in 02016, which is of the right order of magnitude but about 38% higher than the Wikipedia estimate cited above of 76.8 GWp of new PV capacity installed in 02016. One or the other number seems likely to be in error; there's no way that 28% of the year's global PV manufacturing product was stored in warehouses or in transit at the end of the year. (Or, one supposes, on jobsites.)

Though, as I write this, silver prices have spiked to about US$26/troy ounce (US$0.85/g), the average price for several years has been closer to US$16/troy ounce (≈US$0.50/g). As of 02021-01-27 a polycrystalline solar module in China costs US$0.167/Wp on the spot market (one-week average). (PVXchange cites €0.16/Wp for "low cost" modules; at €1.20/US$ that's US$0.19/Wp, about 15% higher.) Multiplying out US$16/troyounce × 119 mg / 4.7 Wp gives US$0.013/Wp of silver, which is about 7.8% of the spot price of the entire module. The cost of mono PERC modules and poly PERC modules are given as US$0.183/Wp and US$0.190/Wp, while the corresponding cell prices are given as US$0.086/Wp and US$0.116/Wp, suggesting that the rest of the module costs in the range of 7¢–10¢ per watt, including the silver paste.

Expected outcomes

There will be continued growth in solar PV will increase global silver demand; the expected 25% or so growth in PV installations in 02021 (continuing the trend of doubling every three years) would increase total global silver demand by about 4% this year, but if doubling were to continue at this rate, PV would consume all of current world silver consumption by the early 02030s.

For a few years investors may cushion the price effects of such demand increases, but the price will probably go higher.

Because, like indium and gallium, most silver is not mined from silver mines, but rather as a byproduct of other mining, it's likely to have a fairly inelastic supply; the price would have to go extremely high before the minimal amounts of silver produced as a byproduct in, for example, a zinc mine, would justify increasing the mine's production. So silver prices can probably go quite high before increased mining limits their rise.

Solar cells that can accept the lower efficiencies that accompany the use of copper-filled conductive paste rather than silver-filled paste will do so, due to copper's much lower price and much more elastic supply. More efficient solar cells using silver will increase in price and experience continued pressure to reduce silver usage.

Old solar cells with larger amounts of silver will become increasingly attractive recycling targets thanks to their high content of now-more-valuable silver. Even at today's US$26/troy ounce price, the 400 mg in a solar cell from 02007 is worth US$0.33, almost half the price of a replacement module (assuming both use 4.7-watt cells). If prices shoot up to US$50 per troy ounce, the silver in the module is probably worth more than the module itself, even aside from the additional possible profits from recycling the silicon; at US$100 per troy ounce there will be bandits roaming the countryside to mine silver from solar cells.

Ultimately silver will probably be abandoned in favor of more sophisticated patterning techniques than screen printing, which will permit the use of copper or aluminum with no loss in efficiency. Fractal-like branching patterns and conductors with high aspect ratios (height divided by shadow width) require more precise control over manufacturing, but can reduce the shadow and surface recombination losses to almost arbitrarily low levels.

Topics

- Materials (p. 1138) (59 notes)
- Pricing (p. 1147) (35 notes)
- Manufacturing (p. 1151) (29 notes)
- Energy (p. 1170) (12 notes)
- Solar (p. 1203) (6 notes)
- Minerals (p. 1210) (6 notes)
- The future (p. 1220) (5 notes)
- Economics (p. 1258) (4 notes)
- Silver (p. 1328) (2 notes)

Snap logic, revisited, and four-phase logic

Kragen Javier Sitaker, 02021-02-08 (9 minutes)

In Derctuo I wrote a note about “majority DRAM logic” about logic elements consisting of two CMOS inverters in a latch, shorted by a pass transistor, as a sort of differential ampifier. It occurred to me today, though, that if for some reason you had to build digital logic out of discrete components, a simpler version involving two transistors rather than five might be more useful: an RTL latch.

The NPN version of this is rather simple: Vcc-(100k-b-NPN[Q1, C=a]||82k-a-NPN[Q2, C=b])-GND, where a and b are two points to which the collectors of the opposing resistors connect. In Falstad’s circuit simulator:

```
$ 1 0.000005 10.20027730826997 66 5 43
R 192 96 192 32 0 0 40 5 0 0 0.5
r 192 96 192 192 0 100000
t 304 304 384 304 0 1 -0.519462912774937 0.02791600975110754 100
r 384 96 384 192 0 82000
t 288 304 192 304 0 1 0.519462912774937 0.5473789225260445 100
w 192 192 192 288 0
w 384 192 384 288 0
w 288 304 384 192 0
w 304 304 192 192 0
g 192 320 192 352 0 0
g 384 320 384 352 0 0
R 384 96 384 32 0 0 40 5 0 0 0.5
s 384 192 464 192 0 1 true
g 464 192 464 352 0 0
s 112 192 192 192 0 1 true
g 112 192 112 352 0 0
o 6 64 0 4097 1.25 0.000390625 0 4 6 3 5 0 5 3
```

This simple simulation, just two resistors and two transistors, had a metastability problem where both transistors were on in forward-active mode. If either ever gets into saturation, for example because you press the other transistor’s switch, it pulls the base voltage of the other down into cutoff, which supplies further base current to the already-saturated transistor. But it doesn’t find its way there if it's ever in this initial balanced DC operating point.

(This is somewhat worrisome, because I’d like to ensure that this equilibrium, which must of course exist, is an unstable one: the positive feedback coefficient around the operating point ought to be more than 1. But I suspect that is not the case, and my attempts to solve it by aimlessly adding resistors have failed. It is of course possible to solve the problem with more transistors, but I feel that it should be possible to fix it with resistors…)

This basic flip-flop element has fairly strong current-sinking capability (as an output) and fairly weak current-sourcing capability,

differing by a factor of β. 5 volts over 100kΩ gives us 44 μA of pullup current, which the presumed β=100 of the transistors amplifies to 4.4 mA. These can be equalized somewhat with emitter resistors, and flip-flops of various strengths can be made; in the case where driving a strong one from a weak one is desired, an inverting buffer of the same nature (NPN[C=1k-Vcc], or even a Darlington) can be interposed.

The LGP-30 approach mentioned in the earlier note, in which flip-flop Set and Reset inputs are driven by separate signals, is likewise applicable.

One interesting clocked-logic approach from the 1960s is Autonetics's four-phase logic, recently explained by Ken Shirriff in his explorations of the 1969 Sharp EL-8 pocket calculator, which was pretty interesting. In PMOS a clocked four-phase-logic inverter was three transistors, one of which was just used as a diode, and the clock signals were the only power connections to the gates.

Here's a working-in-simulation bipolar-logic version of the four-phase inverter Shirriff explains, along with a four-phase clock generator:

```
$ 1 0.000005 2.5790339917193066 65 5 43
R -112 144 -144 144 0 4 40 2.5 2.5 0 0.5
a -32 400 64 400 9 5 0 1000000 0 1.7540000000326756 100000
w -112 144 -32 144 0
w -112 144 -112 224 0
w -112 224 -112 304 0
w -112 304 -112 384 0
w -112 384 -32 384 0
w -112 304 -32 304 0
w -112 224 -32 224 0
R -64 112 -64 80 0 0 40 5 0 0 0.5
r -64 112 -64 176 0 100000
r -64 176 -64 256 0 100000
r -64 256 -64 336 0 100000
r -64 336 -64 416 0 100000
w -64 416 -32 416 0
g -64 416 -64 432 0 0
w -64 336 -32 336 0
w -64 256 -32 256 0
w -64 176 -32 176 0
a -32 320 64 320 9 5 0 1000000 1.2500000000148503 1.7540000000326756 100000
a -32 240 64 240 9 5 0 1000000 2.500000000006107 1.7540000000326756 100000
a -32 160 64 160 9 5 0 1000000 3.7500000000030536 1.7540000000326756 100000
I 64 320 160 320 0 0.5 5
150 160 336 272 336 0 2 0 5
w 64 400 160 400 0
w 160 400 160 352 0
I 64 240 160 240 0 0.5 5
150 160 256 272 256 0 2 5 5
w 160 272 160 352 0
w 64 240 64 192 0
150 160 176 272 176 0 2 0 5
w 64 192 64 112 0
w 64 112 272 112 0
```

```
I 64 160 160 160 0 0.5 5
w 64 192 160 192 0
207 272 336 272 368 4 φ1
207 272 256 272 288 4 φ2
207 272 176 272 208 4 φ3
207 272 112 272 144 4 φ4
d 368 112 464 112 2 default
w 464 112 528 112 0
c 528 112 528 176 0 1e-8 7.368162603581507 0.001
g 528 176 528 208 0 0
t 432 272 464 272 0 1 -7.368162450073534 -0.02239112192644880З 100
R 368 208 368 176 0 0 40 5 0 0 0.5
s 368 208 368 272 0 1 false
r 368 272 368 368 0 100000
g 368 368 368 384 0 0
207 368 112 368 144 4 φ1
w 464 288 464 336 0
t 448 352 464 352 0 1 0.5146280159592591 0.5325562677671504 100
207 400 352 400 384 4 φ2
207 464 416 464 448 4 φ1
r 464 368 464 416 0 100
207 528 112 576 112 4 /Q
r 368 272 432 272 0 1000000
r 400 352 448 352 0 100000
d 464 112 464 256 2 default
207 368 272 336 272 4 Q\sinput
o 35 64 0 4098 5 0.05 0 2 35 3
o 36 64 0 4098 10 0.05 0 2 36 3
o 37 64 0 4098 5 0.00009765625 0 2 37 3
o 38 64 0 4098 5 0.00009765625 0 2 38 3
o 54 64 0 4099 10 0.00009765625 1 2 54 3
```


This inverter is powered from phases φ1 and φ2; the φ1 pulse charges its output capacitor, and then φ2 discharges it if the input is high. The circuit is φ1->-/Q-(10nF-GND||>-x), Q-1M-NPN[Q1, C=x]-y, φ2-100k-NPN[Q2, C=y]-100-GND. So φ1 charges up the /Q output through a diode when φ1 is high; a capacitor to ground then holds the output high when φ1 goes low, unless point x pulls the capacitor down through a second diode (unnecessary in the PMOS version, but necessary to prevent the base-collector junction on the input transistor from going into forward conduction.) Point x is pulled to ground via the input NPN transistor Q1, which sees the input through a 1MΩ base resistor, and whose emitter goes to the collector of Q2. Q2's base is connected to φ2 via a 100kΩ resistor, and its emitter is connected to φ1 via a 100Ω resistor, so Q2 only pulls Q1's emitter to ground when φ2 is high and φ1 is low. Then the circuit's output is valid during φ3 and φ4. Whew!

As it happens φ2 (the "sample phase") is never low when φ1 (the "precharge phase") is high, so the diode nature of Q2's base-emitter junction is not relevant here; a relay winding between the two phases would also work. Similarly Q1 would work just as well being a relay, but of course wouldn't be able to run at 60kHz like in the original calculator.

Series-parallel combinations of input transistors can provide arbitrary monotonic logic functions before the inversion, at a cost of one extra transistor (and perhaps base resistor) per input, and of course you can use diodes too.

φ3 and φ4 work in exactly the same way as φ1/φ2, and you can additionally use the same design with φ1/φ3 and φ3/φ1, although in that case it *does* matter that Q2 stays off when the sample phase goes low, reverse-biasing its base-emitter junction. Otherwise the circuit is exactly the same.

Topics

- Electronics (p. 1145) (39 notes)
- Falstad's circuit simulator (p. 1198) (7 notes)
- Physical computation (p. 1208) (6 notes)
- Four-phase logic

Can you do direct digital synthesis (DDS) at over a gigahertz?

Kragen Javier Sitaker, 02021-02-08 (updated 02021-02-24)
(30 minutes)

I watched a GreatScott! video recently in which he designed and built a direct-digital synthesis waveform generator going up to a few MHz, using a waveform-generator chip which mostly consists of a 28-bit counter driving a sin() ROM attached to a DAC through a mux. When you want a sawtooth wave instead, the mux selects the counter instead of the ROM output, and when you want a square wave, it just selects the MSB.

(I haven't tried any of what is described below, even in simulation, so it wouldn't be unsurprising if there are fatal flaws in my calculations.)

GreatScott's designs

In the video, he compares his €600 Siglent SDG 2082 X, which goes up to 80 MHz and generates 1.2 gigasamples per second; his €70 Ascel Æ20125, which goes up to 10 MHz but only up to ±5 V; the above-mentioned cascade of three LM318N circuits, which only operates over about 1.7 kHz to 40 kHz with the passives he chose, and of course has a nasty temperature coefficient; a €6 kit built around the analog XR-2206 monolithic function generator, which goes up to 1 MHz; and his own €50 design built around the AD9833 DDS function generator IC (which IC goes for US$10.04 on Digi-Key in quantity 1), which goes from DC to a bit past 12 MHz.

He points out the AD9833 gives better results than a popular pure-analog three-opamp circuit which configures the first opamp as a relaxation oscillator and the other two as integrators, in large part because the relaxation oscillator output has shitty RC-decay edges.

The LM318N is pretty fast; TI's LM318N datasheet claims 15MHz "small-signal bandwidth" (typical, not minimum) and 50V/µs slew rate; their plot of unity-gain bandwidth suggests 15MHz at ±5V and 25° increasing to 19MHz at ±20V. Digi-Key lists them for US$1.13 in quantity 10. Its open-loop gain is claimed to drop off from 110dB below 100 Hz at the usual 20dB per decade, so at GreatScott's desired 10MHz it only has about 5dB left. The circuit in question is maybe not very demanding of the op-amp's open-loop gain, since each opamp is just amplifying its own output or the output of the previous stage. The slew rate should also be okay. It should be fine for a sine wave — I think 10MHz is a radian per 16 µs, so at Scott's desired ±12V, the maximum slew rate of a sine wave is 24V/16µs, or 1.5V/µs — and even a 10MHz square wave shouldn't be too trapezoidal at 0.5 µs of rise or fall time followed by 50 µs of high or low time. I conclude the opamp is fine and the circuit design is at fault. Probably a Schmitt trigger to clean up the square-wave transitions and careful control of parasitics would yield totally acceptable results.

Microcontroller-based DDS

At lower frequencies, you might as well just use a microcontroller. A 108MHz GD32 can, in theory, happily spit out 54 megasamples per second of digital data on one of its 16-bit I/O ports, and if you feed that to a simple R-2R DAC feeding an amplifier, you can easily get 6 bits of precision, or 8 bits with careful trimming. And, on the similar STM32F103C8 from ST, StackOverflow user SirSpunk was able to achieve one output word per two clock cycles, which would give the above 54 megasamples per second, though this required some trickiness like keeping the samples in CPU registers. (Chips like the STM32F103 and the GD32F103 also include a 12-bit DAC with supposedly about 10 bits of precision, but the DAC cannot run nearly this fast.) A dedicated DAC chip could improve precision, but improving the sample rate would require using a faster microcontroller. Moreover, even this data rate may not be achievable if the data samples need to come from somewhere else, like an arithmetic operation or fetching from RAM. ST's appnote 4666 details achieving sustained data rates of 8 to 10 megasamples per second using DMA on some other STM32-family microcontrollers, but I don't think the STM32F103 or GD32F103 supports DMA for GPIO.

Up to 1 MHz, 54 megasamples per second is a buttload of samples, technically speaking. The tenth harmonic would be 10 MHz, and its Nyquist frequency 20 MHz, so you should be able to get nice sharp edges on your 1 MHz square and sawtooth waveforms. At 10 MHz, they'll start to look pretty darn fuzzy, though: you only have 5.4 samples per cycle, so you're going to have slow transitions, a lot of ringing, or most likely both, depending on how you set up the analog output filtering.

Adjusting the clock speed is a potential approach to avoiding computation in the inner loop of slamming the samples out; for example, if you can store 8 samples in 8 CPU registers, you can produce those 8 samples in a tight loop, getting a 6.75MHz arbitrary waveform at 54Msps; and by producing them once forward and once in reverse, you can get a 3.375MHz arbitrary symmetrical waveform. But producing an arbitrary *6MHz* waveform would be much easier to do if you can lower the CPU clock to 96MHz.

The great advantage of using a microcontroller is that you can potentially output a very flexible set of waveforms: not just square, sawtooth, triangle, and sine, but also for example AM, FM, QPSK, white noise, and filtered weighted sums of any of the previous ones. But what good is that if your waveform comes out shoddy?

For square waves in particular you may be able to use a separate analog data path with different filtering (sharpening edges with a Schmitt trigger and relying on clamping to the power-supply rails, say), but that doesn't help with other waveforms containing sharp edges.

Non-microcontroller logic

Parallel SRAM

Well, what if you hook up a DAC (R-2R or IC) to the output of

a RAM chip? Digi-Key sells the obsolete CY62256NLL-70ZRXIT for US$0.58, which is a 28-pin 70ns SRAM chip with 8-bit-wide output and 15 address lines; if you gang up two of these mothers you get 16-bit-wide output. As long as they're in read mode, every time you change the value on the address bus, you get your data out 70ns later (or 55ns later in some other grades of the chip, according to Cypress's datasheet). They used to even sell them in DIP and SOIC forms. Too bad it hit end-of-life in 02017. I don't know how glitchy the output is, so you might need external tristate buffering, and also it uses TTL thresholds, so you may need level shifters. (Also, since you need to load the data on the same data bus you're using for the DAC, you might want external tristate buffering to disable the DAC while you're loading that data.)

70ns isn't fast enough, though; if you didn't need any extra time to switch addresses you would only get 14.3Msps that way, and so a maximum sine-wave speed of 7.1MHz, and a maximum square-wave speed somewhere in the neighborhood of 1 MHz.

A much more modern, but still obsolete, part is the 80¢ Cypress CY7C1021BN-12ZXC, which has 16 address lines, 16 data lines, 44 pins, and a 12-ns access time. The CY7C1021BN datasheet, which is for the 15-ns version, claims that it's basically otherwise identical to the older chip, except that it has separate byte-high-enable and byte-low-enable inputs so you can use it with an 8-bit bus if you want.

So this is starting to sound decent; you should be able to get 60 megasamples per second out of such a chip once you've loaded the waveform into it. And you can load up to 65536 samples into it, or 131072 if you just use 8 bits of data and use the /BHE and /BLE lines as an additional address bit; or you could tie some address lines to ground in order to save I/O pins.

A CPLD or something could be configured as a counter to generate the addresses at a higher clock speed than the microcontroller can manage. In the case where the carry chain is becoming too slow, you can use an LFSR instead of a normal binary counter, with the XOR gates interposed between successive register bits, thus getting your critical path delay down to a single XOR's propagation delay.

Non-obsolete parallel memory parts that could be used similarly include the following:

Part number	Price, qty 1	Access time	Maximum clock speed	Address lines	Data lines	Package	Voltage
CY62136EV30LL-45ZSXIT	US$1.11	45ns	(async)	17	16	44TSOP II	2.2–3.6V
CY7C1020D-10VXI	US$3.94	10ns	(async)	15	16	44-BSOJ	4.5–5.5V
CY7C1329H-133AXC	US$2.30	(N/A)	133MHz	16	32	100-TQFP	3.15–3.6V
CY7C1360C-166BZC	US$4.56	(N/A)	166MHz	18	36	165-FBGA	3.135–3.6V
CY7C1360C-200AJXC	US$4.18	(N/A)	200MHz	16	36	100-TQFP	3.135–3.6V

The other 5-volt part also shares the annoying TTL thresholds.

The CY7C1329H and CY7C1360C "include[] a two-bit internal counter for burst operation" when you hold the /ADV line low, which suggests that you could feed it an externally-generated address every *four* samples, which means you could even use a cheap microcontroller to do the address generation, since you need no more than 50 million addresses per second. Amusingly, it even has an "interleaved" mode (selected by tying the MODE pin to V_{DD}) in which it can count either 0123, 3210, 1032, or 2301 rather than the usual 0123, 1230, 2301, and 3012 alternatives; this would be useful for time-reversing a part of a waveform.

These parts, being synchronous, of course produce output data starting in the following clock cycle rather than as soon as possible.

For such synchronous memories, you might need some external glue logic to gate off the memory control lines faster than the microcontroller can do it on its own.

You'd think that someone would have sub-nanosecond SRAM by now, and Cypress used to make a line of what purported to be sub-nanosecond SRAM, but no longer, and anyway it was synchronous at speeds of 200MHz or less. If you want "subnanosecond" RAM today you have to go with DRAM, and none of it can support random write accesses in less than 15ns. Digi-Key has 42 Winbond W971GG6SB-18 chips in stock for US$4 each; these are a gibibit organized 16 bits wide with a 533MHz DDR2-1066 clock. Its CAS latency is 6 clocks (11.3 ns), its write recovery time is 15 ns, and if I understand correctly, there are various other latencies related to random accesses that bring its random access latency up near 100 ns. But if you just want to spew out a sequential stream of data, it can totally give you 16 fresh bits every 940 picoseconds for a good long while.

The async parts would still need some kind of external counter logic to drive their address lines faster than the microcontroller can manage. You could do this with a two- to four-bit counter on the low-order address lines or by wiring up the whole address bus to the thing.

CPLDs and PLDs as counters

One approach here would be to use a generic programmable-logic chip like the US$0.94 Altera MAX V 5M40ZE64C5N 40-(one-bit)-logic-element 7.5-ns CPLD, which can run its output buffers at up to 3.6 volts and run a 16-bit counter at up to 118.3 MHz; the US$1.44 Lattice "ispMACH" 4000ZE-series LC4032ZE-7TN48C 32-macrocell 7.5-ns CPLD, which can also do 3.3-volt output and I think is similar in speed or perhaps could manage up to 260MHz; the US$1.80 Atmel ATF1502ASV-15AU44 32-macrocell 15-ns CPLD, which runs at 3.3 volts natively and I think can reach 77 MHz; or maybe an old-fashioned PLD like the US$2.06 22V10, which comes in a 10-ns grade and astoundingly even a US$2.14 5-ns grade these days, and of course run on 5 volts and have annoying TTL thresholds, but in theory can run at up to 166MHz. Since even a two-bit counter would be enough to lower the burden on the microcontroller by a factor of four, we could even consider smaller PLDs like a 16R4 or 16V4, but they are all obsolete.

Amusingly, the Altera chip also contains an 8192-bit block of Flash

with auto-increment addressing, so if your waveform data is smaller than that and never changes, you don't need an external RAM chip!

And *obviously* with an FPGA like a Lattice UltraPlus ICE40UP5K (US$6, 5280 4-LUTs, 120 kibibits of block RAM, 1 mebibyte of SPRAM, eight 16-bit multipliers, capable of running a 16-bit counter at 100 MHz, fully supported by IceStorm) or a Lattice LFE5UM5G-45F-8BG381C (US$31, 85k 4-LUTs, 72 18-bit multipliers, four 5Gbps SERDES channels, running some functions at up to 400 MHz) you can do all the DDS you want entirely inside the chip, as long as it's not above a couple hundred megahertz.

Dedicated counter ICs

There are various popular counters like the 72¢ 4-bit 90MHz 74AC161, the 54¢ 14-bit 65MHz 74HC4060, the 38¢ dual-4-bit 107MHz 74HC393, and the 50¢ 4-bit 167MHz 74HC161; all of these can use a wide variety of supply voltages. Ripple counters like the [42¢ 12-bit 210MHz 74VHC4040][22] would not work for this.

But what about memories with built-in counters?

Serial memories

Microcontrollers are commonly used with memories with bit-serial interfaces, either SPI, I²C, dual SPI, or quad-SPI. Typically you can write these memories in true bit-serial mode and read them in quasi-parallel mode, but even when not, it would be practical to gang up several of them (4, 5, 8, or 10 chips, say), write to several of them one at a time, then read from them all at once; typically they permit reading an entire sector with a single command. Almost invariably these are Flash — serial SPI SRAM like the 23K256 exists but only up to 45MHz. Some of these SPI Flash memories are too slow to be useful for this kind of application, but others are plenty fast. Specimens of this genre include the US$2.07 166MHz quad-SPI Winbond W25N512GVEIG, the 30¢ 100MHz dual-SPI GigaDevice GD25D05CTIGR, and the 31¢ 85MHz quad-SPI Adesto AT25SF041B-SSHB-B.

Taking the GigaDevice device as typical, we find in its datasheet that it not only permits bit-serial writing, it requires it. Puzzlingly it claims to permit reads at 160Mbps but a 100MHz clock (see below). It holds half a mebibit of data, divided into 16-page sectors of 256-byte pages. The data-rate doubling is implemented by making its serial input pin bidirectional.

A single READ command (0x03, or 0x0B for fast read, followed in either case by three address bytes) will eventually read the whole memory if CS# is held low for enough clock cycles. There are no dummy bits, start bits, stop bits, or command bits in the timing diagram once the data stream starts.

The "AC characteristics" table clarifies the clock speed mystery: "fast read" 0x0b can happen at 100MHz, but regular read (0x03, which appears to be otherwise identical) and dual-output read (0x3b) can only happen at 80MHz.

Because it's Flash, programming a new waveform into the chip is slow (700 µs per page, about 300 times slower than reading), and can only be done some 100k times before risking burning the chip out.

This seems like an acceptable tradeoff.

Normal SPI uses three pins on your master chip plus one per slave: SCK ("SCLK"), MOSI ("SI"), MISO ("SO"), and one /CS ("CS#") per slave. (This chip additionally has a "WP#" pin which must be high for writes, which I suppose is intended to prevent accidental erasure due to EMI or software bugs, but it would probably be acceptable to tie it low.) But if we're going to use various memory chips' SO pins to directly drive a DAC, they can't all be tied to each other as they normally would be.

Ganging up 8 GD25D05Cs in this way could be achieved most simply by just *not routing the MISO pins back to the master* — that is, configuring the memory chip as write-only memory, with their output pins connected only to the DAC. This would prevent the program from reading the status register, or reading back waveforms to verify them, but that's not necessary for the waveform generator to work. Then all that remains is to drive the SCK inputs of the slaves from a free-running 100MHz clock on command from the slower master, so you need a 100-MHz 2-mux on the clock line; SCK, MOSI, and mux-select pins on the master for all slaves; and eight /CS pins on the master, one per slave. When issuing a "fast read" command, the master would broadcast it to all slaves at once.

And that way you get 100Msps of waveform-generation output.

The Adesto chip seems to be very similar, down to using the same opcode bytes and the same pins for dual-output reads, but also supports four-bit-per-clock output by co-opting the /WP pin and a /HOLD pin the GD25D05C lacks; also its multi-bit reads run at full speed, and it has the option of clocking in the address bits on multiple pins as well, and clocking in the data bits on multiple pins when writing the chip.

Augmenting the circuit to support reading from the memory chips without using more pins on the microcontroller is relatively simple: a pullup resistor per memory chip, plus an 8-input AND, NAND, or parity chip; or alternatively pulldowns and an OR, NOR, or parity. Another way to implement this is with diode logic: one diode down from the shared microcontroller pin to each memory chip's MISO pin, and a pullup resistor on the master side, which can be internal on most popular microcontrollers. Or you can just use a separate microcontroller pin for each MISO line, bringing the total to 19 GPIOs for 8 memory chips.

Augmenting these circuits to support the use of multi-bit outputs is potentially more difficult if you don't have all those GPIOs: the MOSI line becomes bidirectional, and you want the master to be able to send bits to any of the slaves, but you don't want the slaves' drivers to be able to fight each other. This is similar to the problem of a bidirectional level shifter, which is in fact a thing you might want in this case anyway.

If not, though, one approach is a pullup on each slave MOSI pin, a diode from each slave MOSI pin to the shared master MOSI pin, and a pull*down* on the master MOSI pin. When all the involved pins are tristated, a weak current will flow through the diodes, maintaining all the relevant pins in an indeterminate state which probably wastes a lot of power. If the master pulls its pin low and the slaves are tristated,

this will bring all the slaves' pins to a diode drop above ground, which hopefully is low enough to count as "low"; if it pulls its pins high, this will overwhelm its pulldown and allow the slaves' pullups to pull their inputs high. If the master tristates its pin and some slave pulls its pin high (because MOSI has become part of a multi-bit bus that the slave is writing to), the master's pin will rise to a diode drop below V_{CC}, which is safely HIGH at most voltages; if the slave pulls its pin low, overwhelming its pullup, then the master's pullup will pull its pin all the way to ground. And in no case can two slaves' outputs fight each other.

(Incidentally, this kind of thing would also be useful for spewing out canned bitstreams at higher rates than your microcontroller can manage, too: generate the bitstream at your leisure in the serial memory, then spew out bits at high speed, possibly repeatedly and into a SERDES.)

Toward a gigahertz

Unfortunately, none of the above approaches get us close to being able to synthesize gigahertz signals. In fact, most of them top out (with easily available hardware, anyway) around 100Msps, where the top sine frequency you could manage would be around 50 MHz, and the top frequency with reasonably sharp edges would be in the neighborhood of 5 or 10 MHz. So I guess that's why "GreatScott" picked that US$10 Analog Devices chip; you *can* do better, but it's not easy.

To manage hundreds of megahertz with an arbitrary waveform, let alone a gigahertz, we'd need a different approach. I think it's feasible without reaching for exotica like indium phosphide, though. AD and TI both have analog-switch ICs reaching from DC up to 1 GHz or more; ADG902-EP (4.5GHz, US$3.39 from Digi-Key, 17-ns on+off switching time), ADG919 (4GHz, US$3.34, 19.5-ns on+off switching time) TMUX1072 (1.2GHz, US$1.18, 260000-ns on+off switching time), and TMUX136 (6GHz, US$0.98, 600-ns on+off switching time) are representative examples. These are MOSFET switches; the more common PIN-diode type typically takes over a microsecond.

MOSFETs have an intermediate "ohmic mode" of conduction, in between "saturated" fully on and "subthreshold" fully off; as can be seen from the above figures, they have much higher bandwidth through the channel than they do for turning the gate on and off. By precisely controlling the gate voltage, you can control the impedance a signal sees going through the channel, and thus its attenuation. This phenomenon is not extremely linear in the gate voltage (especially if you don't subtract the threshold voltage, but even then), and the current isn't even all *that* linear in the drain-to-source voltage V_{DS}. But it's a reasonably good approximation when V_{DS} isn't too high and V_{GS} isn't too low. And, as we will see, in this application we can correct for the nonlinearity in gate voltage in software.

With two such pseudo-variable-resistors, you can make a voltage divider from an output signal terminal, through a MOSFET channel, to an input signal terminal, through a second MOSFET channel to ground, such that the total input impedance seen by the input signal terminal is some constant impedance such as 500Ω. If a short pulse

arrives on the input terminal, some attenuated version of the pulse will be seen at the output terminal. If the upper MOSFET is nearly saturated at 10Ω, the lower MOSFET to ground ought to be nearly off, at about 24500Ω, for the input to see 500Ω. If the upper MOSFET is at 250Ω, then the lower MOSFET should be at 500Ω, and the signal will be attenuated by half (6 dB). If the upper MOSFET is at 400Ω, then the lower MOSFET should be at 125Ω, and the signal will be attenuated by ⅘ (14 dB).

The key point here is that this configuration allows you to selectively attenuate a pulse train. Moreover, a similar voltage-divider arrangement allows you to selectively steer them with reasonably low insertion loss! If the input signal is to be divided evenly between two 500Ω outputs without losing impedance matching, then a MOSFET to each of them operating in the ohmic region with a 250Ω resistance will do the job, wasting only four ninths of the signal energy (3.6 dB), and this is the worst case; by reconfiguring the gate voltages to pass more of the signal to one side, this loss is reduced.

For example, if the MOSFET channel resistance to the left output is 50Ω, then the MOSFET channel resistance to the right input should be 5kΩ. The voltage on the left channel will be 500/550 of the input voltage, or 91%, and the power 83% of the original, a loss of about 0.8 dB. The voltage on the right channel is 500/5500 of the original, or 9.1%, and the power 0.83% (21 dB attenuated).

(The pulse passing through to the MOSFET source will, if positive, reduce V_{GS} temporarily, creating distortion; adding capacitance across those terminals should push that problem out to a long enough timescale that this nonlinearity doesn't provoke harmonic distortion and reflections.)

Multi-way splits with impedance matching are higher-loss than two-way splits; for example, if we split the signal into four equal parts with 500Ω each, we need 1500Ω of series resistance on each branch, thus losing ¾ of the voltage and burning 15/16 of the power in the resistors, a 12 dB loss. (XXX is that right? That can't be right.)

Such attenuated pulse trains can be passed over microstrip (or stripline if necessary) and summed with a resistor network (at the cost of further attenuation), following variable delays imposed by variable lengths of microstrip. Configuring the set of attenuations for each delay, by way of setting the gate voltage on the various MOSFETs, amounts to configuring a convolution kernel in the time domain, which is to say, a single iteration of a waveform; a pulse train at the desired fundamental frequency is then all that is needed to synthesize the desired waveform. If the delays are regularly spaced, thus forming a regular sampling, a spurline filter can notch out the sampling frequency and its harmonics.

So then the problem of gigahertz DDS reduces to the problem of setting the gate voltages on all these MOSFETs and producing a pulse train at the desired fundamental frequency.

How much microstrip are we talking about? Crudely speaking, about 150 mm per nanosecond, so perhaps on the order of a meter for signals with a fundamental frequency down to a few hundred

megahertz.

How should we distribute the delays to the customizable attenuators? If we distribute them evenly over the maximum possible interval — for example, 20 attenuators distributed every 250 ps out to 5 ns — then we will have effectively many fewer data points at even fractions of that interval. That is, if we emit a pulse every 190 ps, we're pretty okay — the first attenuator provides a pulse image at 60 ps from the beginning of a pulse interval, the second at 120 ps, then 180 ps, 50, 110, 170, 40, 100, 160, 30, 90, 150, 20, 80, 140, 10, 70, 130, 0, and finally 60 again, so we have a nice 10-ps effective sampling interval, just scrambled. But if we increase the pulse interval to 200 ps, we suddenly have only four samples per cycle: 50, 100, 150, and 0.

Distributing them at random is of course one possibility, which would be about as good or bad at all frequencies.

If we have room to make our microstrip 20 ns long, which is only about a meter and a half serpentined onto a PCB, we might have time to reconfigure the transistors between one pulse and the next, perhaps using one of the parallel-memory approaches described above, so at this magic point we have no lower frequency limit. AD's existing chips claim to achieve turning their pass transistors fully on and off within 20 ns, so this is apparently physically feasible.

Topics

- Electronics (p. 1145) (39 notes)
- Pricing (p. 1147) (35 notes)
- Microcontrollers (p. 1211) (6 notes)
- Communication (p. 1264) (4 notes)
- Radio (p. 1278) (3 notes)

ASCII art, but in Unicode, with Braille and other alternatives

Kragen Javier Sitaker, 02021-02-10 (updated 02021-02-24)
(9 minutes)

There are a variety of Unicode graphics drawing character sets, similar to the Large Character Microvector Set ROM from the HP 2640 series, which let you draw each character from a “3×3 matrix of smaller characters”.

I was thinking it might be fun to abuse some of these character sets for asciinema plotting and the like.

Unicode math character pieces

These are used to build up large math symbols over several character cells. I don’t know where they come from but I suspect they’re in Unicode for round-tripping compatibility.

(loop for i from #x239b to #x23b3 do (insert i))

```
⎛⎜⎝⎞⎟⎠⎡⎢⎣⎤⎥⎦⎧⎨⎩⎪⎫⎬⎭⎮⎯
```

You can use them like this:

```
⎛    ⎞ ⎡  ⎤ ⎧     ⎫          ⌠ ⎞
⎜⎛  ⎞⎟ ⎢  ⎥ ⎪     ⎪  ⎧        ⎠ ⌡
⎜⎜()⎟⎟ ⎣  ⎦ ⎨     ⎬  ⎪ x² dx
⎜⎝  ⎠⎟   ⎪     ⎪  ⎭       ⎲
⎝    ⎠   ⎣     ⎦        ⎳
```

Box drawings

This is the set from U+2500 to U+257F from Videotex mosaic characters:

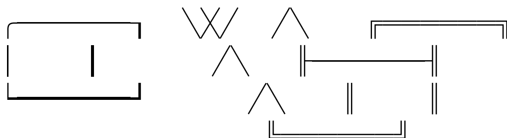
(loop for i from #x2500 to #x257f do (insert i))

```
─━│┃┄┅┆┇┈┉┊┋┌┍┎┏┐┑┒┓└┕┖┗┘┙┚┛├┝┞┟
┠┡┢┣┤┥┦┧┨┩┪┫┬┭┮┯┰┱┲┳┴┵┶┷┸┹┺┻┼┽┾┿
╀╁╂╃╄╅╆╇╈╉╊╋╌╍╎╏═║╒╓╔╕╖╗╘╙╚╛╜╝╞╟
╠╡╢╣╤╥╦╧╨╩╪╫╬╭╮╯╰╱╲╳╴╵╶╷╸╹╺╻╼╽╾╿
```

This contains light, heavy, and double line weights in horizontal and vertical orientations, though not all combinations of these, plus double, triple, and quadruple-dashed variants of light and heavy straight lines, plus light diagonals, plus rounded corners. With these you can do arbitrary combinations of vertical and horizontal lines in half-character-cell increments. I think all the combinations of light and heavy lines are provided; that would be 3^4 = 81 characters, which are the ones from U+2500 to U+254B inclusive except U+2504 to U+250B inclusive, plus the ones from U+2574 to U+257F: (+ (- (- #x254C #x2500) (- #x250c #x2504)) (- #x2580 #x2574)) gives 80,

so either I miscounted or one is missing. All three combinations of diagonal lines are provided, but most of the light–double combinations and all of the heavy–double combinations are missing.

So with these Videotex characters you can do things like this:



Quadrant characters

There's a 2018 "Graphics for Legacy Computing" proposal to add 64 sextant characters to Unicode starting at U+1FB00 compatible with the TRS-80 "pseudopixel" or "semigraphics" set, or teletext systems including Minitel. But since 1991 Unicode has contained "quadrant" characters, like the Sinclair ZX-80 and ZX-81 or the Commodore line, with *four* pseudopixels per character cell, from U+2596 to U+259F:

(loop for i from #x2596 to #x259f do (insert i))

This is visibly missing the all-empty and all-full configurations (■, which follows them, is not the all-full configuration), but normally a space can be used with or without inverse video.

For monochrome or 2-color graphics, these characters plus inverse video permit doubling the character grid resolution, with full color freedom.

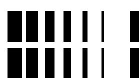
Eighth blocks and scan lines

This is the set from U+2580 to U+2590, in "Blocks" (now "Block Elements"):

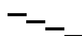
(loop for i from #x2580 to #x2590 do (insert i))

These characters allow you to divide character cells vertically or horizontally (but not both) into two colors with a resolution of ⅛ cell. They're commonly used, for example, for plotting sparklines. They are clearly designed for use with inverse video (^[[7m in ANSI).

This is particularly useful for bar plots, as provided by UnicodePlots.jl, where the blocks divided left to right can provide 640 pixels of horizontal precision for your bars on an 80-character screen. This includes stacked bars in which different colors divide a bar horizontally and may include inline legends. However, experimentation seems to show that my terminal here renders them incorrectly, with some unfilled space above them in the character cell:



Similar are the "horizontal scan line" characters, of which there are only four starting at U+23BA:

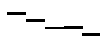


These are explained in Frank da Cruz's proposal L2/00-159 as being for round-trip compatibility with some old terminals:

```
E0D6   Scan 1    DSG 06/15, H19 07/10, WG3 05/00, TVI 09/00, IBM SV300400
E0D7   Scan 3    DSG 07/00, WYA 01/01, WG3 05/00, IBM SV300200
E0D8   Scan 5    DSG 07/01, WYA 02/02, IBM SV300300, IBM SM920000
E0D9   Scan 7    DSG 07/02, WYA 01/03, WG3 05/01, IBM SV300100
E0DA   Scan 9    DSG 07/03, H19 07/11, WG3 05/01, TVI 09/01, IBM SV300600
```

They're intended to join up with U+23B8 "⎸ " and U+23B9 " ⎹" to make boxes in a similar way to the Videotex box-drawing characters above, and there are supposed to be five of them, but this does not work in my current font:

These can be used for sparklines in a similar way to the vertically-divided eighth blocks, but with half the resolution. Sometimes U+2500 is considered a part of the set, but at least in the font I'm using at the moment, it doesn't fit:

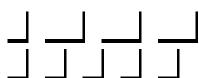


The proposed "legacy computing" characters would augment these with, among other things, 8-position horizontal and vertical lines.

Edge box drawing

As sort of noted above, the characters ⎹ U+23B9 and ⎺ U+23BA link up:

In my current font, successive rows of ⎹ don't quite link up the way they're supposed to, but successive columns of ⎺ do. From the "eighth blocks" area we have another couple of characters (or actually various pairs of characters) that can potentially be applied in the same way:



These also fail to link up from one line to the next in my current font.

The aforementioned "Graphics for Legacy Computing" proposal includes more such characters, including four corners intended to link

up with the above "eighth blocks":

(loop for i from #x1fb7c to #x1fb7f do (insert i))

In theory, with three printable characters like this plus a space, you could lay out a grid of thin lines with the resolution of the character grid, with the lines beginning and ending at, say, the upper left-hand corner of each character cell. This would be potentially more parsimonious than the box-drawing characters we *did* get, which can end and join at the center of each character cell, but be interrupted at half-character-cell intervals, a relatively useless ability. But for this ability we need 15 graphics characters (for a single line width) rather than 3.

Shade characters

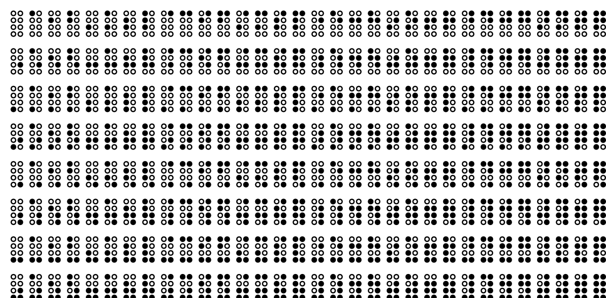
The three "shade" characters from U+2591 to U+2593 can be used to dither between a foreground color and a background color; really you only need two of them if you have inverse video or full liberty in color choice:

This doesn't increase the resolution of your display any, though.

Braille provides the best resolution, though not without drawbacks

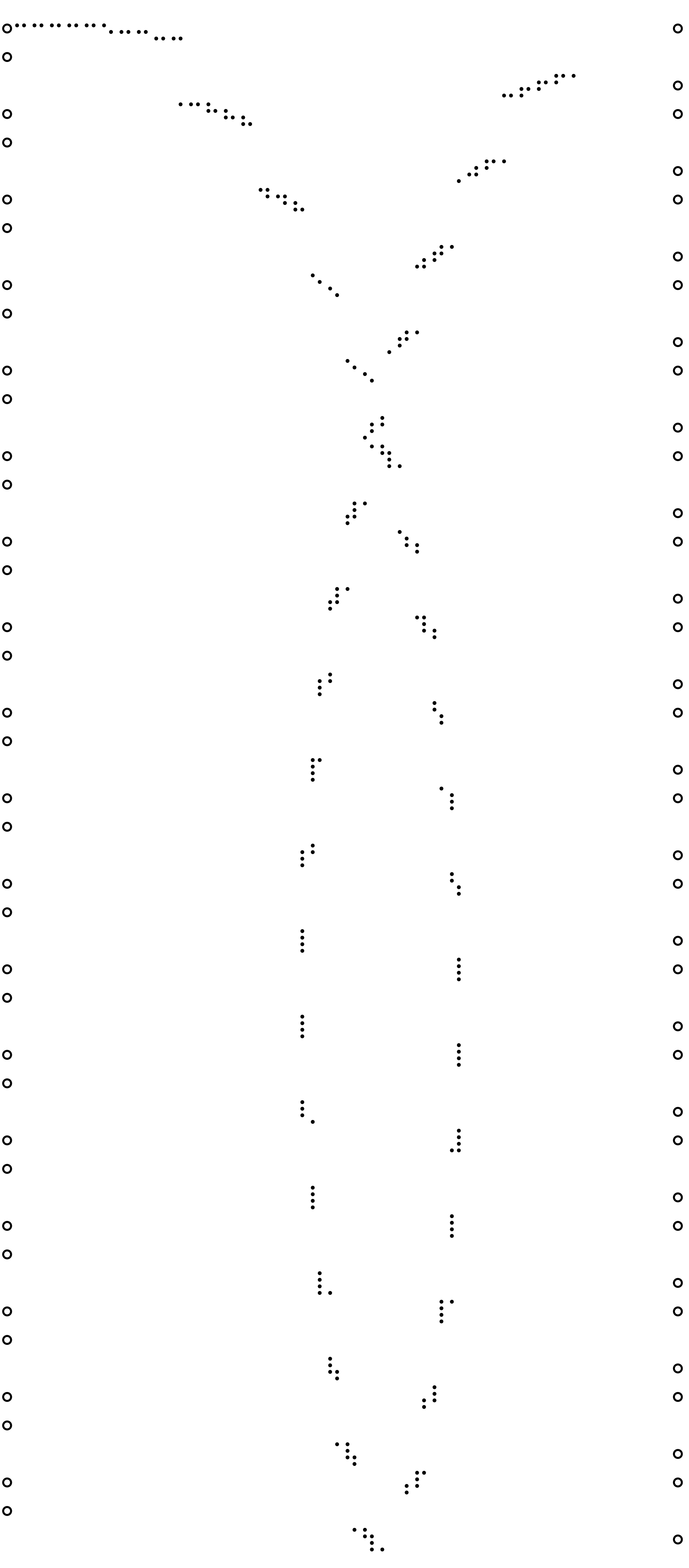
The Braille block from U+2800 to U+28FF offers a full selection of 256 binary patterns of 8 pixels:

(loop for i from #x2800 to #x28ff do (insert i))



At the expense of a little dottiness, background bleedthrough, and spacing jitter, this can be used to get 8× character cell resolution for things like plotting points and lines on a character display; 15360 pixels in a standard 80×24 terminal window. This is better resolution than even the proposed sextant characters, and the pseudopixels are usually squarer. The bit positions within the character cell, with x increasing right and y increasing down, are (0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (0, 3), and (1, 3), in that order.

For example, you can plot this circle:



(Some terminals display this suboptimally with the non-active Braille dots also drawn, as empty circles.)

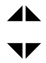
I did that with this simple Python program:

```
from __future__ import division, print_function
import sys

bitpos = [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (0, 3), (1, 3)]

try:
    unichr
except NameError:
    unichr = chr

class Canvas:
    def __init__(self, text_cols, text_rows):
        assert text_rows > 0
        self.pixels = [[0 for x in range(text_cols * 2)]
                       for y in range(text_rows * 4)]

    def pixelsize(self):
        return len(self.pixels[0]), len(self.pixels)

    def pset(self, x, y):
        try:
            self.pixels[y][x] = 1
        except IndexError:
            pass                # clip silently

    def render(self):
        p = self.pixels
        ww, hh = self.pixelsize()
        return '\n'.join(''.join(
                unichr(0x2800 + sum(1 << i if p[y+by][x+bx] else 0
                                    for i, (bx, by) in enumerate(bitpos)))
                for x in range(0, ww, 2)
                ) for y in range(0, hh, 4))
```

```
    def write(self, fileobj):
        fileobj.write(self.render() + '\n')

def draw_circle(canvas):
    ww, hh = canvas.pixelsize()
    mindim = min(ww, hh)
    cx, cy = ww/2, hh/2
    r = mindim/2 - 1
    x, y = r, 0
    for i in range(1000):
        canvas.pset(int(round(x + cx)), int(round(y + cy)))
        x -= y * .01
        y += x * .01

if __name__ == '__main__':
    import cgitb; cgitb.enable(format='text')
    canvas = Canvas(80, 24)
    draw_circle(canvas)
    canvas.write(sys.stdout)
```

Triangle characters

These probably are *not* useful as mosaic characters like the ones in the "Graphics for Legacy Computing" item above; faced with the choice between making them mate properly for mosaicing and giving them 45° angles, font designers have typically chosen the latter:



Topics

- Programming (p. 1141) (49 notes)
- Python (p. 1166) (12 notes)
- Graphics (p. 1177) (10 notes)
- Terminals (p. 1202) (6 notes)
- ASCII art (p. 1305) (3 notes)
- Art (p. 1306) (3 notes)
- Unicode (p. 1315) (2 notes)

Skew tilesets

Kragen Javier Sitaker, 02021-02-14 (updated 02021-02-24) (7 minutes)

Some ideas about tile-based media for constructing systems.

Historical background

After writing ASCII art, but in Unicode, with Braille and other alternatives (p. 128) I remembered a graphics program called SYMED on the Zenith Z-100, capable of drawing, for example, extensive circuit schematics, despite the machine only possessing 128KiB of RAM, a sub-MIPS processor, sub-megabyte floppies, and totally unaccelerated graphics. (I suspect that this program is totally unrelated to the Mentor Graphics program of the same name; although they are used for related purposes, there is no similarity in how the programs work.) In SYMED you defined a tileset, as with Nintendo games, and placed tiles from the tileset to form your drawing. SYMED arranged these into the machine's framebuffer for display (8 colors at 640×225, which works out to a hefty 54KB of video RAM) but stored the tile definitions and tile indices.

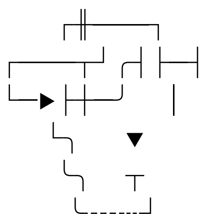
I forget how big the tiles were, but if we suppose they were 8×8 (a common size for such things) you might be able to get a creditable representation of a diode or capacitor, or half a resistor or inductor, into one — you could repeat the "symbol" for the other half of the resistor. Vertical and horizontal wires, four corners, four junctions, four diodes, two resistor sections, two inductor sections, two capacitors, a four-way intersection, a crossover, and a battery might add up to 23 symbols; you could probably do a real circuit with around 64 symbols. The definitions of those 64 symbols, if monochrome, would occupy 512 bytes, and each screenful of 80×28 tiles would occupy 2240 bytes at one byte per tile, so a screenful might be a 3-KB file, quite a lot of compression compared to the 54K of a raw framebuffer dump, and still smaller than a 6K monochrome dump. If the tiles were full color, 64 of them would be 1.5 KiB, so a full-color screen-sized drawing would be almost 4K. (You could do bit-packing tricks with such a small number of tiles, packing 8 tile indices every 6 bytes, but I don't know if SYMED did.)

In theory it would be easy to do PLATO-like or APL-like overstrike with such tiles, or Nintendo-like sprites composited in at arbitrary places, although I don't think SYMED could do this. I don't remember if it even supported text annotations.

For things like schematics you wouldn't need to separately draw and store the four orientations of diodes; you could generate them algorithmically with rotations. This would cut the 23 separate symbols above to 10 and additionally let you reorient the battery as you wished. Nowadays this is interesting not to save space on 400-kibibyte floppy disks but to make the system a more fluent medium for creation, because you don't have to make parallel modifications to various copies of your sprites.

Here's a somewhat similar set of symbols from the Unicode box

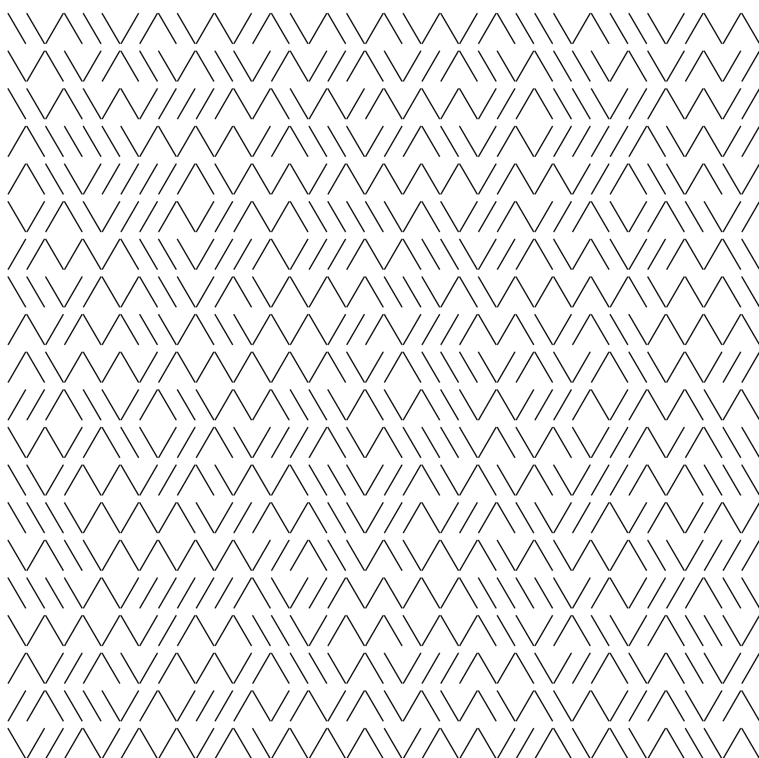
drawing set, although this does not represent a coherent schematic for anything and lacks the circles indicating wire joins:



Still, it would be nice to have a more parsimonious set of tiles. Perhaps this Commodore BASIC program from the *Commodore 64 User's Guide* is relevant:

```
10 PRINT CHR$(205.5+RND(1)); : GOTO 10
```

This outputs a "maze" like the following:



(I generated the above in Python 3:

```
for _ in range(20):
    print(''.join(chr(random.randrange(0x2571, 0x2573))
                  for _ in range(40)))
```

)

This uses a smaller tileset, only two tiles, to produce an interesting range of topologies, though the passages don't branch. On the Commodore, with its square character cells, the lines were perpendicular to each other, just not to the edges of the display. In addition to the diagonal-line characters 205 and 206 above, it also had a crossed-diagonal-lines character 214 (╳ in Unicode, U+2573, `BOX DRAWINGS LIGHT DIAGONAL CROSS`), and of course a space character.

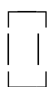
By adding these additional tiles, we can get branching passageways,

which is most interesting when the amount of branching is close to the critical percolation threshold:

```
for _ in range(20):
    print(''.join(' ' if random.random() < .2 else
                  '╳' if random.random() < .1 else
                  '╲' if random.random() < .5 else
                  '╱' for _ in range(80)))
```

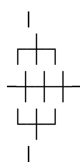
If you make the tiles square and rotate 45°, boxes like this:

become boxes like this:



Correspondingly this

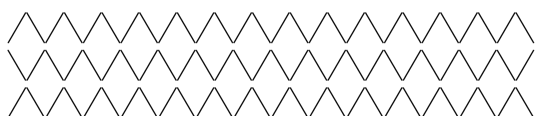
becomes this:

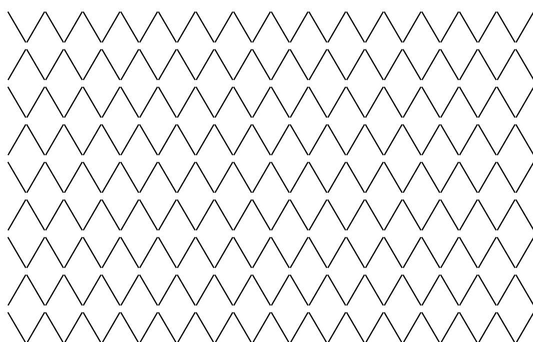


What you give up is the ability for the lines to begin or end on any grid point; now they can only begin and end on half of the grid points, the ones corresponding to the black squares on a chessboard. In exchange, for line drawings, you only need one bit per grid point instead of 4, and you only need to draw four tiles including space, or three if rotation is automatic, instead of 16 (or six if rotation is automatic: ─ ┐ ┤ ╷ ┼ plus space).

Paraxial parallelograms

Now, suppose we divide our board into square or parallellogram tiles whose diagonals are vertical and horizontal:





In each of these tiles we can place a horizontal line, a vertical line, both, or neither, or some other component, such as a resistor or diode. If they're square, you can automatically generate multiple rotated tiles from a single master, so you only need to draw, for example, one diode instead of four.

This skew-tile approach reduces the number of tiles, but with some drawbacks. Unless supplemented with sprites, if you *do* want to, for example, render three-way intersections differently (for example with a dot), you need not just one new tile but two or more, containing the different parts of the dot. In the simplest approach, this also requires the user to replace four tiles when they want to place such an intersection, but there are various possible approaches to avoiding this, including Wang-tile-like approaches. And inserting or deleting rows or columns of tiles is no longer so simple.

Isometric grids

A uniform grid of equilateral triangles can tile the plane, and the isometric projection measures along just such a grid. QBert and Zaxxon (from 01982) take place on such an isometric grid. Marble Madness (01984, by Mark Cerny, running on a 68010) used an isometric projection, but included lines that deviated from the grid for things like sloped surfaces.

One approach to creating a tileset for such a grid is to pick one of the three isometric planes and draw tiles for the parallelograms in that plane. This can display integer displacements perpendicular to that plane as long as the displacements are quantized.

Ugh, I guess I should implement these things to see how well they work.

Topics

- History (p. 1153) (24 notes)
- Python (p. 1166) (12 notes)
- Graphics (p. 1177) (10 notes)
- Composability (p. 1188) (9 notes)
- Tiled graphics (p. 1269) (3 notes)
- Illinois PLATO (p. 1280) (3 notes)
- BASIC (p. 1303) (3 notes)
- ASCII art (p. 1305) (3 notes)
- Overstrike (p. 1347) (2 notes)

Threechet

Kragen Javier Sitaker, 02021-02-16 (updated 02021-02-24)
(4 minutes)

Truchet tiles are squares divided into a black right isosceles triangle and a white one, and by tiling a surface with them at various orientations you can achieve a wide variety of interesting patterns, as observed by Truchet in 1722 (and earlier in briefer papers published in 1704 and 1707, it seems).

A variant of the Truchet tile introduced by Cyril Stanley Smith in 1987 has two quarter-circles connecting the centers of its square sides, and can occur in two orientations; Smith comments early on:

> Truchet's patterns are superficially similar to those used in the construction of the mosaic tiles so prominent in Islamic architecture, the construction and philosophy of which, based on the intersection of circles of differing radii, has been so well treated by Keith Critchlow, but the principles are more fundamental.

Then later, introducing his variant:

> Of course, there are many other tile shapes with interesting properties, for example the non-periodic tilings described by Martin Gardner. Then, more Truchet-like, are hexagons divided into two tetragons which assemble to give vertices of average valence 4 if uncolored or 5 if colored with a single internal line, and the square tiles of Fig. 19 with eight vertices and three internal polygons the boundaries of which on assembly in any orientation generate nothing but quadrivalent vertices and form continuous lines extending or closing on any desired scale. As with any net of quadrivalent vertices, the first selection of one of two colors for one polygon determines the pattern of contrast throughout.

(This is, I think, very poorly explained, like the entire paper.)

These Fig. 19 "polygons" made of arcs are the patterns shown in Adrian Likins's 1998 xscreensaver hack "Truchet", along with a variant that replaces the quarter-circle arcs with straight diagonal lines.

(The Gardner reference is evidently to "Extraordinary non-periodic tilings that enrich the understanding of tiles," SciAm 236, No. 1, 110–221 (01977), which seems to have been about Penrose tilings, (certainly the cover depicts a Penrose tiling) but I don't have the article.)

It occurred to me that, instead of dividing each side of the square tile into two equal parts with an arc, you could divide each divide them into *three* parts with two points of division, perhaps with the center part being smaller or larger than the other two. If we letter the sides clockwise A, B, C, D and number the two points of division clockwise as 0 and 1, then we have eight points of division (in clockwise order: A0, A1, B0, B1, C0, C1, D0, D1), which we can connect with arcs, or not. One obvious possibility is A1-B0, B1-C0, C1-D0, D1-A0, and another is A0-A1, B0-B1, C0-C1, D0-D1. More interesting, perhaps, is A1-B0, A0-B1, which (unless we allow intersections or unconnected points) forces the connections C0-C1, D0-D1, and even that is only possible if the center interval isn't too long. This tile has four rotations. Allowing straight lines as well as arcs gives us another pair of tiles: A0-C1, A1-C0, B0-B1, D0-D1, and its 90° rotation. The lines on all of these (if, unlike Smith, we

omit the tile boundaries) join to form closed curves with no sharp angles and no intersection or branching.

Using two points of division per side enables us to also use a regular triangular tile, with division points A0, A1, B0, B1, C0, C1, and tiles including A0–A1, B0–B1, C0–C1; A1–B0, B1–C0, C1–A0; and possibly A1–B0, A0–B1, C0–C1 and its other two rotations. But this last tile is only free of intersections if the middle interval is fairly small.

A regular hexagonal tile with only a single point of division per side has only one obvious base tile: A–B, C–D, E–F, with two rotations. If we permit straight lines we also have A–B, C–F, D–E and its other two rotations.

Topics

- History (p. 1153) (24 notes)
- Math (p. 1173) (11 notes)
- Graphics (p. 1177) (10 notes)
- Composability (p. 1188) (9 notes)
- Art (p. 1306) (3 notes)

Thumbnail views in a Unicode character-cell terminal with Braille

Kragen Javier Sitaker, 02021-02-17 (updated 02021-02-24) (1 minute)

Braille Unicode characters fit 8 square pixels per character cell. We can make a thumbnail view of a text file (such as a C program) by mapping its character cells to these pixels; if we reduce 8× linearly, then each Braille character cell represents 8 columns of 8 lines with its 2×4 pixels. This maps 4 columns of 2 lines to each pixel. Some kind of threshold for how much text needs to be in this area to light up the pixel ought to work reasonably well to give a thumbnail view; if you can fit 40 lines of text on the screen then you can fit thumbnails of 320 lines of text, which is a lot; 410 out of the 443 "source files" (`*.c *.cc *.h *.ml *.java *.py *.lisp *.html`) in ~/dev3 are shorter than this. Also the thumbnail will only take up 10 columns at a standard 80-column width, so maybe you can do multiple columns of thumbnail.

Alternatively you could scroll horizontally by pages/columns, and put the thumbnailed pages at the top or bottom of the screen. This would take up 5 lines out of the 40.

Topics

- Graphics (p. 1177) (10 notes)
- ASCII art (p. 1305) (3 notes)
- Unicode (p. 1315) (2 notes)
- Temrinals

Energy autonomous computing

Kragen Javier Sitaker, 02021-02-18 (updated 02021-12-30)
(58 minutes)

I spent some time trying to figure out what it would take to be able to read, write, and interactively compute, without a connection to a power grid, with maximal autonomy. A big part of this is power usage, and it looks like it should be possible to get a self-sufficient computing environment that runs on under a milliwatt and doesn't need batteries, though batteries might enable orders of magnitude more computational power. See also How do you fit a high-level language into a microcontroller? Let's look at BBN Lisp (p. 160) for thoughts on how to design a software environment for such a computer.

In particular, it seems like with Sharp Memory LCDs, several of Ambiq's new line of subthreshold microcontrollers, amorphous solar panels, and supercaps, you should be able to do a low-resolution black-and-white GUI on the order of what you could do on a PowerMac 7100, SPARC 5, or 486DX2/66, without a battery, on 0.5 mW, with an average write bandwidth to your SD card of some 10 kilobytes per second (say, 10 megabytes per second at an 0.1% duty cycle). You could run a Web browser and PDF viewer, though not DHTML apps like Slack and Fecebutt, because it would only have a few megabytes of RAM across all the CPUs, and PDFs might be difficult to read at the low resolution of the screen. The whole computer might weigh 100 grams.

It would take a week to discharge when not in sunlight, but need only an hour or so of sunlight per day, or a few seconds of being plugged in, to stay fully charged. By scaling the clock frequency of the CPUs and turning more of them on, within a few milliseconds you could scale up to a billion instructions per second (comparable to a turn-of-the-millennium Pentium III or iMac G4, or an iPad 2 from 02011, or the original Raspberry Pi), though this would be limited by your available energy, since it would use close to 50 milliwatts.

Computation needs

Although computation isn't the only power cost in a computer, it's a fundamental one. But how much computation do you need? About 0.1 DMIPS for a minimal responsive computing environment (Apple][, Commodore 64, SDS 940); about 1 DMIPS for a reasonably comfortable one with a lightweight GUI and IDE (VAX 11/785, HP 9000/500, IBM PC/AT, Sun 3/60); maybe 10 DMIPS for workstation-class performance (SPARC 2, 386/40); maybe 100 DMIPS for browsers and rich GUIs (SPARC 20, RS/6000 250, PowerMac 7100, Pentium 120).

Historical computer performance: a 1-MHz 6502 was a bit less than 0.1 DMIPS

A Commodore 64 or Apple][were also capable of running the VisiCalc spreadsheet, the Berkeley Softworks GEOS GUI and geoPaint and whatnot, and Contiki, though not, say, the GEM

desktop. The 5MHz and sub-MIPS Apple Lisa was capable of running a non-janky GUI, but even on the Macintosh (7.8 some MHz, 16-bit ALU, 0.40–0.52 Dhrystone MIPS even though some 68000 machines were faster) it was slow enough that you could see the order in which the lines of dropdown menus got painted, top to bottom. With GEOS on the Commodore 64, though, you can see that it paints each line of the dropdown menu left to right, even with a memory expander cartridge, and in geoWrite, typing onto the end of a short line of centered text makes it flicker quite noticeably as it gets erased and repainted left to right in the new position.

The Magic-1 4-MHz homebrew microcoded TTL minicomputer gets 506 Dhrystone repetitions per second, while the same page says the Mac 512 gets 625. I guess those work out to `(mapcar (lambda (x) (/ x 1757.0)) '(506 625))` 0.29 and 0.36 respectively. 0.36 is a little lower than the 0.40–0.52 range in the netlib page cited above, but it's pretty close. The same page reports that a 2.5-MHz Z-80 did 91 Dhrystones per second, which is 36.4 Dhrystones per second per MHz, and that an Apple IIe only squoze 37 Dhrystones per second out of its 1.02 MHz 65C02, which by the same calculation is `(/ 37 1757.0)` = 0.021 DMIPS, and thus 0.021 DMIPS/MHz. And so that seems to be close to the minimum CPU power to run a usable GUI.

So the STM32F1 does about 50 (!) times as much Dhrystone work per clock cycle, and about 4–5 times as many instructions per clock cycle, so it's doing about 10–12 times as much Dhrystone work per *instruction*. This is substantially larger than the factor of 2 I had guesstimated for the 8-bit vs. 32-bit difference, and I suspect it's unrealistically large — an artifact of trying to benchmark the C-unfriendly 6502 with a C program, and perhaps using a lousy compiler to boot.

SRI's oN-Line System ran on an SDS 940 system at around 0.1 DMIPS

The Mother of All Demos, demonstrating the mouse, windowing, networked hypertext, multimedia computerized documents including images, and IDEs, was done in 01968 on an SDS 940 (one of some 60 ever built, over a third of which were sold to Tymshare) which supported 6 concurrent users, using specialized analog hardware for video compositing. The system's interactive response slowed notably when more than one person was using it actively. The SDS 940 had a 24-bit CPU and up to 64 kibiwords of 24-bit memory. An integer add instruction on its predecessor the 930 took 3.5 μs, and the memory's cycle time was 1.75 μs, so we can estimate it roughly at 200,000 instructions per second, about the same as the 1-MHz 6502 in the Commodore 64; but they were 24-bit instructions instead of 8-bit instructions, so it might have been perhaps twice as fast as the C64.

The 940 they were running NLS on was exactly the same in those respects: 0.7-μs memory access time, 1.75-μs cycle time, 3.5-μs "typical execution time" for integer addition "(including memory access and indexing)", and 7.0 μs for integer multiply.

The manual/brochure for the machine, which was built for the Berkeley Timesharing System under which NLS ran, says:

System response times are a function of the number of active users. Typical times are:

6 active users . . . . . 1 second
20 active users . . . . . 2 seconds
32 active users . . . . . 3 seconds

It's very unlikely that anybody ever ran Dhrystone on the SDS 940; its successor the SDS 945 was announced in 01968, and that was the last of the whole SDS 9 line; SDS continued to introduce upgrades to the 32-bit SDS Sigma series until 01974 (though that series began earlier, in 01966), until Xerox sold them to Honeywell in 01975. I think the last operational SDS 940 probably got decommissioned in the mid-1970s. (These things weren't cheap to run; the SDS 940 manual cited above says it used 3 "Kva", which is roughly kilowatts.) But Dhrystone wasn't written until 01984.

Amusingly, some former SDS employees refounded the company in 01979. Guess what CPU their new computer used?

A 6502A.

The vague handwaving argument above that the 1-MHz 6502's 0.02-DMIPS number is a little lower than would be realistic, along with the estimate that the SDS 940 was probably about twice as fast, combine to form a vaguer, even more handwaving argument that the SDS 940 was about 0.1 DMIPS, which was barely able to support 6 concurrent users with specialized analog display hardware.

This reinforces the above argument that 0.1 Dhrystone MIPS is close to the minimum practical for an interactive computing system.

Estimating the necessary performance for basic interactive computation: 0.1 DMIPS

My previous estimate in Dercuano was that basic interactive computation like word processing takes about 7500 32-bit instructions per keystroke. At one point, I said, "WordStar on a 2MHz (≈0.5MIPS) 8-bit CPU would sometimes fall behind your typing a bit," but then later calculated that a Commodore 64 (now AR$12000 = US$80) or Apple][would only do about 200 000 8-bit instructions per second and were usable for word processing, and a 32-bit instruction is roughly equivalent to two 8-bit instructions, so you need about 0.1 32-bit MIPS, and you might be typing like 160 wpm (13.3 keystrokes per second), which works out to about 7500 instructions per keystroke.

Also in Dercuano, I estimated that painting text in a framebuffer fast enough that it doesn't slow down reading at 350wpm might take about 50 bytes of I/O per glyph and 100 instructions / glyph × 350 wpm × 6 glyphs / word × 1 minute / 60 seconds = 3500 instructions/second. But that's orders of magnitude lower than the computations I discussed above. Indeed, old computers like the Sinclair ZX-81 with its 3.25-MHz Z-80, lacking an external framebuffer, would use the CPU to repaint the screen 50 times a second.

Modern microcontrollers run at around 1 Dhrystone MIPS per MHz

How fast are modern microcontrollers? dannyf compiled

Dhrystone 2.1 with a modern compiler and got 921 repetitions per second per MHz on an STM32F1 with what I guess is the vendor compiler, or 736 with GCC; to convert that into Dhrystone MIPS I think we divide by 1757, so that's 0.52 and 0.42 DMIPS/MHz. However, other commenters say the ARM Cortex-M0+ used in the STM32F1 is 0.93 DMIPS per MHz, and the STM32F103x8/STM32F103xB datasheet says it's actually "1.25 DMIPS/MHz (Dhrystone 2.1)". And apparently a CoreMark is about half a DMIP.

Computational energy consumption

How much battery you need, and whether you need a battery at all, and what other kind of power source you need, depends on how much power you need. So, how much power do computers need?

Laptops need tens of watts and often run off old 18650s

My "new" HP laptop's `/sys/class/power_supply/BAT0/power_now` produces numbers ranging from about 11 million to about 32 million, with CPU usage seeming to be the biggest determinant. (I'm guessing these are microwatts.) Its battery (four 18650 cells, I think) is so shot that it only runs for about an hour and a half on it, which suggests a capacity in the 60–180 kJ range, probably close to 100 kJ. The (HP V104) notebook battery is *labeled as* "14.8 V" and "41Wh" (though the broken off-brand spare says "2200mAh/33Wh"), and `/sys/class/power_supply/BAT0/energy_full_design` says 23206000, and bizarrely so does `energy_full`, and `energy_now` approaches that level (20364000 at the moment). `cycle_count` reports 208, but then after popping the battery out and back in, only 200. The docs say that energy is reported in μWh; `power_now` is not documented there but SuperUser says it's in μW, and indeed 23 watt-hours divided by 15 watts is about an hour and a half. I suspect the battery is worn out down to 57% capacity and just doesn't report its design capacity. 23 Wh at 14.8 V is 1600 mAh, which is in a reasonable range for the half-worn-out 18650s it presumably contains.

It can do something like 10 or 15 billion instructions per second, so this is something like 2000–3000 pJ per instruction, including the monitor.

An HPC paper from 02011 found that these numbers were typical of the most efficient large processors from the 02000s and 02010s: the Lenovo ThinkPad X201s i7-640LM (15 W idle, 33 W run, 13917.89 Dhrystone MIPS, 2400 pJ/"insn"), the Dell Inspiron 910 Atom N270 (7 W idle, 10 W run, 4683.57 Dhrystone MIPS, 2100 pJ/"insn"), and the SHARP PC-Z1 i.MX515 (2.2 W idle, 4.4 W run, 1184.58 Dhrystone MIPS, 3700 pJ/"insn"). Many other contemporary processors they tested (Pentium D, MV88F5281-D0, Xeon E5530, Core 2 Quad) were less efficient by an order of magnitude or even more.

Ereaders use on the order of 200 mW; maybe reprogram one?

The Amazon Swindle 2 uses a 3.7V-1530 mAh battery (20 kJ),

which reportedly lasts 7 days after Amazon tweaked the firmware, which I think is based on some nominal usage level per day — Amazon shill sites claim it's only 15 hours of active use, and people say they normally charge their e-readers around once a week or every 32–64 hours of use or so, with 6–8 weeks of standby time. So this is about 100–300 mW during active use. Other e-ink-based "ereaders" have similar battery lives. All of them have much higher display resolution than a Nokia cellphone display (typically 300 dpi instead of like 20 dpi), but evidently they also use two orders of magnitude more power, a problem which may not be entirely fixable in software.

Reprogramming and possibly rewiring an ereader (Amazon or otherwise) might be a reasonable approach. Since the Swindle Touch in 02011, most of them have touchscreens, and since the Paperwhite in 02012, most have backlights or frontlights, which can be turned off. New, these cost on the order of US$150 here, but older models are available used and supposedly working for AR$7000–10000 (US$50–70), 86% Amazon-branded (344 out of 398 current listings). Older devices tend to have more I/O options, while newer ones tend to be waterproof. An already-jailbroken Swindle is currently for sale for AR$17000 (US$115; Paperwhite 3, 1072×1448, 300dpi, LED frontlit, 1GHz CPU, 0.5 GiB RAM, "4/3" GB Flash, touchscreen). Non-Amazon ereaders like the Noblex ER6A15 (AR$12500, US$85) tend to run Android and have SD card slots, which have been removed from more recent models of the Swindle — though that one in particular has no data radios.

Some Swindles have LCDs instead of e-ink displays, and these have larger batteries; the Swindle Fire HD 7's is reportedly 4440 mAh, bringing its weight to 345 g, for example.

The Kobo seems to have the best reputation for hackability and can run the portable open-source ereader firmware "KOReader" (though there aren't many for sale; Forma 8 and Clara HD seem to be the options, but one Aura N514 remains, at AR$16000), and at least some models of the Barnes & Noble Nook pair a small, fast LCD with a large e-ink display, enabling you to have both rapid feedback and low power use, but none of them on sale now have this.

Report on rooting the Aura:

> I had read that some newer models had the NAND flash soldered onto the board, but mine is a Sandisk sdcard in a slot. So I pulled the card out, dd copied it, and I can restore if I do anything really bad that makes it stop booting.
>
> There is a well-marked ttl level serial port on the back. uboot is accessible and allows for interrupting boot. You can log into a root shell onto the running system without a password. It's basically open for business.

The Kobo company does have some despicable practices you have to protect yourself from.

The Kobo Clara is reputedly pretty easy to program without needing to jailbreak. It uses the same .kobo/KoboRoot.tgz firmware upgrade process earlier Kobos did.

Jailbreaking the Swindle is quite a pain by contrast, because Amazon keeps causing trouble.

Lower-power computing: low-power MCUs use 200

pJ/insn, Ambiq uses 30

Ordinary microcontrollers (without a monitor) are comparable to the laptop's 2000–3000 pJ/intruction power usage, or a bit lower, or much worse for floating-point or SIMDable computations, but low-power microcontrollers like the STM32L0 or the Atmel SAMD picoPower ARM chips are in the 150–250 pJ/insn range, and the MSP430 just a little higher (though only 16-bit).

Recent reports are that the new RISC-V-based microcontroller line "GD32V" are better by another factor of 3 or so. The datasheet for the GD32VF103 doesn't yet provide a lot of detail on lowest possible power consumption, but the numbers they do give say that it uses about 2.1 mW/MHz at 2 MHz, which drops to 0.7 mW/MHz at 36–48 MHz and 0.6 mW/MHz at 72–108 MHz, at 3.3 V, executing from Flash, with all peripherals off. You can probably improve this by running at 2.6 V, which is still kind of sad because the RISC-V core itself is running at 1.2 V. (The datasheet and user manual claim it's a Harvard architecture, so you probably can't execute from RAM, and executing from RAM does improve power consumption on ARM microcontrollers.) If we assume that's about one instruction per clock cycle, 0.6 mW/MHz is also 0.6 mW/MIPS (not Dhrystone MIPS!), which works out to 600 pJ per instruction. This is a little better than the STM32F0 (which I think is 12 mA at 3.6 V and 48 MHz: 900 pJ/instruction) but a lot worse than the STM32L0.

Ambiq has apparently finally brought to market subthreshold computing, Sparkfun sells an Ambiq Apollo3 devboard for US$15 with a 48MHz Cortex-M4F with 1MB Flash and 384k SRAM on the chip, running at 6 μA/MHz at 3.3V, which I think is roughly 5 μA/DMIPS, plus floating point. The 6μA figure is running a while loop, though; more typically the datasheet says it's supposed to weigh in at around 10 μA/MHz, which works out to 33 pJ/instruction, plus 68 μA overhead. This is higher than the 3pJ/insn research subthreshold processors I mentioned in file `low-power-micros` in Dercuano, but it's dramatically lower than the STM32L0, a dramatically less capable processor. Direct from Ambiq, the chip costs US$4.21.

XXX I uh have gotten confused between μA and μW in most of the rest of this note and need to redo a lot of calculations which are off by a factor of 3.3 now

0.1 MIPS at 1000 pJ/insn works out to 0.1 milliwatts, and at 33 pJ/insn it's 3.3 μW; 1 MIPS would be 33 μW; 10 MIPS would be 330 μW. It's worth noting that these are *average* figures; a 10-MIPS 80386-40 couldn't go any faster than 10 MIPS, but the Apollo3 can reportedly "burst" to 96MHz occasionally for Pentium-Pro-like responsiveness while averaging 100 μW.

Display energy consumption

At such low power levels, the processor might not be what consumes most of the power. For example, unless the UI is purely audio, you need a display, and it's easy for the display to use much more than that. In modern hand computers such as cellphones, it's common for the display to use the majority of the power.

My estimate from Dercuano was that updating an e-paper display

takes about 25 μJ per glyph; at 350 wpm and thus 35 glyphs per second, this works out to 875 μW, which is several times more than the 0.1 MIPS or even 10 MIPS. My friend Eric Volpe tells me that he's gotten old Nokia SPI screens (like the 84×48 Nokia 3310 screen, about 25 words of text) to maintain their display on less than 1 mA at 3.3 volts (though others say it needs 6–7 mA, and he reports that it consumes more when you're updating it, too). They're readable without backlight in direct illumination, but if you use it, he says the backlight also uses nearly a milliamp.

But now there are displays with much lower power consumption than that. Adafruit is selling a Sharp LS013B7DH05 memory LCD breakout board for US$25; it's 168×144 pixels in 24.5 mm × 21 mm (174 dpi), and they claim it draws 4 μA at 3.3V "with 1Hz data refresh", so 13 μW. The datasheet says you can write individual lines of data to it with a serial protocol at up to 1Mbps, but it's write-only, and the maximum frame rate is 60fps. The datasheet provides no information at all about power consumption, so I guess Adafruit's testing is all we have to work with. Delightfully, they've used a ZIF socket and double-stick tape so you can take the module off the breakout board without desoldering anything. Like cellphone screens, it's readable in sunlight. Sharp's brochure from 02015 says its power consumption is "60 μW*" and gives the power consumption of similar-sized displays as "10 μW static, 45 μW dynamic" and "15 μW static, 150 μW dynamic".

Although it's out of stock, in theory Adafruit also sells a bigger Memory LCD for US$45, 400×240 pixels and 58.8 mm × 35.3 mm (173 dpi), which I guess is the LS027B7DH01A, running on 5 V and "50 μW static, <175 μW dynamic", and which is monochrome but transflective. This is the display the Playdate handheld console (press blurb with demo video) is planning to use, according to the Wikipedia article. This resolution would give you 40 lines of 114 characters at 3.5×6 or 30 lines of 80 characters, which is quite comfortably usable. In Adafruit's demo video it seems to be at least 30fps.

A fellow named Mike made a video in 02011 with one of these 400×240 displays; he reported that you can update a single 400-pixel line at a time, and that he measured the power draw at 5 μA to hold a static display, or 3–4 μA if your "frame signal" is at its minimum speed of 4 Hz (max is 100 Hz). He also said the datasheet says you can clock pixel data in at 2 MHz, but he's been able to clock it up to 6 MHz with success, updating the whole screen in 40–50 ms. Also, interestingly, for the reflective (not transflective) ones, he suggests the interesting possibility of bouncing light off them for a projector, which could help with the small physical size of these displays.

There's also an EEVblog video from 02019 and forum thread about using these memory LCDs for "super" low-power design.

Digi-Key also has in stock a 320×240 LS044Q7DH01 Sharp Memory LCD for US$70, which is a 4.4-inch diagonal, and the tiny 184×38 LS012B7DD01 for US$16, demonstrated in the EEVblog video.

The smallest we can practically make readable English text is about 6 pixels tall and 3.5 pixels wide, in a proportional font, giving 24 lines

of 48 characters in 168x144; more traditional than proportional 3.5×6 is fixed-width 5×8, as used in lots of terminals in the 01970s and 01980s, which would give 18 lines of 33 characters. Either of these would qualify as "barely usable", I think. If we guess that Adafruit's 13 μW measurement is 10 μW of static power plus 3 μJ to refresh the whole screen, then Sharp's 60 μW* rating would be at 16.7 fps, but updating an 8-pixel line of text on the display would cost 170 nJ, two orders of magnitude less than my 25000 μJ Dercuano estimate for e-paper. (The datasheet says you can update a line at a time, but Adafruit says you have to update the whole screen.)

Doing this 35 times per second would cost 6 μW on top of the base 10 μW for a total of 16 μW. But my "35 glyphs per second" was to keep ahead of someone *reading*, not typing; that's really one line of text per second, and you probably don't have to redraw it 35 times! So it's more like 10.2 μW under the assumptions above. Basically, updating the display at full text reading speed probably costs an insignificant amount more than the display's static power consumption.

Suppose we have the 400×240 display consuming 100 μW, and an Ambiq CPU consuming another 100 μW, probably providing about 40 DMIPS, 400 times the minimum above, and closer in speed to an i860, an Alpha 21064, a SPARC 5, a 486DX2/66, or a PowerMac 7100/66, than to a Commodore 64 or an SDS 940. You'd have less RAM (more like an Amiga, a Mac, or a 386 than like these late-90s CPUs) but you can leverage enormous amounts of fast Flash to make the RAM feel bigger. These 0.2 mW are then plenty to get a *really good* responsive interactive computation environment. Smalltalk, GUIs, spreadsheets, IDEs, the whole works. Just, not modern web browsers, not without a lot more memory.

Memory energy consumption

Generally you probably want some kind of memory besides on-chip RAM, both so you don't lose your data when you run out of energy, and because it allows you to have more data than fits in on-chip RAM.

And you probably want a *lot* more than the few hundred K of on-chip RAM these microcontrollers give you. CP/M machines used for development work typically had dual 8-inch floppy drives, holding half a meg or a whole meg per disk, with a disk storage cabinet with 16–256 of these floppies in them, for a total capacity in the tens to hundreds of megs; a Commodore 64 might have a 170-KiB 1541 drive, but, again, a stack of hundreds of floppies, for tens of megs of space. A Sun-3 in the late 01980s might have 16 MiB of RAM (the base US$12000 models came with 4 MiB, while mine had 48 MiB, but that's because I bought it in the late 01990s after RAM was cheap); an internal disk of 71, 141, 327, or 654 MiB; and access to an NFS server with more than a gibibyte. That's what was needed to be comfortably productive with these machines.

Having a few hundred K of RAM on-chip, like the Ambiq chip does, can ease the cost of using such memory enormously. The kinds of paging and swapping schemes considered in How do you fit a high-level language into a microcontroller? Let's look at BBN Lisp

(p. 160) need much less off-chip traffic, and thus power usage, when most of their working set fits in the chip's own RAM.

This involves much less of a *speed* cost than accessing data in external memory in the historical systems I talked about above. Paging, swapping, or overlaying on floppies was painfully slow, with access times around a second and bandwidths of a few kibibytes per second, and even on hard disks access times were in the tens of milliseconds and bandwidths under a mebibyte per second. Modern spinning-rust hard disks can manage latencies in the milliseconds and bandwidths in the tens of mebibytes per second. By contrast, the forms of memory discussed here have latencies in the 10ns to 100_000ns ranges (except for writing to NOR, which takes tens of ms) and bandwidths in the range of 2–32 MiB/s, while the microcontrollers we're talking about can only access their internal SRAM at speeds around 64–128MiB/s. So it's much more a question of the *energy* hit.

Suppose we're willing to spend another 200 μW on accessing external memory on average. With crude factor-of-2 approximations, this gives us these fairly stringent, floppy-disk-like bandwidth limits (though, as with computation, these limits are averages rather than peaks):

	read a byte	**write a byte**	**read BW**	**write BW**	**cost per megabyte**
SPI SRAM	10 nJ	10 nJ	20kB/s	20kB/s	US$20
parallel SRAM	5 nJ	5 nJ	40kB/s	40kB/s	US$5
NOR Flash	0.5 nJ	2_000 nJ	400kB/s	0.1kB/s	US$0.50
NAND Flash	1 nJ	10 nJ	200kB/s	20kB/s	US$0.01 (or US$0.0001 as an SD card)
Ambiq MCU (!)	0.05 nJ	0.05 nJ	4MB/s	4MB/s	US$5

If instead we're willing to splurge 10 mW (average) on accessing external memory, it becomes a much more appealing option, though we are more often limited by interface speeds:

	read BW	**write BW**
SPI SRAM	2MB/s	2MB/s
parallel SRAM	2MB/s	2MB/s
NOR Flash	20MB/s	5kB/s
NAND Flash	10MB/s	1MB/s
Ambiq MCU	50MB/s	50MB/s

Even with this expanded power budget, we can still read faster from Flash than from parallel SRAM because the Flash chips use so little power.

The standby currents of the memories are in almost all cases insignificant.

Details follow.

SD cards

SD cards and even eMMC memory are very cheap nowadays, and Flash does not use energy to maintain data the way SRAM or DRAM does. Matthew Green tells me current Sandisk MicroSD cards can handle 90MB/s of write traffic and 2000 iops for a few minutes, but then start to have multi-second garbage collection pauses, and can deliver 200MB/s of read traffic. Sandisk 32GiB MicroSD cards,

possibly of this speed and possibly not, are locally available for AR$925, which is US$5.

NOR Flash

For things that get written rarely but need instant random-access reads, NOR is an option. You can get surprisingly cheap and fast SPI flash, as outlined in Can you do direct digital synthesis (DDS) at over a gigahertz? (p. 119): the 30¢ GD25D10C contains half a mebibit of NOR flash and can read it at 160 megabits per second. It uses about 0.1 μA (0.33 μW) in standby — you'd have to cut its power line with a MOSFET or something if you want to avoid that. Actively reading from it at this 160Mbps speed is supposed to cost 2.5 mA (8.25 mW), and *writing* costs *20* mA (**66 mW**) and is also slow as dogshit. After a snowstorm! Doing the math, reading a byte (of a long stream of them) at that speed costs about 400 pJ, plus whatever the processor spends on frobbing the SPI lines.

(How slow is writing? 100 000 000 ns (a suspiciously round number!) to erase a 4KiB sector (thus 24 000 ns to erase each byte), 30 000 ns to program the first byte, and 2500 ns to program each subsequent byte. This staggering total of 10 300 000 ns to write an erased sector brings the total time to 110 000 000 ns to erase and rewrite it, or 27 000 ns per byte, and the *energy* to write the byte to some 1800 nJ, on the order of executing 3000 instructions.)

SRAM

CMOS SRAM uses a little energy, but much less than DRAM; for example, the Cypress CY62136EV30LL-45ZSXIT is a 2-mebibit (256-kibibyte) 45-ns asynchronous parallel SRAM chip for US$1.11 that claims it typically uses 1 μA at 2.2–3.6 V to retain its data when in standby mode. The big problem with SRAM is that fast SRAM is all parallel-interface, so you need at least to spend at least 26 pins to talk to this chip, though this is less of a problem if you spend a 27th pin to pull its /CE line high — you can share the pins with other signals as long as they go to other things that also have some kind of /CE-like mechanism.

Still, spending 3 μW to get an extra quarter-mebibyte of 45ns SRAM seems pretty cheap. But that goes up to 6600 μW/MHz when you're actively frobbing it (2 mA at 1 MHz, says the datasheet front page.) That's, I guess, 3300 pJ per off-chip SRAM access, which is kind of high when we recall that we're only paying on the order of 500 pJ per instruction (or much less with Ambiq) and 400 pJ per byte read from SPI NOR Flash. Accessing the SRAM *once* costs as much as running 7 instructions, or *100* Ambiq instructions. You can write a byte of SRAM in 45 ns instead of the Flash's average 27 000 ns, and spending 6600 pJ instead of the Flash's 1 800 000 pJ, and it really is random access both for reading and writing, which the Flash very much is not.

The CY7C1020D-10VXI also mentioned therein is a smaller parallel CMOS SRAM with 10-ns access time and fifteen times higher cost per byte. But it's enormously more power-hungry, too: 3 mA rather than 1 μA in standby, according to the datasheet, and *80 mA* when being accessed at 100 MHz, which (at 5 V) is *400 mW*. That's still less energy per access when going full tilt — 4000 pJ per

byte instead of 6000 pJ — but the 3000-times-higher high quiescent draw means this chip has no place in milliwatt computing.

Another option is SPI SRAM chips like the US$1.20 Microchip 23K256, which has 32 kibibytes of SRAM, a 20MHz SPI interface, and runs on 3.3V; it uses 10 mA (33 mW) reading at 20 MHz (20Mbps) and idles at 1 μA (typical). The datasheet doesn't specify the write power usage; if we assume it's the same as the read power usage, then they're both around 1700 pJ per bit, or 13 000 pJ per (sequential) byte. (Random access costs four times as much.)

"Quad SPI" chips like the US$2.10 Microchip 23LC1024, with 128 KiB, are generally faster. It runs at 2.5–5 V and up to 20 MHz, purporting to use only 3 mA (10 mW at 3.3 V) reading at 20 MHz. Moreover, it can transfer data at 80Mbps instead of 20Mbps, so this ends up being 120 pJ/bit or 1000 pJ/byte, nearly as low as reading the GD SPI NOR Flash above. Typical standby current is higher at 4 μA, but not so high as to matter here.

NAND Flash

NAND Flash requires orders of magnitude less energy than NOR to write to, and it also costs orders of magnitude less per byte to buy it.

SD cards have NAND flash on them and can normally write with bandwidths of several megabytes per second, with latencies in the tens of μs.

Four promising bare NAND chips are the US$3 104MHz quad SPI 128-mebibyte Winbond W25N01GVZEIG TR, the US$2.50 120MHz quad-SPI 128-mebibyte GigaDevice GD5F1GQ4RF9IGR, the US$1 45-ns/25000-ns 48-pin parallel 128-mebibyte Cypress S34MS01G200TFI900 (whose datasheet has been memory-holed from Cypress's site but I found a datasheet for a clone on Mouser via Yandex after filling out a captcha in Cyrillic), and the US$2.30 50MHz quad-SPI 128-mebibyte Micron MT29F1G01ABBFDWB-IT:F TR whose datasheet I found the same way.

The SkyHigh Memory datasheet for the S34MS01G2 claims SLC, 25 μs (max) for random access, but 45 ns (min) for sequential access, and there are versions with 8-bit and 16-bit I/O buses (this is the 8-bit version); for writing, it takes 300 μs to program a 2048+64-byte page and 3 ms to erase a 64-page block. It uses the same 8-bit or 16-bit bus for address and data bits, so I have no idea why it has 48 pins; only 23 are used in the 8-bit version, and 31 in the 16-bit version, and 8 of those are power pins! So you only need 15 GPIO pins to talk to it.

To access the memory, first you clock in 1–4 command bytes, and then you feed in the address on the bus in four successive clock cycles while signaling the desired operation with some other control lines. A read command copies a page from the Flash into a buffer (in 25000 ns, apparently), signaled by the "ready/busy" pin going high, and then you can read out a word (8 bits on this chip, but 16 bits on 16-bit parts) every 45 ns, as you choose to toggle the read-enable pin.

Writing the memory may fail and need to be retried — at a different page address, probably. Also typically NAND chips have about 2%

bad bits.

It supports prefetching pages so you can overlap the 25000-ns copying-into-buffer with your reading of the previous page, and similarly you can send it data you're planning to write to another page while it's still burning in the data you sent before.

So it looks like kind of a pain to talk to, but still easier than you'd think from the 48-pin package; but how much *power* does it use?

It runs on 1.8 volts, and it claims to use 15 mA typical, 30 mA max, for all of read, program, and erase! That can't possibly be correct. (Can it?) And 10 μA typical standby current ("(CMOS)", whatever that means).

But if that *were* correct, it would work out to 27 mW for 22 million bytes *read* per second, assuming the 25000-ns overlap thing works out. So that's 1.2 nJ per byte, three times the cost of reading from NOR. *Writing* 131072 bytes (not counting the 64) supposedly requires 21.2 ms at the same 27 mW, plus 5.9 ms to clock them into the device, potentially overlapping, which would be only 4.4 nJ per byte. Avinash Aravindan of Cypress explains that this two orders of magnitude faster erasure, using much lower power, is characteristic of NAND, and Edouard Haas has an insightful article on the same subject, where he points out among other things that NOR permits single-byte write operations, and in his QSPI NAND article he points out that NOR uses 100 times more energy for erase+write than NAND.

The "obsolete" GD5F1GQ4RF9IGR is another 1.8 V 128-mebibyte NAND Flash, but this time SPI/dual-SPI/QSPI, with broadly similar performance: 400 μs (700 μs max) to program a (2048+128)-byte page, 3000 μs to erase a 64-page block, 80 μs to read a page, using 40 mA maximum active current (again, for all of read, program, and erase, so I guess that *can* be real) and 90 μA standby current. It has internal ECC, so you don't have to worry about bad bits. It actually looks like *higher* bandwidth than the 8-bit parallel chip — 120 MHz and quad-SPI gives you 60 megabytes a second instead of 22 — but its internal slowness more than compensates. It doesn't seem to have the pipelining feature the Cypress part has to overlap fetches with reads, or burns with loads. The Digi-Key page linked above is the 8×6mm 8-VLGA package.

This works out to 72 mW and 80+34 μs = 114 μs to read a page, so 56 ns and 4 nJ to read each byte (again, disregarding the "extra data"); writing a 131072-byte block takes 3 ms to erase, 25.6 ms to program (plus 34 μs per page to clock in the data, which might add another 2.2 ms) for 28.6 ms: 220 ns per byte, which means 16 nJ per byte.

I'm going to assume the other two gibibit NAND chips are similar.

More Ambiq MCUs

This is a pricey option, but for RAM, it might actually be the most reasonable one. Ambiq doesn't make memory chips, but for 68 μA you can stick another Ambiq Apollo MCU next to the first one and get another 384K of SRAM, and then you can communicate with it at tens of megabytes per second, though the capacitive load of traces in between the chips can become a problem. Charging 1 pF of

parasitic capacitance to 3.3 volts costs 5.4 pJ; doing it at 10 MHz (and then dumping the charge to ground) costs 54 μW, and in this context that is a significant cost. Still, I think we can estimate that communicating a byte between the processors will cost on the order of 50 pJ. This is at least an order of magnitude cheaper than anything offered by conventional CMOS memory chips.

This suggests that, although we probably want at least NAND flash in our low-power system both for mass storage and for stable memory, it may be more effective to add more RAM by turning it into a cluster rather than by adding RAM chips. Making it a cluster costs a little more, and it makes writing the software more complicated, but it also offers a lot of opportunities for reducing costly off-chip communication.

In addition to the Apollo3 Blue microcontroller on the Sparkfun board above, there's apparently also an Apollo3 Blue Plus with 768KiB of RAM for US$4.82, but only as a BGA. That's about the same price per byte as dedicated SRAM chips in conventional CMOS, although those are much faster.

Batteries

Maybe you need batteries. If so, what kinds of batteries are there, and what are their advantages and disadvantages?

Lead-acid batteries: 9–36 kJ/US$ at retail, mostly around 20 kJ/US$

Lead-acid batteries are generally cheaper than the lithium-ion type, even in 02021. At the very low end joules per buck drops dramatically; a 1.3-amp-hour 12V HiStarX LA612 battery goes for AR$900; at AR$147/US$ that's US$6.10 for 56 kJ, or 9.2 kJ/US$. By contrast, a 2-kg 7-amp-hour Risttone battery goes for AR$1220, US$8.30, 300 kJ, 36 kJ/US$. A 24-amp-hour deep-cycle golf-cart Press PR12240D goes for AR$9000, US$61, which is getting low again: 17 kJ/US$; while car starter batteries are in theory much cheaper, like a Rosler 65-amp-hour starter goes for AR$4900, US$33, 84 kJ/US$, but of course you can only use a fraction of that before you start killing the battery. Even with starter batteries, prices per joule go way up at the low end: a Yuasa 5.3-amp-hour 1.5kg 12N5-3B 12-volt motorcycle starter battery (230 kJ) is sold for AR$2500 (US$17, 13 kJ/US$).

Digging further suggests higher-capacity options like the 2.8-kg 9-amp-hour Moura 12MVA-9 for AR$2500 (US$17, 390 kJ, 23 kJ/US$), or, in the extreme, the Ultracell UCG 100-12 100-amp-hour deep-cycle gel cell for AR$38400 (4.3 MJ, US$261, 17 kJ/US$).

Low-power lithium-ion batteries are more expensive at 3–16 kJ/US$

Lithium-ion batteries are trickier to buy because of the profusion of fakery, but this Sanyo NCR20700b cell is specified at 4250mAh and 3.7V for AR$2500, which would be US$17 and 57 kJ, or only 3.3 kJ/US$. (The seller falsely claims it's an 18650. It's tested at 3.7–4.2 amp hours at 0.2–15 amps of discharge rate by what I think is an

independent tester, who weighed it at 61 g; this 53 kJ then works out to 870 kJ/kg.) But there are a lot of fake lithium-ion batteries like this UltroFite GH 18650 "6800 mAh" which sells for AR$427, which would be 91 kJ, US$2.90, and 31 kJ/$, nearly an order of magnitude cheaper and up in the lead-acid price range. (Lithium-ion batteries are already immensely cheaper per watt or amp rather than per joule, but so are capacitors.) USB "power banks" are even less controlled, but much more convenient to use; this Tedge H555 claims 10 amp-hours for AR$1700, US$11.50, 180 kJ, 16 kJ/$, and it probably has 18650s inside, which could be replaced, while this offbrand Libercam powerbank claims 20 amp-hours for AR$1500 and is too thin to contain 18650s. I'm guessing it's fake.

My "10050 mAh" powerbank (180 kJ) can recharge my phone about four times, which can keep it alive for about a week.

High-power lithium batteries are down around 1.4–1.6 kJ/US$ but 25 W/US$

I thought maybe the motorcycle starter batteries were about to get murdered by lithium, since lithium is so great at rapid discharge, but it's not so clear. The Yuasa 12N5-3B above is only 35 or 39 cold cranking amps, depending on who you believe, which is only like 450 watts (26 W/US$). At 3.7 volts and 15 amps the Sanyo cell, which is the same US$17 price, delivers only 56 watts; you'd need 9 of them (9 times the price!) to deliver the same starter power as the lead-acid beast, though admittedly the resulting 0.550 kg of lithium battery is noticeably lighter the 1.5 kg of the Yuasa battery.

However, 4 amp hours and 15 amps is a discharge rate of only "3.75C", and lithium-ion batteries for drones come in "C ratings" of "15C", "20C", "25C", "30C", and even "50C", though at a substantial penalty in joules per buck. Does this make them competitive for starting motorcycles? Well, a Blomiky SDL-853562 7.4V 1600mAh 25C radio-controlled car battery, for example, is listed for AR$6900 (US$47) and hypothetically ought to hold only 43 kJ (0.91 kJ/US$) but be able to deliver 40 amps. But that's still only 300 watts, and it costs more than twice as much as the lead-acid battery. Cheaper drone batteries like this Kitch Tech 30C 7.5-V 1200-mAh YZ-803063 for AR$3500 come closer — if real, that's 32 kJ for US$24 (1.4 kJ/US$), 36 amps, and 270 watts, but lead-acid still beats it by a substantial factor. This Zippy 25C 2200mAh 11.1V drone battery can purportedly deliver 610 watts (or 850 watts, 35C, in bursts) and is listed at only AR$4700 (US$32, 2.7 kJ/US$). 850 W / US$32 is 27 W/US$, within a stone's throw of the Yuasa price — but far from dramatically undercutting it.

Moreover, advertised C ratings are often fake, even outside Argentina.

Lithium thionyl chloride batteries might last decades

Lithium thionyl chloride batteries are really good at low self-discharge (0.5%–1% per year, self-discharging about 20% in 9 years — according to their plot, if you discharge their battery in 1500 hours at 2 mA, you get 2.6 Ah, and if you discharge it at 25 μA, it takes 80000 hours, or 9 years, and yields 2.1 Ah, so about 20% of its capacity was lost to self-discharge) and high energy density

(710 Wh/kg). About 0.175% of them fail each year, according to Tadiran's brochure. "High cost and safety concerns limit use in civilian applications. Can explode when shorted. Underwriters Laboratories require trained technician for replacement of these batteries," says Wikipedia.

Sadly, it seems that they are not available on MercadoLibre.

Strategies for avoiding batteries

XXX

Batch processing can wait until the sun is shining

For batch processing, it might make sense to wait until the daytime: an average watt of batch-processing power might be 7 watts during the 15% of the time that the sun is shining full force, and 7 Wp of solar panels only costs US$2.30, which is a lot less than US$6.60. Also you don't need a high-power battery charge controller.

Lower power opens up alternative power sources and stores, like supercaps

As I pointed out in Dercuano, a pullstring can yield 500 mm of pull at 50 N, which is 25 J, which (if harnessed with a dynamo) would run the laptop for a second or two, but that's enough to run a 10-milliwatt computer for 40 minutes, or a 100-milliwatt Swindle for 4 minutes. For that you don't need a battery; a capacitor will do.

You might think to go with ceramics, but that's still impractical for a pocket computer or laptop. Although you can get a 25V 4.7μF X5R 0805 Samsung cap for 2.2¢ or a 25V 10μF X5R 0805 Taiyo Yuden cap for 4.1¢, either in quantity 1000, ½CV² at the rated voltage works out to 1.5 mJ and 3.1 mJ respectively, so you'd need thousands of caps.

Getting to 25 J at under 50 V requires 10000 uF, and for that you need electrolytics or supercaps. You can get 22000-μF TDK electrolytics for a buck fifty, but those caps are only 16V, so you'd need five of them (or 18 of them to have a comfortable 2× margin on the voltage rating), and they're a bit bulky: 30 mm diameter, 32 mm tall.

Electrolytics are really optimized for charge/discharge frequencies of 60Hz up to a few kHz, though. This application has frequencies closer to a millihertz, albeit kind of sawtoothy. So a supercap like the US$4 5.5V 5F Illinois Capacitor DGH505Q5R5, the US$5 5V 1.5F Maxwell BMOD0001 P005 B02, or the US$2.50 5.5V 1.5F Illinois Capacitor DGH155Q5R5 would probably work; these are rated at 75 J, 18 J, and 22 J, respectively, and they're pretty small, in the last case 12 mm × 17 mm × 8.5 mm.

Supercaps are notorious for leakage, but that's in different contexts; the Maxwell supercap, for instance, is rated for 5 μA, which is a loss of 12 mV per hour, so it will lose its charge in a matter of weeks.

Another reason to use two or more is to allow faster charging — the Maxwell cap is only rated for 3.1 A of one-second surge current, and the DGH155Q5R5 for 2.8 A, so your pull-cord

charge would need to take a few seconds. The DGH505Q5R5 is rated for 8.4 A, though. Smaller supercaps might include the US$1 1F 2.7V Eaton HV0810-2R7105-R, which is rated to hold 3.6 J, but rated for 1.1 amps of pulse current, 8 mm in diameter, 13.5 mm long, and 1.2 grams. You'd need to use about 8–16 of these, giving you 8.8–17.6 amps of pulse current at up to 2.7 V, which would probably be enough for the pull cord.

Me, I'd be tempted to vastly oversize the capacitor bank, but that could be dangerous.

At 3.1 A, 5.0 V, and 1.5 F, you could fully charge the 3.4 g Maxwell supercap (datasheet) to its 18.8 J full charge (5.5 J/g) in 2.4 seconds. It's rated for 500,000 cycles, 4 years shelf life uncharged, or 10 years DC life at room temperature. I think all of these are typical of supercaps.

To charge the capacitor from the pull cord you need not just a dynamo but also something like a MPPT buck converter that regulates its output voltage to just above the present voltage of the capacitor bank (any voltage drop from the capacitor bank's ESR is, after all, wasted, and produces heat), then varies it a bit to do MPPT on the pull-cord dynamo. At some point, if overloaded as I did with the regenerative braking on Trevor Blackwell's scooter, it might need to just open-circuit the dynamo or connect it to a power resistor instead, but smoothly feathering into that by easing off of the maximum power point would be better than suddenly releasing all mechanical resistance. (That is what the scooter did, falling out from under me, but I think the mechanism was that I blew a fuse.)

Prospects for energy-independent computing

So I was thinking it might be worthwhile to buy a 12-volt gel cell like those mentioned above and rig up some power supplies for offline computation. The AR$1200 7-amp-hour Risttone battery mentioned above ought to be able to run this laptop at 15 watts for 6 hours (given appropriate boost conversion), or recharge this cellphone about 6 times, and there might be better deals out there too. Two or three such batteries, or a single larger battery, could perhaps power the laptop, or a fan, through a long night.

Standard photovoltaic solar panel modules are 990 mm × 1650 mm or thereabouts, deliver 200–400 Wp, and, at retail in Argentina, cost AR$12000–AR$24000 (US$80–160), on the order of 30¢–40¢/Wp. Smaller panels like this AR$3200 20-Wp jobbie do exist but cost more per watt (US$22, US$1.09/Wp in this case).

Suppose the solar capacity factor for residential solar panels here is 15%, so 100 Wp delivers 15 watts average. (It's fairly sunny here in Buenos Aires, but we get more clouds than California and Arizona deserts with their 29% and 25% capacity factors, and also residential panels may have to deal with shadows and suboptimal angling.) And suppose we want 24 hours of "autonomy", meaning, we can keep computing even when it's super cloudy; so a watt of average usage requires 24 watt-hours (86 kJ) of battery.

So each watt of usage requires 86 kJ of battery, which at 20

kJ/US$ costs US$4.30, plus 7 Wp of solar panel, about US$2.30 at retail, for a total of US$6.60, plus some amount of power electronics. So running the laptop all the time at 15 watts would cost a bit over US$100 of equipment; at 32 watts we're talking US$210. I paid AR$50k (at the time, about US$320) for the laptop a couple months ago. So powering it autonomously nearly doubles its cost! Also, 32 watts average at a 15% capacity factor means 213 watts of solar panel, which is a whole square-meter panel. It would occupy a significant fraction of the balcony and might attract unwanted attention.

In file `garden-light-panel.md` in Derctuo, I tested a 38-mm-square amorphous panel at 8 mW in full sunlight, though a more careful MPPT calibration might yield more, and I didn't measure the "full" sunlight. Surprisingly, unlike some other amorphous panels, I wasn't able to get a usable amount of power from it under indoor lighting conditions.

Topics

- Contrivances (p. 1143) (45 notes)
- Electronics (p. 1145) (39 notes)
- Pricing (p. 1147) (35 notes)
- Ghettobotics (p. 1169) (12 notes)
- Energy (p. 1170) (12 notes)
- Solar (p. 1203) (6 notes)
- Systems architecture (p. 1205) (6 notes)
- Microcontrollers (p. 1211) (6 notes)
- Independence (p. 1215) (6 notes)
- Memory hardware (p. 1250) (4 notes)
- Displays (p. 1261) (4 notes)
- RISC-V (p. 1276) (3 notes)
- Batteries (p. 1302) (3 notes)
- Ambiq (p. 1391) (2 notes)
- Ereaders

How do you fit a high-level language into a microcontroller? Let's look at BBN Lisp

Kragen Javier Sitaker, 02021-02-23 (updated 02021-08-18)
(76 minutes)

I've been thinking a lot about how to do low-power computing systems, like around 10 mW (see Energy autonomous computing (p. 143)), and I think I have a handle on how to do a reasonable interactive computing environment in that energy budget. This note mostly focuses on prospects for fitting reasonable *software* into the hardware constraints of such low power. See Energy autonomous computing (p. 143) for details on other aspects of the issue.

I skimmed Bobrow's 01966 "The Structure of a Lisp System using Two-Level Storage" the other day, and I was struck by his claim that BBN LISP was fast enough to be usable, even on an 18-bit PDP-1 (one of 53 ever made! And possibly the first!) with a 17–33 *milli* second drum memory (5–10 μs per word sequential read) and a 16-kibiword core memory (5 μs per random word access, four times the standard size), even for 2–3 concurrent users. This is about half the speed of a Commodore 64 and only 36 KiB of RAM, but the drum was 88 kibiwords and much faster than a Commodore floppy. I thought I'd go back and read it in greater depth to see how this was possible.

Having read it, I think a number of modern techniques, particularly compressed oops, index registers, lexical scoping, bytecode, JIT compilation, and generational garbage collection, should make it possible to build a system with considerably more bang per computron than Bobrow had to settle for. Also, modern low-power microcontrollers have amounts of RAM comparable to the PDP-1's, but are on the order of 1000 times faster, and have on the order of 1000 times more secondary storage available, which is about 100 times faster than the PDP-1's; this should allow real, comfortable self-hosting microcontroller development. All of this can run on about ten milliwatts.

Review of Bobrow's paper

He allocated 4 KiW of core to compiled code (mostly in a 3400-word "ring buffer"), 4 KiW to the system ("supervisor and permanent code", divided into six overlays), and the other 8 KiW to the stack and heap, and used two-word CONS cells. When paging a 256-word page of compiled Lisp in from the drum, the system linked ("relocated") it.

It seems like the system could manage at most some 6000 calls per second:

> Closed subroutine references to an in-core word through the [virtual memory page] map takes approximately 170 microseconds (because of the poor set of operation codes on the machine, and the lack of an index register).

Also, p. 16 (20/26) characterizes performance; it says that what we

might now call the "cache hit rate" for RAM was about 97.5%, with only 2.5% of CAR and CDR operations going to the drum. This consumed only 10% of system time, presumably because all parts of the system were unbelievably slow: 31k CONSes and 150k CDR and CARs in 35 minutes, thus about 5000 of these operations per minute or 90 per second.

This seems to have been the paper that first published shallow binding (though in a strange way that uses pointers into the call stack), tagged pointers for small integers ("offset by 300,000 octal", both positive and negative) and a form of BIBOP typing. I thought it also used CDR-coding, which I thought was the most interesting, but it doesn't really.

The primary objective of the BIBOP typing was to avoid having to reach out to the drum for type tests, especially for symbols (where both of the usual predicates can be answered without waiting for a drum reference, namely ATOM and EQ) and so a contiguous chunk of the 17-bit (!) virtual address space was assigned to almost every type: value cells† starting at 00150_000, property list cells†, full words (large integers), pushdown list (the call stack) starting at 00200_000, function cells†, print-name pointers†, print-names, and "hash table", which last runs from 00270_000 (0x17000) to 00300_000, although all of this this seems to conflict with the earlier statement that small ints from -32767 to +32767 were offset by 00300_000, which would have them occupy the "address space" from 00200_000 to 00400_000. The virtual addresses from 0010_000 to 00150_000 are shared between "list structure" (shown growing down) and "compiled code" (shown growing up); although I have no idea how you would then distinguish a code pointer from a CONS, perhaps code pointers as such weren't first-class values.

(A secondary benefit not mentioned is that CONS cells occupied two words instead of three, because they don't need a type tag, but I guess devoting an entire word to type tags for dotted pairs was not considered as an option.)

He says this static allocation is preferred to "provid[ing] an in-core map of storage areas" because it reduces resident memory pressure (at the expense of virtual memory space efficiency). I suppose the 12 KiW of physical core being used for paging amounts to 48 pages, which is indeed pretty tight. If you divided the 17-bit address space up to 00400_000 into "storage areas" of 004_000 homogeneous words, your type map would have 64 entries. If you could get by with a 3-bit type code for each "storage area", you could squeeze 6 of them into each memory word; 4 per word would require 16 words of memory for the BIBOP map. They must have been pretty desperate to save 16 words of memory. But then, a static type allocation avoids writing code for dynamic allocation, too.

Above there are four items marked with "†"; these are the four virtual memory areas associated with symbols ("each literal atom"). Each is 0010_000 in size, so I guess the system supported a maximum of 4096 symbols, and these were in essence parallel arrays. "A pointer to an atom* points to its value cell." They're separated in virtual memory because accesses to them tend to be not merely uncorrelated but actually negatively correlated — symbols used for functions tend

not to be used for variables ("value cells"), almost nothing uses property lists, and print-name pointers are not normally used during actual computation. 4096 words is 16 256-word pages, so I guess you'd page in a sixteenth of the symbols' value cells or function cells or whatever at a time.

* They did also consider numbers to be "atoms", just not "literal atoms".

Aha, here's the resolution to the dilemma above about overlapping memory areas: although the call stack starts at 00200_000, followed by function cells, print-name pointers, print names, and the hash table, none of these are first-class values! If you want to refer to a symbol, you point to its value cell, not its function cell or print name, so it's totally kosher to use the addresses at 00200_000 and up for small ints. Moreover the sequence simplifies some other type tests: anything over 00150_000 was an "atom" (symbol or number), but anything over 00170_000 was an integer, and anything over 00200_000 was a *small* integer. So the boundaries were chosen with an eye to subset relations.

The "CDR-coding" didn't actually use less space; the CDR pointers were still fully materialized! It just attempted to maximize locality by allocating `(cons x y)` on the first possible of `y`'s page, `x`'s page, some page in core, or some swapped-out page. This used a per-page free list and an in-core map of per-page free lists, which I suppose must have used 192 words (!!) for the 49152 words of virtual memory available for "list structure".

They also mention an "additional scheme for dumping onto secondary storage (magnetic tape)" which I thought might have done the full CDR-coding thing, but it doesn't seem to have done so. This is referenced to "Storage Management in LISP", Bobrow, in preparation, Proc. IFIP Conf. on Symbol Manipulation Languages, and "A LISP Garbage Collector Algorithm Using Serial Secondary Storage", Minsky, AIM-58, 01963.

Minsky's paper seems to be scanned in full but with the page order askew; it contains the longest program I've ever seen written in the M-expression dialect of LISP, which is not very long at all: 27 lines of code. It focuses on writing a sequence of (x, y, z) triples out onto the drum, which, when read back in, put the dotted pair `(y . z)` into register (memory location) `x`. So a linked list of 20 addresses stored in the cars of 20 consecutive dotted pairs will need *60* words on the drum, even more than in BBN LISP. Minsky comments:

> *Collect* has the additional feature that all *cdr* sequences end up linearly packed! There are probably some important applications of this.

But I think compaction was his main focus, and I don't think of that as a hard problem, so I haven't taken the time to grok *collect* in fullness.

Prospects for microcontroller systems

An STM32F103C8 as used in a Blue Pill runs about 72 MIPS, 64 or usually 128 KiB of Flash, and 20 KiB of SRAM; the preassembled board costs about US$2. This chip uses about 1.5 nJ per 32-bit instruction (32.8 mA at 72 MHz, mostly one instruction per cycle, at 2.5 V, is 83 mW, or 1.1 nJ per clock cycle; but this is the datasheet's

max current consumption at 85° with all peripherals disabled; and lower voltages and higher clock speeds may be doable), and it can run ARM code from SRAM. But 20 KiB isn't a lot of space, so you need some external memory.

As mentioned in Energy autonomous computing (p. 143) and Can you do direct digital synthesis (DDS) at over a gigahertz? (p. 119), you can get a variety of memory chips to interface to the STM32; a rough outline of chips I think may be representative follows:

			read		write				
	type	**Cost**	**ns/byte**	**nJ/byte**	**ns/byte**	**nJ/byte**	**bytes**	**nUS$/byte**	**GPIOs needed**
GD25D10C	2SPI NOR	30¢	50	0.4	27000	1800	512Ki	4600	4? (SPI)
CY62136EV30	SRAM	111¢	45	6	45	6	256Ki	4200	25? (parallel)
23LC1024	QSPI SRAM	210¢	100	1	100	1	128Ki	16000	4? (SPI; 6 if QSPI)
S34MS01G2	SLC NAND	100¢	45	1.2	160	4.4	128Mi	7.5	15? (parallel NAND)
GD5F1GQ4	QSPI NAND	250¢	56	4	220	16	128Mi	19	4? (SPI; 6 if QSPI)

The parallel SRAM could actually be used in 16-bit-wide mode and get double the bandwidth and half the power usage, but use 7–8 more GPIO pins.

In theory it should be possible to gang up a bunch of SPI RAMs or SPI flash memories (of the same type!) and read from them in parallel, whether or not you can write to them in parallel. This would allow you to address them with a one-pin address bus.

NAND Flash must be read and written a page at a time — the two NAND chips profiled above use a 2048-byte page size. It's very appealing as a form of secondary storage, especially for frequently-written data: the energy cost per byte written is lower than anything but a (volatile!) SPI SRAM, and the dollar cost per byte of capacity is about three orders of magnitude lower than the alternatives. The access latency doesn't show up in the table, but it's on the order of 50–100 μs, which is nearly 1000 times slower than the 50–200 *ns* of the other memory types, coincidentally almost precisely in inverse proportion to the price per byte.

NAND also comes conveniently packaged with a ready-made FTL and SPI interface in the form of SD cards and MicroSD cards, but I imagine there might be power costs as well as the well-known reliability problems introduced by shitty FTLs.

Writing frequently-read data to a NOR Flash might actually save power as well, but this is an extreme measure: you'd have to read a given byte from NOR instead of the 23LC1024 SRAM 3000 times before paying for the cost of writing it into the NOR.

So, suppose we have a system consisting of an STM32F103C8 microcontroller, a 23LC1024 QSPI SRAM, and a GD5F1GQ4 QSPI NAND chip. When idle it's using 20 μA at 2.5 V in the MCU, 4 μA at 2.5 V in the 23LC1024, and 90 μA at 1.8 V in the NAND unless we turn it off with a transistor — 162 μW for the Flash and another 60 μW for the rest of the system. A 2048-byte major page fault

would initially take 114 µs at 72 mW in the Flash, working out to 8200 nJ, and about 34 µs at, I don't know, 20 mA and 2.5 V in the MCU, so 50 mW and another 1700 nJ, total about 9900 nJ. At that point it's occupying one of the 10 page slots in the MCU's internal SRAM. Evicting that page to the SRAM (you might want to evict a subpage instead) would take another 200 µs and 10 000 nJ in the CPU, plus 2000 nJ in the SRAM, which brings up the very interesting point that low-power SPI SRAM stops being so low-power if it keeps the CPU busy longer!

(The above numbers may be too optimistic about the MCU; it's more like 30 mA at full speed than 20 mA, according to the datasheet, though that's the worst case at 85°.)

A tiny QSPI controller like the Raspberry Pi Pico's "pioasm" coprocessors that allowed the CPU to page while continuing to compute would be super helpful. The STM32F103C8 does have a 7-channel DMA controller which can run its two hardware SPI peripherals, so that might actually be doable. The SPI peripherals support dual SPI but not quad SPI, so instead of 100 ns and 56 ns per byte, it would be 200 ns and 112 ns per byte, perhaps with correspondingly increased power usage in the peripherals — but you could power down the CPU or have it work on something else. This might increase the cost of major page faults to 16 µJ, with a latency of 150 µs, and decrease the cost of minor ones to 4 µJ, with a latency of 400 µs.

Paging out a dirty page to NAND would also take about 400 µs.

At this rate you could do 2500 minor page faults (plus evictions and pageouts) or 6700 major page faults per second, but the first would cost 10 mW and the second would cost 110 mW. 2500 page faults is about 20 times what you can do on spinning rust.

(Uh, but actually the SPIs on this MCU are only good up to 18 Mbit/s... maybe parallel memory isn't such a bad option? Or a beefier chip?)

Bigger microcontrollers

The STM32F103C8 mentioned above costs US$5 in the usual venues when it's not out of stock, though evidently the Blue Pill makers have found cheaper chips. But 20 KiB of internal SRAM is small enough to be restrictive. What about bigger STM32F chips, or maybe an ESP32? The STM32F line goes up to 512KiB SRAM. Beefier chips might also help to speed up off-chip communication and thus get it over with faster, perhaps saving energy. To my surprise, many of them also use less power!

Chip	SRAM size	Price	Power draw at 72MHz
STM32F103C8T6	20 KiB	US$5	30 mA?
STM32F410C8U6	32 KiB	US$3.50	8 mA?
STM32F401RCT6TR	64 KiB	US$5.50	9 mA?
STM32F730V8T6	256 KiB	US$6	28 mA?
STM32F765IGT6	512 KiB	US$15	28 mA?
ATSAMG53N19B-AU	96 KiB	US$1.50	7.2 mA (but only goes up to 48MHz)
CY9AF156NPMC-G-JNE2	64 KiB	US$2	¿no datasheet⸮
M481ZE8AE	64 KiB	US$2.50	¿no datasheet⸮

Some of these, like the STM32F730 and STM32F765, have built-in memory controllers designed to interface to external SRAM, SDRAM (!!), NOR Flash, and even NAND Flash.

See also Energy autonomous computing (p. 143) for notes on the Ambiq chips, which are both much lower power and have more RAM.

Full context switching and deep power down

Suppose that we want to save the whole RAM state of the microcontroller in stable memory, like the NAND Flash, either because we want to shut the whole system down or because we want to switch to a totally different task. (Maybe we don't even want to save, just reboot.) How expensive is that?

Using the STM32F103C8 and the S34MS01G2 numbers above, writing 20 KiB at 160 ns per byte and 4.4 nJ per byte is 3.2768 milliseconds and 90 μJ. Reading it at 45 ns per byte and 1.2 nJ is 900 μs and 24 μJ. The S34MS01G2 claims 10 μA "typical standby current". The STM32F103C8 uses another 20 μA at idle (not "standby"! "Stop," with the RTC.) So you can drop power usage by 95% by powering off the NAND but not the microcontroller. Saving the microcontroller's state this way might cost 9 seconds of the NAND's standby energy usage, or 3 minutes of the processor "stopped", which is 3 milliseconds of running the processor flat out at 30 mW or so.

So it should be energy-affordable to checkpoint the microcontroller's state this way on the order of 20 times per second during continuous computation, if that's a useful thing to do. And it might make sense to cut power to the NAND when it's okay for it to be unresponsive for a while — it claims to need 5 ms to get back to operational when you power it back on, and 3 ms to erase a page, so it might make sense to power it down with a MOSFET whenever it's been idle for tens of milliseconds, thus cutting power use by a third and extending battery life by 50%.

When you're handling 90wpm keystrokes, we're talking about like 110 ms on average between keystrokes. If we need 48000 instructions per keystroke, which seems grossly excessive, that's 1 millisecond of computation, so about a 1% duty cycle. This might add 500 μW of power usage, 50 μJ per keystroke, which is strikingly close to the cost of reloading the microcontroller's entire state from the NAND! Still, it's probably better to leave the NAND powered down most of the time.

The STM32F103C8 has a lower-power shutdown mode called, confusingly, "standby", where it uses two or three μA instead of the 20 in the "stop" mode discussed above, which would extend idle battery life by a factor of 6–10. It's harder to wake from, it takes 65 μs to wake instead of 5 μs, and the SRAM loses its contents, so if you go into standby mode you *do* have to reload the whole system from NAND or whatever. This seems reasonable for events like keystrokes where 6 ms is an acceptable response time, but still, rebooting could cost tens of microjoules if it involves reloading the

full context. The 30 μW “stop mode” uses this much about once a second, so when keystrokes are coming in more frequently than that, it’s cheaper to just stop and not reboot for every keystroke.

For handling 1kHz mouse movements (125 Hz is a more typical mouse sampling rate) these deep power-down modes clearly don’t make sense.

The STM32L line includes microcontrollers like the STM32L011x3/4 with even lower power usage: 0.54 μA in “stop” mode rather than 20, and at full speed, 1950 μA at full speed (though that’s only 16 MHz) instead of 29000 μA. It might make sense to use a processor like this as a “supervisor” processor for the whole system, always on, and powering the other hardware up only when needed. That chip in particular has only 2KiB of SRAM, but to the extent that that’s sufficient for a user interaction, the rest of the system could remain turned off.

The 23LC1024 SRAM, if present, uses another 4 μA of standby current. It’s 128 KiB, so saving it to the NAND Flash (in preparation for unplugging it) would take almost 600 μJ and 21 ms at 4.4 nJ and 160 ns per byte. Leaving it unplugged for a minute (at 4 μA and 2.5 V) would pay back that energy cost, so unplugging it that way is only worthwhile when its idle time, or potential idle time, is up in the tens of seconds to minutes.

The upshot of all this is that it should be possible to reduce the system’s idle power consumption from some 35 μA (and nearly 100 μW) down to 20 μA with just the STM32F103C8, or down to 0.54 μA with an STM32L011, while keeping latency-to-full-responsiveness well under 10 ms. Even when actively editing text or browsing hypertext, it should be possible to stay below 500 μW, only leaping up to the full 83000 μW figure mentioned earlier when you’re actually computing something new, and maybe when you have the thing plugged in.

Power store	**1 μW**	**50 μW**	**100 μW**	**500 μW**	**83000 μW**
2 J supercap	23 days	11 hours	5 hours	67 minutes	24 seconds
25 J pull on a pullstring	10 months	6 days	3 days	14 hours	5 minutes
2.2 kJ CR2032 coin cell	70 years†	17 months	8 months	7 weeks	7 hours
4250 mAh Li-ion 20700 (57 kJ)	1800 years†	36 years†	18 years†	43 months	8 days
2-kg 7Ah 12V lead-acid (300 kJ)	9500 years†	190 years†	95 years†	19 years†	6 weeks

† A CR2032 or 20700 only has a shelf life of about 10 years, so it will not last 18 years, much less 9500.

Compact in-memory representations

The NAND Flash discussed above is huge compared to everything else; the only penalty for bloated data representations in it is power usage and memory bandwidth usage. But for storing data in the microcontroller’s RAM, it’s crucial to represent it efficiently. Lisp data is pretty convenient to compute on but takes up a lotta fricking bytes.

Backwards CDR-coding stacks occurred to me, but that's probably dumb

What I originally thought when I skimmed Bobrow's paper was that he was going to CDR-code lists in something like the following fashion. When some datum z produced by (cons x y) is placed on a page different from its cdr, it gets allocated a shortish buffer, with a counter and length packed into a word at its beginning:

```
|-----+---+---+---+---+---+---|
| 2/7 | y | x | - | - | - | - |
|-----+---+---+---+---+---+---|
```

If later we execute (let ((w (cons a z))) ...) the first thing we check is to see whether there's more space available in z's buffer. There is, so we stuff a into it:

```
|-----+---+---+---+---+---+---|
| 3/7 | y | x | a | - | - | - |
|-----+---+---+---+---+---+---|
```

Now, if we later cons something else onto w, like b, we may well be able to pack that into the same buffer:

```
|-----+---+---+---+---+---+---|
| 4/7 | y | x | a | b | - | - |
|-----+---+---+---+---+---+---|
```

Eventually the buffer gets full, or we try to cons two things onto the same thing, so then we allocate a new buffer, ideally on the same page. If the buffer got full we might want to allocate a bigger buffer.

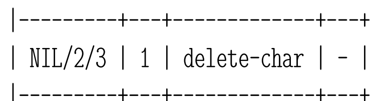
This is all assuming there's some sensible way to represent and dereference pointers into these buffers, and of course it would make RPLACD (set-cdr!) quite tricky.

With a buffer size of 8 words, like the above, we can store 6 cons cells in it, which would normally take 12 words, which is a best-case savings of 33%; as the buffer size grows, this approaches 50%. The worst-case inflation for 8 words is 4× (300%) but most lists are longer than that. The break-even point is at 4-item lists.

A quick Scheme program to count the lengths of lists, applied to itself, found 2 lists of length 0, 3 lists of length 1, 33 lists of length 2, 22 lists of length 3, 12 lists of length 4, and one list of length 5, which was the whole program and therefore possibly should not be counted. This is clearly not a fully representative sample of the data structures Lisp programs manipulate but it's probably true that lists of length 2 and 3 are much more common than other lengths. (Also, none of the lists were improper.)

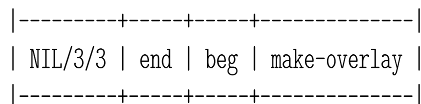
So, possibly a more optimal solution would be to use an initial buffer of size 4, and use a header bit to indicate whether the list is

NIL-terminated or improper (or continued elsewhere). Then a 2-item list would be 4 words, the break-even point:



```
|---------+---+-------------+---+
| NIL/2/3 | 1 | delete-char | - |
|---------+---+-------------+---+
```

And a 3-item list would also be 4 words, a 33% saving over the standard approach:



```
|---------+-----+-----+--------------|
| NIL/3/3 | end | beg | make-overlay |
|---------+-----+-----+--------------|
```

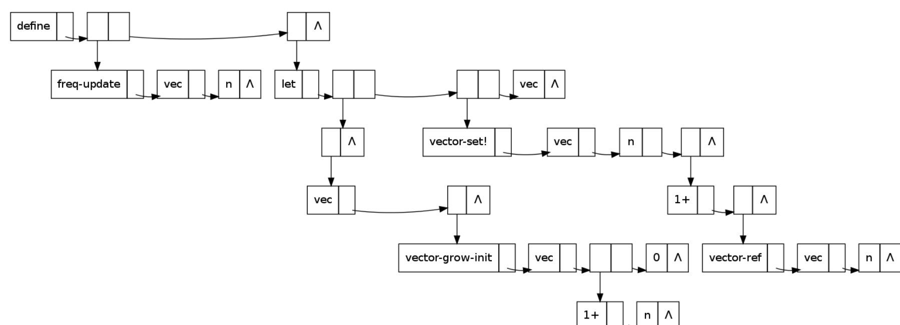
If we initially allow such buffers ("obstacks"? "arenas"?) to extend to the end of their page, only capping them off at a bit over their current capacity when we want to allocate something else, we can reduce the number of spills. For example, consider parsing this Scheme into lists through recursive descent:

```
(define (freq-update vec n)
  (let ((vec (vector-grow-init vec (1+ n) 0)))
    (vector-set! vec n (1+ (vector-ref vec n)))
    vec))
```

Here's the conventional dotted-pair representation, as outlined for example in "A Command Structure for Complex Information Processing" from 01958 (p. 11, 13/54):

> Each word is divided into two parts, a *symbol* and a *link*. ... The link is an address; if the link of a word *a* is the address of word *b*, then *b is adjacent to a*. That is, the link of a word in a simple list is the address of the next word in the list.

(See Some notes on IPL-VI, Lisp's 01958 precursor (p. 196) for more notes on this amazing paper.)



In Graphviz:

```
digraph fu {
        node [shape=record, label="<car>|<cdr>"]
        {
                rank=same;
                1 [label="define|<cdr>"];
                1:cdr -> 2;
```

```
        2:cdr -> 6;
        6 [label="<car>|Λ"];
}
2:car -> 3;
{
        rank=same;
        3 [label="freq-update|<cdr>"];
        4 [label="vec|<cdr>"];
        5 [label="n|Λ"];
        3:cdr -> 4;
        4:cdr -> 5;
}
6:car -> 7;
{
        rank=same;
        7 [label="let|<cdr>"];
        7:cdr -> 8;
        8:cdr -> 10;
        10:cdr -> 12;
        12 [label="vec|Λ"];
}
8:car -> 9;
9 [label="<car>|Λ"];
9:car -> 13;
{
        rank=same;
        13 [label="vec|<cdr>"];
        13:cdr -> 14;
        14 [label="<car>|Λ"];
}

14:car -> 15;
{
        rank=same;
        15 [label="vector-grow-init|<cdr>"];
        15:cdr -> 16;
        16 [label="vec|<cdr>"];
        16:cdr -> 17;
        17:cdr -> 18;
        18 [label="0|Λ"];
}

17:car -> 19;
{
        rank=same;
        19 [label="1+|<cdr>"];
        19:cdr -> 20;
        20 [label="n|Λ"];
}

10:car -> 21;
{
        rank=same;
        21 [label="vector-set!|<cdr>"];
        21:cdr -> 22;
```

```
            22 [label="vec|<cdr>"];
            22:cdr -> 23;
            23 [label="n|<cdr>"];
            23:cdr -> 24;
            24 [label="<car>|Λ"];
    }

    24:car -> 25;
    {
            rank=same;
            25 [label="1+|<cdr>"];
            25:cdr -> 26;
            26 [label="<car>|Λ"];
    }

    26:car -> 27;
    {
            rank=same;
            27 [label="vector-ref|<cdr>"];
            27:cdr -> 28;
            28 [label="vec|<cdr>"];
            28:cdr -> 11;
            11 [label="n|Λ"];
    }
}
```

With recursive descent, the first cons we do is of `vec` onto `'()`, creating a new buffer:

```
|-----------+-----+-----
| NIL/1/256 | vec | ...
|-----------+-----+-----
```

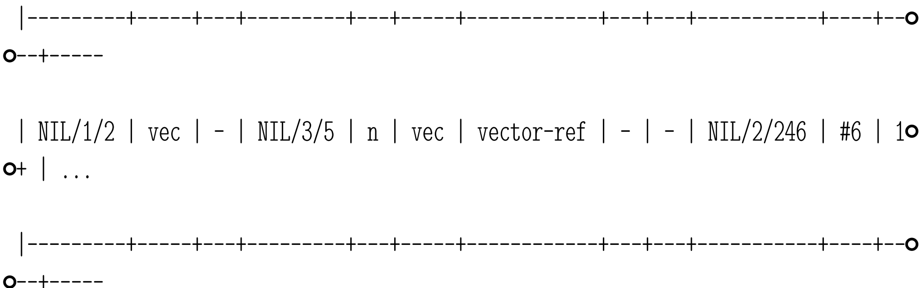
Then we start a totally new list by consing `n` onto `'()`, so we cap off this one with a little room to grow and start a new one:

```
|---------+-----+---+-----------+---+-----
| NIL/1/2 | vec | - | NIL/1/252 | n | ...
|---------+-----+---+-----------+---+-----
```

Then we cons `vec` and `vector-ref` onto that:

```
|---------+-----+---+-----------+---+-----+------------+-----
| NIL/1/2 | vec | - | NIL/3/252 | n | vec | vector-ref | ...
|---------+-----+---+-----------+---+-----+------------+-----
```

Then we cap off that entire list and cons it onto `'()` followed by `1+`:



```
|---------+-----+---+---------+---+-----+------------+---+---+-----------+----+--o
o--+-----

| NIL/1/2 | vec | - | NIL/3/5 | n | vec | vector-ref | - | - | NIL/2/246 | #6 | 1o
o+ | ...

|---------+-----+---+---------+---+-----+------------+---+---+-----------+----+--o
o--+-----
```

So far we have 6 conses in 12 words, which is a singularly uninspiring performance.

A depth list

A maybe more interesting representation of the above S-expression might annotate a flat list of atoms with their depths:

```
(define (freq-update vec n) (let ((vec (vector-grow-init vec ...
    1          2       2  2    2     4             5          5
```

but of course this is not only just as many words of memory, it does not contain enough information to reconstruct the S-expression!

An RPN operation stream

An RPN approach to reconstructing it might intersperse a series of N-ary operators on the symbol list:

```
1+ vector-ref vec n 3! 2! vec 4!
```

which at least compresses the six conses above down to 7 words instead of 12. For functions of fixed arity the N-ary operators are, strictly speaking, unnecessary — forward Polish notation is sufficient, so you could imagine representing that S-expression with the following whitespace-separated tokens:

```
define( freq-update vec n ){
    let( vec vector-grow-init vec 1+ n 0 ){
        vector-set! vec n 1+ vector-ref vec n
        vec
    }
}
```

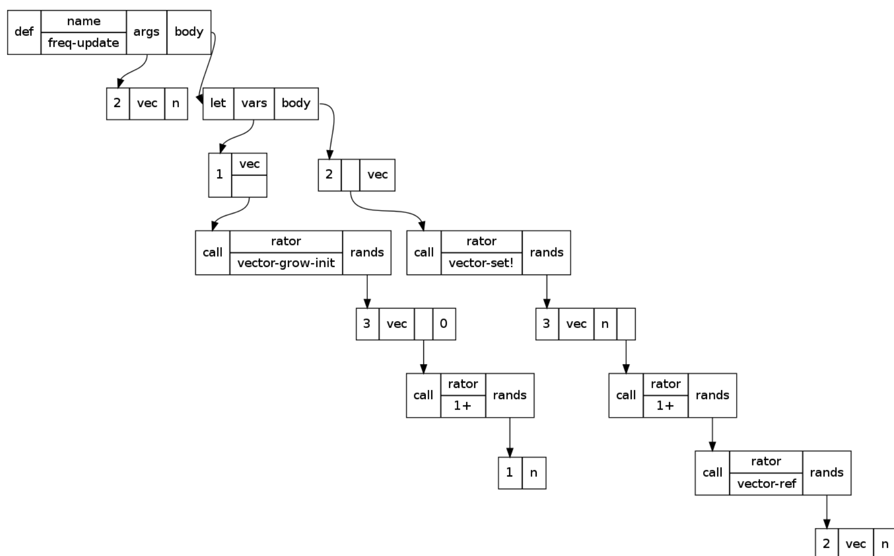
That successfully represents the function code in 23 words, rather than 28 conses and thus 56 words, but at the cost of pushing most of the arity information into the symbols instead of the conses, which makes it quite awkward to compute with.

Algebraic data types, or variant records

In the particular case that what you want to represent is something analogous to Scheme source code, a structier approach may work

better, using ML-like pattern-matching and constructors, but Golang-like slices for things like argument lists. That is, while you're building up an argument list, by all means use a cons list, but once you're done building it, freeze it into a counted array, which you can comfortably cdr down with a slice. So, for example, `let` here takes some variable-initializer pairs and some body expressions, in this case one of the first and two of the second; and a function call takes a function expression and some number of argument expressions.

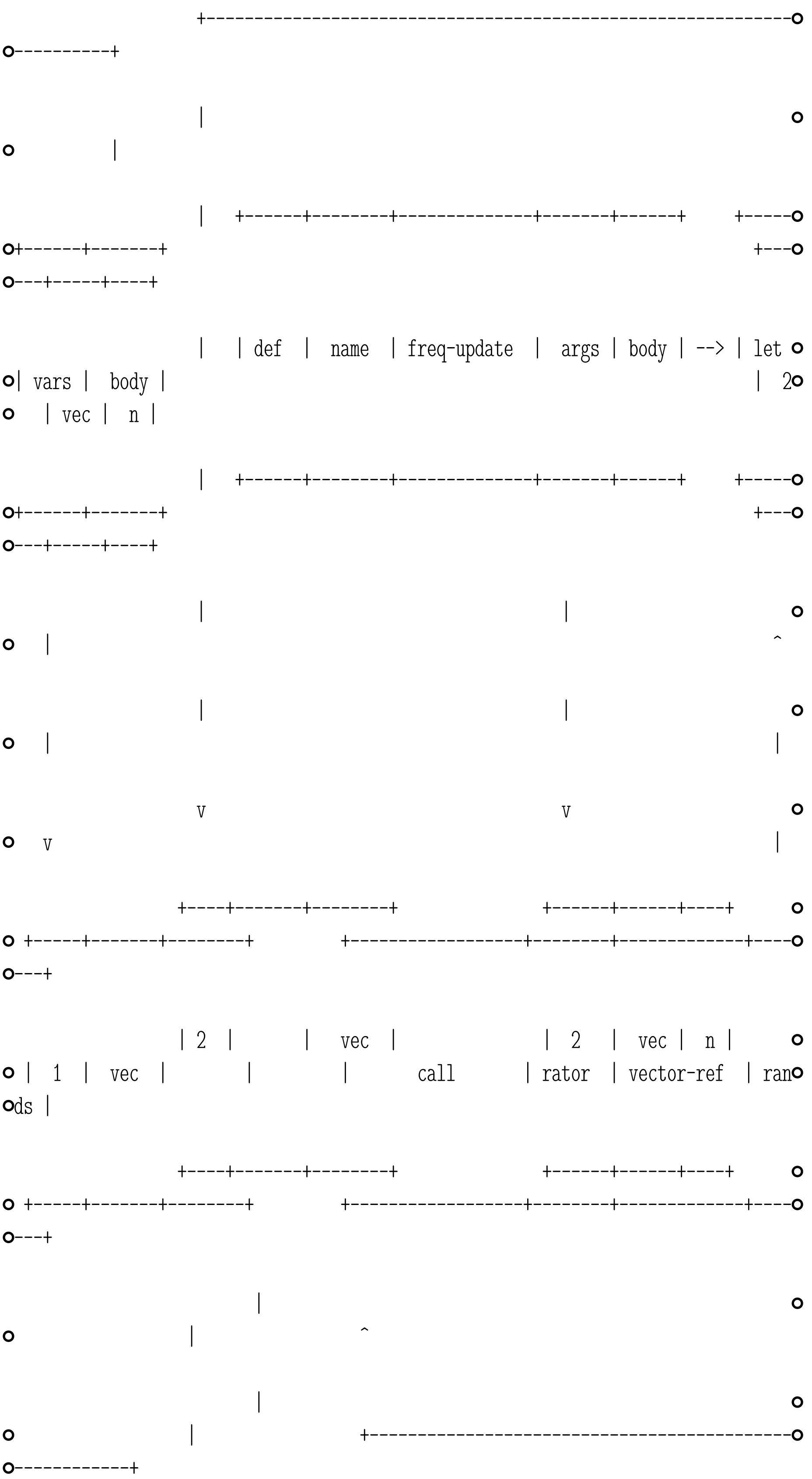
Here's a thing that might look like in Graphviz:

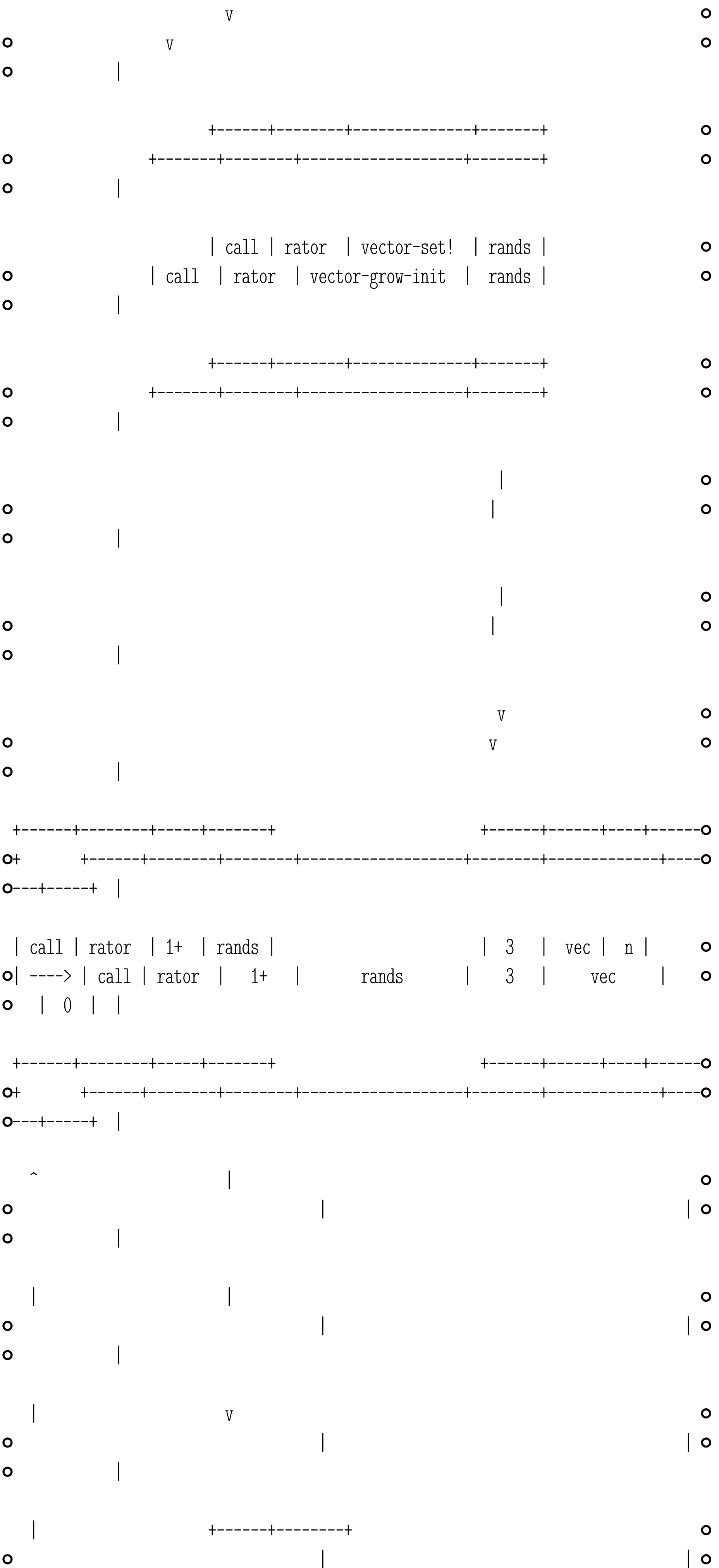



```
digraph x {
        node [shape=record]
        func [label="<head>def|{name|freq-update}|<args>args|<body>body"]
        func:args -> funcargs:head;
        funcargs [label="<head>2|vec|n"]
        func:body -> let:head;

        let [label="<head>let|<vars>vars|<body>body"];
        let:vars -> vars:head;
        vars [label="<head>1|{vec|<init1>}"];
        vars:init1 -> vecinit:head;
        vecinit [label="<head>call|{rator|vector-grow-init}|<rands>rands"];
        vecinit:rands -> vrands:head;
        vrands [label="<head>3|vec|<incn>|0"]
        vrands:incn -> incn:head;
        incn [label="<head>call|{rator|1+}|<rands>rands"];
        incn:rands -> incnrands:head;
        incnrands [label="<head>1|n"];

        let:body -> body:head;
        body [label="<head>2|<1>|vec"];
        body:1 -> x:head;
        x [label="<head>call|{rator|vector-set!}|<rands>rands"];
        x:rands -> y:head;
        y [label="<head>3|vec|n|<z>"];
        y:z -> z:head;
        z [label="<head>call|{rator|1+}|<rands>rands"];
        z:rands -> zz:head;
```

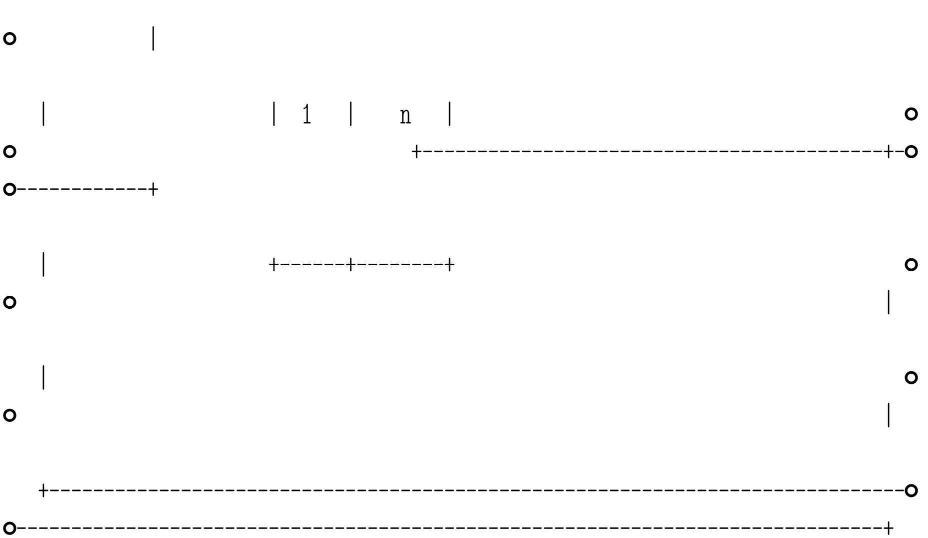
```
        zz [label="<head>call|{rator|vector-ref}|<rands>rands"];
        zz:rands-> zrands:head;
        zrands [label="<head>2|vec|n"];
}
```

It looks better rendered with Graphviz dot, but Graph::Easy manages a crude ASCII-art rendition which I've cleaned up here:





call	rator	vector-set!	rands
call	rator	vector-grow-init	rands
call	rator	1+	rands
3	vec	n
---->	call	rator	1+	rands	3	vec
0



Here “rator”, “rands”, “vars”, and “body” are just field labels; they don’t occupy space on their own behalf, but they’re associated with pointers which do. The type-tags “def”, “let”, and “call” might possibly be able to be erased to some degree at runtime, or possibly handled in a BIBOP fashion. The “vars” of “let” are special in that the list is a list of 2-tuples, one of them in this case. If the type-tags are not erased, by my count this is 44 words of memory, which is better than 56 but still not great.

If we can erase the type-tags (but not the vector lengths), that drops us by 7 to 37 words. Also, in most cases, the vectors could be tacked onto the end of their parent record, eliminating a pointer, although this might conflict with type-tag erasure (since they’d be allocated within the same page as their parent record) as well as mutability (since this would necessitate copying them if another such record wanted to “reference” them, which would also be a possible solution to the erasure problem); the exceptions are func:args and let:vars. This reduces the space cost to 31 words.

(I notice one error in this diagram: the vector-ref call is being referenced directly from a rands field rather than by way of a 1-element vector. This makes 44 46, 37 39, and 31 32.)

32 memory words is 43% less space than 56. The ASCII Scheme version, which is inconvenient for computation, is 133 bytes, while 32 32-bit words is slightly smaller at 128 bytes — not counting the print-names of the symbols, which are shared with all of their other occurrences.

I think this sort of thing, inspired by ML ADTs, is likely to actually be *more* convenient for programming than S-expressions, as well as more space-efficient.

BIBOP tagging for variant record tags

A possible BIBOP-alternative approach to handling type-tags, which still doesn’t require wasting an entire word in every object, is to store them on a separate page. Suppose we do embed the “rands” into call records, so their size depends on the number of arguments: two words if no arguments, three words if one argument, four words if two arguments, and so on; and suppose we use an allocator that allocates a page or subpage for each allocation size to prevent

fragmentation (which of course conflicts with the locality-of-reference objective...). A 2048-byte page of 4-byte words has 512 words on it; if it's divided into four-word objects, there will be 128 of them. Perhaps the memory allocator maintains a 128-bit bitmap of them elsewhere. Perhaps on a type-tag page there are 128 bytes that tell us the types of those 128 four-word objects: some may be two-argument calls, while others are single-expression-body lets, and others are single-expression-body defs. (This presumes that there are no more than 256 total type tags for four-word objects, and may lead to "read amplification" where we fault in an entire type-tag page even though 15 of the 16 pages it maps are swapped out.)

Probably a more reasonable approach is to dedicate pages to particular types of objects, such as calls, and subdivide them by size. So a 2048-byte page of calls might have 512 bytes (128 words, 64 objects) of two-word zero-argument calls; 512 bytes (42 objects) of three-word one-argument calls; 512 bytes (32 objects) of two-argument calls; and 512 bytes (25 objects) of three-argument calls. This allows you to minimize fragmentation and still do type-tests without faulting any pages in or limiting your number of types.

An immutable vector, or packed tuple, per list

A different memory representation, much closer to the Lisp approach, would be to just replace lists with vectors. You can see that the Scheme code contains 10 lists and 19 atom references, for a total of 28 conses; here's the code again:

```
(define (freq-update vec n)
  (let ((vec (vector-grow-init vec (1+ n) 0)))
    (vector-set! vec n (1+ (vector-ref vec n)))
    vec))
```

(Or you can calculate it with this dumb code:

```
(define (atom-count sexp)
  (cond ((null? sexp) 0)
        ((pair? sexp) (+ (atom-count (car sexp))
                         (atom-count (cdr sexp))))
        (#t 1)))

(define (cons-count sexp)
  (cond ((null? sexp) 0)
        ((pair? sexp) (+ 1
                         (cons-count (car sexp))
                         (cons-count (cdr sexp))))
        (#t 0)))

(define (nil-count sexp)
  (cond ((null? sexp) 1)
        ((pair? sexp) (+ (nil-count (car sexp))
                         (nil-count (cdr sexp))))
        (#t 0)))
```

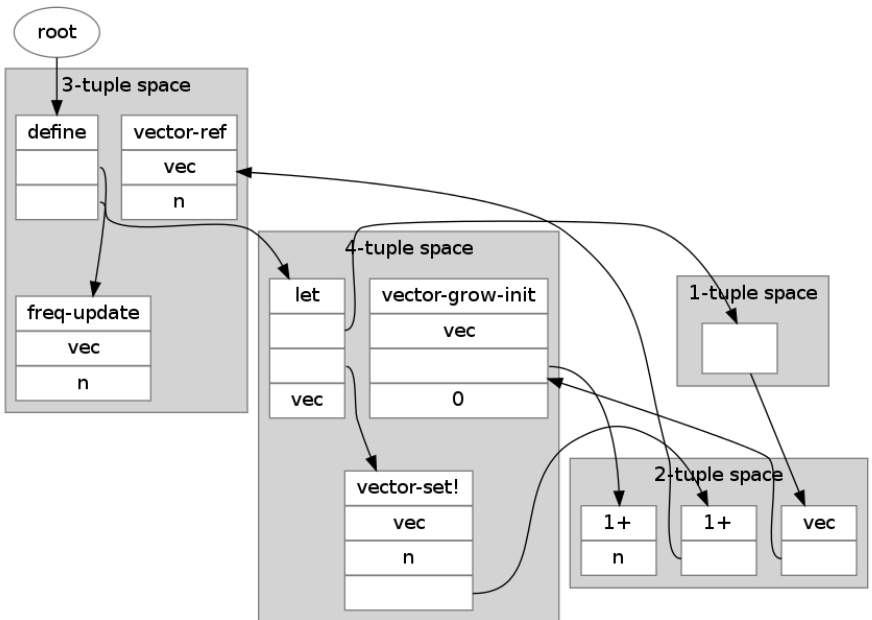
)

If you were to replace each list with a counted vector, these 28 conses in 10 lists would be only 38 words:

```
|---+-------------+-----+---+   +---+----+---|
| 3 | freq-update | vec | n |   | 2 | 1+ | n |
|---+-------------+-----+---+   +---+----+---|
```

And if you were to segregate these vectors or tuples into pages or subpages by size, you could then omit the count fields (except on vectors too long to belong to a whole category of same-sized vectors, which would still need counts). Then instead of 38 words they would be 28 words. Optimal! All we had to sacrifice was mutability and tail sharing.

A Graphviz visualization looks like this:




```
digraph fu {
        node [shape=record, style=filled, fillcolor=white, color="#7f7f7f"]
        graph [style=filled, fillcolor=lightgrey, color="#7f7f7f"]
        root [shape=oval];
        root -> 6;

        subgraph cluster_1 {
                label="1-tuple space";
                5 [label="<0>"];
        }

        subgraph cluster_2 {
                label="2-tuple space";
                2 [label="{1+|n}"];
                4 [label="{vec|<1>}"];
                9 [label="{1+|<1>}"];
        }
```

```
        subgraph cluster_3 {
                label="3-tuple space";
                1 [label="{freq-update|vec|n}"];
                6 [label="{define|<1>|<2>}"];
                10 [label="{vector-ref|vec|n}"];
        }

        subgraph cluster_4 {
                label="4-tuple space";
                3 [label="{vector-grow-init|vec|<2>|0}"];
                7 [label="{let|<1>|<2>|vec}"];
                8 [label="{vector-set!|vec|n|<3>}"];
        }

        3:2 -> 2;
        4:1 -> 3;
        5:0 -> 4;
        6:1 -> 1;
        6:2 -> 7;
        7:1 -> 5;
        7:2 -> 8;
        8:3 -> 9;
        9:1 -> 10;
}
```

An astonishing thing is that these 28 words are *smaller* than the 32 words taken up by the ML-like data structure outlined above, and under much less dubious assumptions.

This is clearly a broadly applicable data representation for things like syntax trees, binary search trees, and hash tables; and because you can do random access within the counted array it's a reasonable structure for things like binary search and (non-in-place, because immutable) quicksort as well.

Several approaches to efficiently CDRing down such vectors

First, we could use ordinary pointers into the vectors as the iterator states. Unsafe cdr would then just be a pointer increment, but the null test would then require decoding the pointer's bit representation and doing some kind of lookup to figure out what size the vector was, and maybe doing a modulo by the vector size (3 or 5 or something) to find the offset into the vector. This sounds like it would make cdring down a list spectacularly slow.

Second, we could use some kind of fat-pointer representation of iterators/ranges, like a base pointer and an offset, or an iterator pointer and an upper-bound pointer. CDR and NULL? are then fast but their argument no longer fits in a register; it needs two registers. (And, in the case of CDR, the result.) This has the additional advantage that you can efficiently refer to any range of a list, not just suffixes, and algorithms like binary search in the list also become more natural to express. With a Lua-like calling convention this isn't necessarily a big practical problem for programming but it does add complexity, and you need list→iter and perhaps iter→list functions of some kind. list→iter in particular has to figure out how big the list is,

through the kinds of pointer decoding and lookup that NULL? would have had to do in the previous option, but fortunately without the division operation.

Third, we could try to cram the fat-pointer representation into a single word somehow. For example, we could 8-byte-align all our vector pointers and use the three low-order bits to indicate an offset from 0 to 7, and chain together vectors to make lists of more than 8 items.

Fourth, we could maybe burst the vector into a conventional pile of dotted pairs when we start iterating over it, in some kind of very cheap garbage collection nursery or something. This obviously sacrifices mutability, which we already sort of did, but also damages EQ on lists, and it adds to the load on the GC.

Fifth, we could use a stateful generator coroutine, which is in some sense another version of “fat pointers”.

Incrementing a pointer in a loop, testing it against an end pointer, to do a linear search for a key, looks like this in reformatted GCC listing output for amd64, with the loop preamble removed:

```
  25 0010 4883C708      .L8:    addq    $8, %rdi        # increment pointer

  27 0014 4839FE                cmpq    %rdi, %rsi      # compare against end pointer

  28 0017 7405                  je      .L2             # (fused with previous) exit loop

  31 0019 483907        .L6:    cmpq    %rax, (%rdi)    # compare against search key in %rax
  32 001c 75F2                  jne     .L8             # loop while not found
  33                    .L2:
```

So it’s about 3–5 RISC instructions or micro-ops — and *zero* data memory accesses! — per loop iteration.

A context switch to a stateful generator coroutine can be as simple as an indirect jump or call, followed by another indirect jump or return later; but more generally you should expect to pay nearly the usual procedure entry/exit cost, because a lower cost would imply partitioning the register file between the generator and the consumer. Whatever registers the consumer clobbers aren’t available to maintain generator state, and whatever registers the generator clobbers aren’t available to maintain consumer state. The generator can keep its stack frame around from one yield to the next, but it still has to save any relevant callee-saved registers upon resuming and restore them on yielding, and vice versa for caller-saved registers.

Not CDRing down packed vectors

Many lists — though few in the above example — are used as fixed-arity tuples rather than variable-length lists, and for these lists we’re probably most interested in pattern-matching them against patterns of the same arity, rather than iterating over their members. This is clearly much more efficient to do with these packed tuples

than with linked lists of dotted pairs.

Building such packed-tuple vectors

If you knew how many things are going to be in your output list, you could maybe preallocate it. MAPCAR, for example, could do this.

But the usual kind of Lisp code doesn't know in advance how big a list it's building, and knowing would make it more complicated. Consider this example (part of my solution to an exercise from Essentials of Programming Languages):

```
(define (set-subtract includes excludes)
  (cond ((null? includes) '())
        ((member (car includes) excludes) (set-subtract (cdr includes) excludes))
        (#t (cons (car includes) (set-subtract (cdr includes) excludes)))))
```

You could quibble with the crude coding style and the O(MN) algorithm, but I think it's fair to say that writing recursive definitions of that sort and having them run with reasonable efficiency is a big part of the appeal of Lisp. But how do you write such a recurrence in this natural way, without losing the space-efficiency benefits of the packed vector representation?

One thing you could of course do is have a function that converts explicitly from the unpacked chain-of-dotted-pairs form into the packed form, and invoke it after the final recursive call:

```
(define (set-subtract includes excludes)
  (list->packed
        (let recur ((includes includes))
          (cond ((null? includes) '())
                ((memb (ca includes) excludes) (recur (cd includes)))
                (#t (cons (ca includes) (recur (cd includes))))))))
```

Exposing the difference to the user in this way would eliminate the necessity for the ordinary functions `car` and at least `cdr` and so on to handle both vector iterators (whatever their form) and dotted-pair iterators, which might make them run faster. (However, if they're doing a run-time safety check, I think the extra cost to fall back to a more general dispatch when the expected-type check fails is probably insignificant.)

Alternatively, you could let the garbage collector take care of it: when it copies a linked list of immutable dotted pairs out of the nursery, it can count its length and copy it as a packed tuple into the appropriate packed tuple bucket, instead of copying the dotted pairs individually. This does require `cdr` to check which representation is in use at any given moment, though. And sometimes it's important for both space and time to preserve some tail sharing (though you can always restructure algorithms to do their sharing via the `car`).

Such approaches were inconceivable before the invention of generational garbage collection in the mid-01980s.

(It might even be worthwhile to include the list length in the

dotted pairs when you create them; although this fills up the nursery faster, it means the GC doesn't have to pretraverse the list to count its length before packing it.)

As a third alternative you might consider explicit *mutability*:

```
def set_subtract(includes, excludes):
    result = []
    for item in includes:
        if item not in excludes:
            result.append(item)

    return result
```

But this doesn't really help with the problem at hand; CPython has to preallocate a bunch of extra slack space for `result`, probably several times, and the last time most of it never gets used. If you say `return tuple(result)` you eliminate this space waste at the cost of an extra copy, similar to the `list->packed` approach.

It sure is a lot less fucking code than the Scheme, though, isn't it?

In this particular case an additional angle on the problem is to consider `set-subtract` as a potentially lazy sequence transformer, usable for example as a generator coroutine:

```
def set_subtract(includes, excludes):
    for item in includes:
        if item not in excludes:
            yield item
```

If you want to materialize it, though, that still doesn't help; the consumer of the generated sequence can do whatever they want with it, including packing it into a vector, but *they* don't know in advance how big it is either!

Locality through duplication

The packed-tuple representations above eliminate tail sharing by duplicating the shared list tails, which is valid for immutable structures; the only question is whether it is more or less wasteful than representing the lists with materialized dotted pairs (in the words of "A Command Structure for Complex Information Processing" from 01958:

> A list structure can be established in computer memory by associating with each word in memory an address that determines what word is adjacent to it, so far as all the operations of the computer are concerned. We pay the price in memory space of an additional address associated with each word, so that we can change the adjacency relation as quickly as we can change a word of memory.

One of the benefits of copying these shared tails is that it ensures that each list is on the same memory page as its tail, so no additional page faults are incurred in walking down the list. But this is not the only relevant locality criterion; we would also like the contents of the list to be on the same page, as BBN LISP's allocation strategy attempts (see above about "CDR-coding"). With immutable list

contents and the gigantic resources of NAND, we can achieve this to a significant extent by peremptorily copying immutable referents when we construct lists.

Compressed oops

So far everything described has been described in terms of "words", but I think the "Large Object-Oriented Memory" approach may be suitable for this kind of microcontroller system. Load and store instructions to internal SRAM on ARM Cortex-M3 microcontrollers commonly take 2 clock cycles, though sometimes this can be pipelined, so I'm guessing this would be 28 ns on the STM32F103C8 at 72 MHz, or 7 ns per byte. This is around 8–32 times faster than the access times to any of the other memories considered above, and consumes almost proportionally less energy, so avoiding off-chip access as much as possible would seem to be paramount.

The LOOM Smalltalk paper used 32-bit ordinary object pointers ("oops") on disk, but 16-bit "compressed oops" in RAM. An in-RAM object table kept track of the correspondence. Such an approach could probably nearly double the effective capacity of the 20-KiB RAM on such a microcontroller. 20 KiB is enough to hold 20480 bytes, 10240 16-bit compressed oops, or 5120 32-bit oops or two-element vectors of 16-bit compressed oops. Moreover, for the NAND Flash sizes I was talking about above, you might want 40-bit or 48-bit uncompressed oops!

Even the largest low-power microcontrollers considered above have only 96 KiB of on-chip SRAM: 98304 bytes, 49152 compressed oops, 24576 dotted pairs or two-element vectors made of compressed oops, or 12288 32-bit oops. So clearly a 16-bit field is enough to address all the objects that will fit into SRAM at a time.

Compare these to the 4096 symbols and 00140_000 = 49152 words devoted to compiled code and list structure in Bobrow's paper: that's also 24576 dotted pairs! But shared only with compiled code, not with symbols and integers and whatnot. We can take some comfort in being able to page things in 6000 times a second instead of 30 times a second — but in proportion to CPU speed this swapping rate is actually much slower. Bobrow's 17-ms average rotational latency was only about 1000–2000 machine instructions, while 150 μs is about *ten* thousand instructions for our CPU. However, we can expect even much faster performance than that: Bobrow was only getting 6000 subroutine calls per second, about 15 instructions per call, which he attributes to the weaknesses of the PDP-1's instruction set, but we ought to be able to get something like two or three times that amount.

LOOM supposedly paged objects in and out one at a time, but I can't imagine that working very well for things like disk or NAND. Evicting individual objects would be fine for things like external SRAM, and faulting *in* individual objects would be fine for things like NOR.

Compressed oops in secondary storage

You might also be able to use the compressed-oop approach

advantageously within memory pages in NAND storage. With 40-bit uncompressed oops, a 2048-byte NAND page can contain only 409 pointers, or 204 dotted pairs, enough for a list of 204 things, or a counted vector of 408 things. Suppose that we allocate 1024 bytes of the page to a "link table" of 204 uncompressed oops, and the other 1024 bytes to 682 12-bit ultracompressed oops, organized into up to 682 tuples of various sizes, plus a little metadata. The 4096 slots in the 12-bit address space for these ultracompressed oops are divided among the 204 external pointers, the 682 or less internal objects, small integers, other popular objects like nil, and perhaps some objects cataloged on other "indirect pages". We can still make a list of 204 external things, occupying a counted packed tuple in 206 of these 1024 bytes, but we can also represent quite complex data structures within this space. In a near-typical case where it's entirely allocated to 3-item packed tuples mostly pointing within the page, the 409 uncompressed oops would have bought us 136 of them, but this compressed layout buys us 227 of them, a 67% increase in capacity, or from another point of view, a 40% decrease in the number of pages required. This 40% reduction in bulk reduces the number of page faults to fault in a given set of objects by some amount less than 40%, but that depends on the locality with which the objects are allocated: in the extreme case of no locality of reference, you still have one page fault per object, but in the extreme case of purely sequential access, you'd have a 40% reduction in page faults.

(Weirder representations are plausible too, like variable-size link tables and 10-bit ultracompressed oops; you could imagine, say, a page with 512 3-item packed tuples on it, packed 10 bits per ultracompressed oop, and a table of 25 uncompressed oops.)

Remember that faulting in this 2048-byte NAND page costs us around 10 μJ, which is worth around 7000 instructions at 1.5 nJ per instruction. We also another 10 nJ or so to write it back if we dirty it. So decreasing the number of pages needed by 40%, and the number of page *faults* by, say, 20%, is worth spending a fair number of instructions on decompression, though only 5 instructions or so per object.

In the microcontroller's SRAM, it isn't useful to try to maintain this ultracompressed representation, because the expense of maintaining three copies of each oop (uncompressed, ultracompressed, and 16-bit compressed) is greater than the savings from the ultracompressed version; moreover, the uncompressed oops are likely to be duplicated between pages in the working set.

By promiscuously duplicating immutable list structure onto such ultracompressed pages, we can both conserve their uncompressed-oop slots and improve locality of reference further; probably this is a very difficult problem to solve optimally, but admits robust heuristics that very frequently do much better than a totally naïve algorithm. Greedy depth-first packing, for example. This potentially also permits glomming objects together into a page in whatever order is convenient for paging things out, rather than along with whatever objects they previously shared a page with, thus perhaps improving the write-amplification problem common to all paging systems.

One variant of such an ultracompressed page containing the

`freq-update` definition used above as an example might look like this in hex, using 8-bit indices instead of 12-bit for ease of reading and understanding:

```
09 00               ; link table of 9 40-bit uncompressed oops
01 01 00            ; followed by one 1-tuple, starting at index 9
02 03 00            ; three 2-tuples, starting at index 0a
03 03 00            ; three 3-tuples, starting at index 0d
04 03 00            ; three 4-tuples, starting at index 10
00                  ; and nothing else
f0 ad 55 27 ff      ; uncompressed oop for symbol `define` (index 0)
20 ae 55 27 ff      ; uncompressed oop for symbol `freq-update`
50 ae 55 27 ff      ; vec
80 ae 55 27 ff      ; n
b0 ae 55 27 ff      ; vector-ref, index 4
e0 ae 55 27 ff      ; let
80 b0 55 27 ff      ; vector-grow-init
20 b1 55 27 ff      ; vector-set!
b0 b1 55 27 ff      ; 1+, index 8
0a                  ; 1-tuple at index 9 refers to first 2-tuple
02 10               ; 2-tuple at index 0a refers to symbol vec and first 4-tuple
08 0d               ; 1+ and first 3-tuple
08 03               ; 1+ n, index 0c
04 02 03            ; first 3-tuple: vector-ref vec n
00 0f 11            ; second 3-tuple: define <last 3-tuple> <second 4-tuple>
01 02 03            ; last 3-tuple: freq-update vec n

06 02 0c f0         ; first 4-tuple: vector-grow-init vec <third 2-tuple> 0, index 10
05 09 12 02         ; let <1-tuple> <last 4-tuple> vec
07 02 03 0b         ; vector-set! vec n <second 2-tuple>
```

If I haven't screwed this up, this is a 15-byte header, a 45-byte link table of uncompressed oops, and 28 bytes of list structure, which would be 42 bytes in the 12-bit representation suggested above. 15+45+42 = 102 bytes, about 3.6 bytes per oop or per cons, 10 bytes per list, a number which should go down as the number of oops in a page rises from 28 up to 300–600. In SRAM the list structure might occupy 56 bytes of 16-bit compressed oops, if we're content to forget about the origin of each packed tuple, but if not, we might need an additional 50-byte-or-more chunk of the in-SRAM uncompressed oop table to remember where each of these 10 vectors came from.

This last consideration suggests that such compressed pages should list not only *outgoing* references in a link table, but also "labels" for *incoming* references. The idea is that labeled objects in the compressed page would be assigned an uncompressed oop so they could be referenced from outside the page, while unlabeled objects (necessarily immutable) can only be copied, not referenced. This might also allow us to use multi-gigabyte Flash effectively with uncompressed oops weighing only 32 bits rather than 40.

Consider a 256-gibibyte SD card, containing 2^{38} bytes, which is currently near the ceiling for what's easily available as an SD card (though 512 gibibytes isn't unheard of, even 256 gibibytes costs

US$40, 32 gibibytes is more common at around US$5, and 16 gibibytes is now under US$3 at retail). If our uncompressed oops were only 32 bits, the average object size would have to be 64 bytes, which is too big for cons cells or 3-tuples. If it's divided into 2048-byte pages, it has 2^{27} pages; 32-bit uncompressed oops would give us 32 labels per page. (You might want some kind of extra layer of indirection somewhere so that the oop numbers don't directly give you physical block numbers, though). It's easily plausible that you could get down to 32 externally-referenced atoms or chunks of list structure per 2048-byte page.

Bytecode

Bobrow doesn't mention bytecode at all. Given the killer advantages of bytecode for what he's doing, this surprised me, but apparently compiling to bytecode interpreters started with Wirth and Weber's EULER in 01965, where it was presented not as a hack to reduce memory usage but an alternative to the λ-calculus for formally defining programming-language semantics, so it wasn't a well-known technique at the time (p. 56):

> The definition of the *compiling* system consists of the parsing algorithm, given in paragraph III.B.1., a set of syntactic rules, and a set of corresponding interpretation rules by which an EULER text is transformed into a polish string. The definition of the *executing* system consists of a basic interpreting mechanism with a rule to interpret each symbol in the polish string.

Where introduced, the "bytecode" in question didn't have a defined representation as a sequence of bits or numbers, so it's maybe okay to call it "p-code" but not than "bytecode"; however, the appendices of the TR do include a full program listing in Burroughs B5500 Extended Algol, evidently written by Wirth and Bill McKeeman† in 01964 and 01965. Stanford's scan is badly corrupted, and I'm not sure my understanding is entirely correct, but it doesn't seem to try to pack operands into bytecode bytes, or indeed even pack operators into single bytes. (The family resemblance to the current Oberon compiler is remarkably close!) I may be misunderstanding something but it looks like the "bytecode" was actually an array of records named `PROGRAM` with record fields named `AFIELD`, `BFIELD` and `CFIELD`. `BFIELD` was used for "immediate operands" for loading numbers and symbols onto the stack. These seem to be defined as follows, making me think they may be bitfields in the B5500's 48-bit words:

```
DEFINE AFIELD=[39:9]#,BFIELD=[9:30]#,CFIELD=[1:8]#;
```

So, as implemented, it wasn't so much a bytecode as a "wordcode".

However, Zork, old versions of Multiplan and Excel (especially on the Macintosh), GNU Emacs, and Smalltalk systems get a lot of mileage out of bytecode virtual machines, which allow them to squeeze a lot more code into very small computers than you would think possible. This command from my .emacs is 16 lines of code, but it compiles to 44 bytes of Elisp bytecode, plus some tables of constants and external references (literals, symbols):

```
(defun markdown-tt-word ()

  "Hit N times to enclose previous N chunks of nonwhitespace in `` (for Markdown)⚬
⚬."
  (interactive)
  (if (looking-back "`")
      (save-excursion
        (backward-char)

        (search-backward "`")
        (delete-char 1)

;; (backward-word) previously
        (search-backward-regexp "\\S-")
        (search-backward-regexp "\\s-")
        (forward-char)

        (insert "`"))
    (progn
      (insert "`")
      (save-excursion
        (backward-word) ; should this use the same simpler approach?
        (insert "`")))))
```

I think it's atypically dense to get just 3 bytes of bytecode per line of code; 6 bytes per line is probably closer.

Here's a disassembly of the 44 bytes. You can see that they use a stack bytecode with a single operand — usually packed into the same byte, much like Smalltalk bytecode, but in the case of `goto-if-nil` it seems to have a two-byte immediate operand following the bytecode. The operands packed into `constant` and `varref` bytecodes are indexes into pools of such things, again as in Smalltalk. Specific opcodes are allocated to popular Emacs operations like `save-excursion`, `forward-char`, and `forward-word`.

```
byte code for markdown-tt-word:
  args: nil
 interactive: nil
0       constant  looking-back
1       constant  "`"
2       call      1
3       goto-if-nil 1
6       save-excursion
7       constant  -1
8       forward-char
9       discard
10      constant  search-backward
11      constant  "`"
12      call      1
13      discard
14      constant  delete-char
15      constant  1
16      call      1
```

```
17      discard
18      constant  search-backward-regexp
19      constant  "\\S-"
20      call      1
21      discard
22      constant  search-backward-regexp
23      constant  "\\s-"
24      call      1
25      discard
26      constant  nil
27      forward-char
28      discard
29      constant  "`"
30      insert
31      unbind    1
32      return
33:1    constant  "`"
34      insert
35      discard
36      save-excursion
37      constant  -1
38      forward-word
39      discard
40      constant  "`"
41      insert
42      unbind    1
43      return
```

Let's see how our example code from earlier fares:

```
(define (freq-update vec n)
  (let ((vec (vector-grow-init vec (1+ n) 0)))
    (vector-set! vec n (1+ (vector-ref vec n)))
    vec))
```

In Elisp that's spelled:

```
(defun freq-update (vec n)
  (let ((vec (vector-grow-init vec (1+ n) 0)))
    (aset vec n (1+ (aref vec n)))
    vec))
```

Emacs compiles the code part of this to 18 bytes of bytecode, 4½ per line; this is a lot less than 28 compressed oops (56 bytes).

```
byte code for freq-update:
  args: (vec n)
0       constant  vector-grow-init
1       varref    vec
2       varref    n
3       add1
```

```
4       constant  0
5       call      3
6       dup
7       varbind   vec
8       varref    n
9       varref    vec
10      varref    n
11      aref
12      add1
13      aset
14      discard
15      varref    vec
16      unbind    1
17      return
```

It's a little bit unfair not to count the argument list and constants vector here, because they allow the references to `vec` and `n` and `vector-grow-init` to fit into a few bits of a byte, while normally they would require 8 bytes, as the elements of the argument list and constants vector do. Only a little unfair, though.

It is surely not at all coincidental that the ultracompressed-oop stuff in the previous section looks very much like this bytecode structure. The argument list and constants vector are very similar to the "link table" for a page, but the link table is shared among the several hundred ultracompressed oops on that page, rather than just the several dozen bytecodes of an Emacs function.

There exists previous work on this. PICBIT was a bytecoded Scheme for PIC microcontrollers, but its bytecode was an order of magnitude fatter than the 6 bytes per line of code I suggested above; it was a successor to the slimmer BIT, which fit the R4RS Scheme library into under 8000 bytes of bytecode.

Immutable list structure can be productively represented by a sort of bytecode as well, but this probably isn't worthwhile. Given the same "link table" vector of 9 atoms as above in the section on "compressed oops in secondary storage", in Elisp notation `[define freq-update vec n vector-ref let vector-grow-init vector-set! 1+]`, we could represent our example Scheme definition as the 39-byte string "(0(123)(5((2(62(83)@)))(723(8(423)))2))". Here "(" is the "nest" bytecode, ")" is the "nil" bytecode which terminates a list, the ASCII digits are indices into the link table, and "@" is the small-integer 0; pointers into the list structure are represented by pointers before its bytes. `null?` is straightforward (is a pointer pointing before a ")"?) and `car` either decodes the byte following the pointer or, in the case of "(", returns an incremented pointer, which is the same distinction `atom?` must make. Only `cdr` is tricky, since it must increment the pointer past either an atom or an arbitrary-length list. A table of minimal nesting depth per 8-byte chunk would probably make this adequately efficient by enabling `cdr` to rapidly skip over large, deeply-nested chunks:

```
(0(123)(  0
5((2(62(  2
```

```
83)@)))(  2
723(8(42  2
3)))2))   1
```

You could assign better bytes, but this is probably not worthwhile. Not counting the link table, this bytecoded list structure uses 39 bytes (as opposed to the 18 bytecode bytes for the compiled code), 44 including the skip table, while in its packed-tuple form it occupies only 28 oops: 56 bytes in the 16-bit case, 42 bytes in the 12-bit case. If we were to use 8-bit compressed oops (as the above bytecode implicitly does), limiting the bytecode to only 256 referents, they would be 28 bytes, which is less than 39. Also, the packed tuples are more efficient to traverse and can support mutation.

It probably *is* worthwhile to burn large amounts of bytecode into the microcontroller's Flash and interpret it from Flash; that way you don't have to spend precious RAM on code. (128 KiB would hold about 20,000 lines of high-level code by the above estimate — the whole VPRI STEPS complexity budget — but of course the system has to use some memory too.) It's probably okay for loading a new application to take a good fraction of a second if that frees up RAM for data. Most modern microcontrollers support pagewise reprogramming of their internal (NOR, painfully slow) Flash during operation, so this kind of thing should be acceptable even for exploratory interactive development.

† McKeeman also evidently invented peephole optimization in 01965, differential testing in 01998, and was Hehner's advisor's advisor, but he wasn't credited in the TR.

JIT compilation

In a memory-constrained environment, it's surely worthwhile to compile all code to bytecode like the above, rather than interpreting directly from dotted pairs as BBN LISP did (or from packed tuples). Compiling into machine code is a more dubious proposition, since machine code tends to be much bulkier than bytecode, but the typical interpretive slowdown is a factor of 10–40, which in this case could translate to 10–40 times shorter battery life for compute-bound tasks.

Ur-Scheme uses a very simple and dumb compilation scheme (the same one as Bigforth, the one Crenshaw uses in "Let's Make a Compiler") for expressions, and open-codes the common cases of built-in operations, preceded by a dynamic type check (typically provided by "millicode", though I didn't know the name at the time). The consequence is that it compiles its own 1553 lines of Scheme into 90168 bytes of i386 code (plus 45K of data, mostly read-only), almost 60 bytes of machine code per source line, doing things like this:

```
call *%ebx
pop %eax
push %eax
movl $2 + 256<<2, %eax   # Ur-Scheme's representation of nil
.section .rodata
# align pointers so they end in binary 00
```

```
        .align 4
_parse_eofP_9:        # in-memory read-only representation of string
        .long 0xbabb1e
        .long 3
        .ascii " ()"
        .text
        push %eax
        movl $_parse_eofP_9, %eax
        # get procedure
        push %eax
        movl (_parse_string_1), %eax
        # apply procedure
        call ensure_procedure
        movl 4(%eax), %ebx
        movl $1, %edx                               # argument count
        call *%ebx
        # get procedure
        push %eax
        movl (_emit_malloc_n_4), %eax
```

This compiles to something like the following:

```
805d9f4:      ff d3                    call   *%ebx
805d9f6:      58                       pop    %eax
805d9f7:      50                       push   %eax
805d9f8:      b8 02 04 00 00           mov    $0x402,%eax
805d9fd:      50                       push   %eax
805d9fe:      b8 48 2f 06 08           mov    $0x8062f48,%eax
805da03:      50                       push   %eax
805da04:      a1 74 95 06 08           mov    0x8069574,%eax
805da09:      e8 58 a7 fe ff           call   8048166 <ensure_procedure>
805da0e:      8b 58 04                 mov    0x4(%eax),%ebx
805da11:      ba 01 00 00 00           mov    $0x1,%edx
805da16:      ff d3                    call   *%ebx
805da18:      50                       push   %eax
805da19:      a1 24 94 06 08           mov    0x8069424,%eax
```

These 40 bytes of machine code in 14 instructions (one invoking a millicode routine) are *part of* the compilation results from this line of code, which invokes no macros:

```
(assert-equal (parse-string " ()") '())
```

It's easy to do a little better than this with peephole optimization, but I think doing much better requires not just register allocation (at least in an extremely stupid form) but also some kind of static type system so we can avoid the endless type tests. I mean you could imagine the above getting compiled to the following:

```
    movl $_parse_eofP_9, %eax   # load string pointer
    call _parse_string_2
```

```
    xor %ecx, %ecx              # load nil
```

And that would be 12 bytes instead of 40, and 3 instructions instead of 14 (plus the 5 instructions in `ensure_procedure`). But for that you'd have to know statically that parse-string was a procedure (and not a closure or, say, an integer) and that it took one argument. To avoid additional redundant checks inside of it, you'd also need to know that that argument was a string.

By contrast, the source code would be four oops of list structure, and it compiles to 4 bytes of Elisp bytecode out of these 7:

```
byte code for test-assert-equal-parse-string:
  args: nil
0       constant  assert-equal
1       constant  parse-string
2       constant  " ()"
3       call      1
4       constant  nil
5       call      2
6       return
```

As a very crude sort of estimate:

form	weight
unoptimized i386 code	40 bytes
source text	25 bytes
dotted pairs, 16 bit	16 bytes
optimized i386 code	12 bytes
packed tuples, 16 bit	8 bytes
packed tuples, 12 bit	6 bytes
bytecode	4 bytes (plus constant vectors, etc.)

I haven't tried it on ARM or RISC-V but I think the results will probably be about the same as i386, maybe a bit smaller. So we should expect easily generated machine code to be around 4 times the size of the list-structure source code, reasonable machine code to be about the same size as the source code, and bytecode to be smaller than that by a factor of 2–4.

When compiled with itself, Ur-Scheme, despite the slowdown factor of 4–8 evident in the above code, executes only 678,737,113 userland instructions to compile itself a second time (180 ms, only ⅔ in userland), about 437 instructions per line of source code. The executable contains 29355 instructions, so generating each output instruction takes about 23 instructions. Presumably it would be about 30 kilobytes of code and 8000 instructions, and need to run about 128 instructions per source line, if it were compiled with a decent compiler — but also generate a smaller amount of output code, so the 23-instruction ratio stays the same. *Being* such a decent compiler might slow it back down to 512 instructions per source line or, say, 128 instructions per output instruction. (Then again, it might speed it up.)

This suggests that, when you can execute from RAM, the iteration count at which a JIT compilation pays off timewise is on the order of

4–32. If you have a straight-line chunk of bytecode that can be JIT-compiled to a sequence of 32 instructions, then doing that compilation (at 128 compiler instructions per output instruction) might cost you 4096 instructions. Each time you interpret the bytecode at an interpretation slowdown of 16×, you spend 512 instructions on interpretation, so after 8 repetitions you've wasted the 4096 instructions you could have used on compiling it. If it's possible to cut the compilation time down to 24 compiler instructions executed per compiled instruction output, then the breakeven point is 1½ iterations!

So, particularly if decent static type information is available, you'd get a pretty good reduction in battery usage by compiling most bytecode to machine code, even if you had to discard the compiled machine code after a small number of executions. It might even be worthwhile to *always* JIT-compile the bytecode. This approach, in turn, reduces the pressure on bytecode to be fast to unpack, so you can use bytecode representations that require more computation to understand, enabling you to fit more bytecode in RAM at once. For example, you could use variable-length instruction and operand encodings.

Compiled machine code that is executed many times is a good candidate for promotion to the microcontroller's flash to free up RAM. Since this is NOR, we can expect to pay on the order of 2000 nJ per byte to erase and program it, so "many times" means something on the order of $4096 = 2^{12}$ times — but perhaps it should also outweigh whatever compiled code must be evicted to make room for it, since we can only erase and reprogram Flash pages $100k \approx 2^{17}$ times or so. Otherwise we are going to wear out a Flash that can hold 2^{16} instructions after something like $2^{12+17+16} = 2^{45}$ instruction executions. That's about an hour.

Topics

- Programming (p. 1141) (49 notes)
- History (p. 1153) (24 notes)
- Performance (p. 1155) (22 notes)
- Safe programming languages (p. 1172) (11 notes)
- Lisp (p. 1174) (11 notes)
- Virtual machines (p. 1182) (9 notes)
- Microcontrollers (p. 1211) (6 notes)
- Bytecode (p. 1236) (5 notes)
- Reverse Polish notation (RPN) (p. 1243) (4 notes)
- Memory models (p. 1285) (3 notes)

Panelization in PCB manufacturing

Kragen Javier Sitaker, 02021-02-25 (updated 02021-02-26) (7 minutes)

PCB contract manufacturers like JLCPCB (“CMs”) can not only cut out your circuit board to specified shapes with a router (2-mm endmill), but also score it with V-grooves on one or both sides, so you can break it into panels. Looking at the Boréas Technology evaluation board, I see that it’s scored this way on one side to be able to break it into three boards, but has traces across the scoring on the opposite side of the board, so *until* you *do* break it into panels this way, it’s a single board; but once you do, you can reconnect the previously-connected pieces at a distance using a cable, or use them separately. Among other things, this approach is useful for testability.

Other forms of panelization include routing that leaves tabs (“tab-routing”) and drilling a series of holes, called “mouse bites,” most often used to weaken a tab, avoiding the need to score it with a knife, cut with diagonal cutters, or use a depaneling nibbler. Mouse bites in particular can create curved breakaway edges, and they leave a rough edge on the board after breaking, which can be useful or harmful. JLCPCB in particular is willing to do some amount of tab-routing without increasing their price of US$2 for 10 boards of 100 mm × 100 mm. There are established panelization guidelines.

SparkFun has a whole “ProtoSnap” product line based on this idea; their ProtoSnap Pro Mini, for example, had an Arduino Pro Mini SBC already wired up to a USB-to-serial converter, two actuators, two sensors, and a prototyping area, and their LilyPad ProtoSnap Plus has a LilyPad similarly preconnected to several sensors and actuators and conductive-thread connectors, which can then be broken apart at the tabs and reconnected elsewhere, like in your clothes.

LED tape often comes with similar cut points, where you can cut it with scissors to a given length, but if you don’t, the whole tape can be hooked up to from 2–4 wires. (Such tapes might cost US$19 for 5 m, US$43 for 5 m, or US$17 for 4½ m at retail here in Argentina.) “Flat flex PCBs” are printed on thin Kapton and commonly used as connector cables, and can be cut with scissors in a similar fashion.

And, of course, prototyping perfboard is pretty easy to cut along the perforation lines.

There’s also a technique to allow electrical continuity across V-scores cut on both sides of a board: a plated-through hole that the V-score runs through, so the plating in the hole electrically bridges the V-score.

Circuit board “edge connectors” are interesting to mention in this connection: ISA cards, PCI cards, DIMMs, and many USB-A devices have no separate connector, just exposed copper, either on one side of the board or both. (Zebra-strip, Z-tape, and pogo-pin “connectors” on PCBs are also commonly just exposed copper.) Such connectors can be provided at a place that’s exposed by snapping at

such V-grooves.

You could imagine designing a circuit board that works as a whole, but can also be broken into smaller circuit boards, or cut with scissors, to get more reconfigurability.

Custom PCB manufacturing is amazingly cheap; TrickyNekro reports €10 for 100 boards of 38 mm × 18 mm each, plus normally US$20 for "shipping". And if the panel size is *exactly* 100 mm × 100 mm, they used to not charge *anything* for V-scoring (though reportedly that was only if all the boards in the panel were identical, and they now charge something like US$8 for V-scoring, while others report problems getting JLC to V-score), although I don't know if there was a limit to how many lines you could request on your "board outline layer (*.GKO)". In the same thread, georges80 reports 3–4 day turnaround to the US West Coast, paying:

- US$20 for "the protos" (???)
- US$32 for "10 pieces with 3 [2-layer] boards on a panel with 'jlpcb' panelization which is v-scored"
- US$68 for 25 165 mm × 75 mm panels of 10 [2-layer] boards each, including paying extra white solder mask

And blazini36 reports getting 10 100 mm × 75 mm prototyping boards for US$27, including shipping and an extra US$8 for the boards being blue, in 3 days, on several occasions, while OSH Park wanted to charge them US$300 and delay 12 days. Battlecoder reports that they paid US$12 for 10 prototype boards, and had to wait a month, because JLCPCB doesn't have free shipping to their (unspecified) country.

Nobody in the thread is sure if they can do V-scores at arbitrary angles as well as straight across.

It occurs to me that this sort of thing might also be useful for *mechanical* construction, although the shapes you can V-score are even more limited than the shapes you can laser-cut — no sharp curves or inside corners. JLCPCB uses a cutting wheel to cut the V-scores so you can't even do curves. But if you can get some slots routed out in addition to the V-scores, you could get a pretty productive "construction set" pretty cheap. (You'd have to wear gloves to keep the fiberglass out of your fingers.) And then you can solder the joints to hold the assembled pieces in place, like the FR4 mill.

If you could get a 100-mm-square panel panelized with V-scores into a hundred 10-mm-square pieces, which would take 18 V-scores (higher than the usual 5 or so, but not by that much) it seems like you could get a pretty large set of circuit components. With 2.54-mm-spaced holes along the edge (which I think normally costs extra) you could have six breadboard-compatible/Dupont-cable-compatible pins on each such microboard. Backing off a little bit, if you only panelized the panel into 50 10 mm × 20 mm pieces (13 V-scores), you could have 6 breadboard pins along each edge and a 4-pin USB-A connector at one end of the board.

Going to the extreme, SparkFun sells a US$8 perfboard totally crisscrossed with V-scores so you can break it into 1250 pieces.

Topics

- Contrivances (p. 1143) (45 notes)
- Electronics (p. 1145) (39 notes)
- Pricing (p. 1147) (35 notes)
- Manufacturing (p. 1151) (29 notes)
- Composability (p. 1188) (9 notes)

Some notes on IPL-VI, Lisp's 01958 precursor

Kragen Javier Sitaker, 02021-03-02 (4 minutes)

Reading "A Command Structure for Complex Information Processing" from 01958, and it's pretty astonishing.

It talks about a memory made of dotted pairs in the same year as LISP, and a CPU without arithmetic decades before Steele's Scheme chip.

They hadn't invented garbage collection yet, so they tried to use a "responsibility bit" to distinguish the "owning" pointer to a substructure, with the intent of giving their system a purely hierarchical structure like a hard-link-free Unix filesystem. They didn't know about symlinks, though, and they only associated this bit with `car`, not `cdr` (pp. 17–18 (19–20/54)):

> The single bit, *e*, is an essential piece of auxiliary information. The address, *d*, in a symbol ["symbol" here means what LISP calls `car`] may be the address of another list structure. The responsibility code in a symbol occurrence [`car`] indicates whether this occurrence is "responsible" for the structure designated by *d*. If the same address, *d*, occurs in more than one word, only one of these will indicate responsibility for *d*. ...The need for a definite assignment of responsibility can be seen by considering the process of erasing a list. ...although a system that will handle merging lists also requires a responsibility bit on the link [`cdr`] *f*.

Their design is a two-stack machine significantly before Forth; the operand stack is "L_0, *Communication List*" and the return stack is "L_2, *List of Current Instruction Addresses (CIA)*" (p. 14).

They had the idea of a generator coroutine or OO or thunking or lazy evaluation or duck typing too. p. 7:

> Without breakout devices [this is difficult to gloss], this format would ... permit the operand [argument] of a process [subroutine] to be specified only by giving its address. ...these limitations are removed... by allowing the address for an operand to refer either to the operand itself or to any process that will determine the operand.

Note that this was *before* thunking was invented for ALGOL-60 call by name! On p. 8 we see duck typing, coroutines, and laziness:

> *Identity of Data with Programs*
>
> In current computers, the data are considered "inert." They are symbols to be operated upon by the program. All "structure" of the data is ... encoded implicitly into the programs that work with the data. The structure is embodied in the conventions that determine what bits the processes will decode, and so on.
>
> An alternative approach is to make the data "active." All words in the computer will have the instruction format; there will be "data" programs, and the data will be obtained by executing these programs. ... a list of data, for example, may be specified by a list of processes that determine the data. Since data are only desired "on command" by the processing programs, this approach leads to a computer that, although still serial in its control, contains at any given moment a large number of parallel active programs, frozen in the midst of operation and waiting until called upon to produce the next operation or piece of data. This identity of data with program can be attained only if the processing programs require for their operation no information about the structure of the data programs — only information about how to receive the data from them.

Topics

- Programming (p. 1141) (49 notes)
- History (p. 1153) (24 notes)
- Lisp (p. 1174) (11 notes)
- Memory ownership (p. 1346) (2 notes)

Refreshing Flash memory periodically for archival

Kragen Javier Sitaker, 02021-03-02 (1 minute)

Consider the family of hardware designs explored in How do you fit a high-level language into a microcontroller? Let's look at BBN Lisp (p. 160). Could you use these for centuries-long data retention?

Flash chips are typically specified for 10-year data retention. That means the 4.4 or 16 nJ cited in that note to write each byte aren't forever; that's per byte per decade. If we want to keep 128 mebibytes refreshed indefinitely, as discussed in "Egg of the Phoenix", we need 0.6–2 J per decade, which is 2–7 nW, or about 40 nW per gibibyte. So the STM32L011 mentioned in How do you fit a high-level language into a microcontroller? Let's look at BBN Lisp (p. 160), with its microwatt of stop-mode power, uses about as much power as keeping 25 gibibytes.

If you coupled that chip with 64–256 GiB of NAND Flash, all you'd need is a power source that reliably provides 3–10 μW for decades or centuries. No conventional battery can do this; alternatives include an Atmos-clock-style air-pressure energy harvesting system, as I suggested in Dercuano (where I calculated that the daily barometric variation of a few hundred pascals amounts to a theoretical maximum of a few μW per liter, though Atmos clocks are reported to only use about 250 nW), or perhaps something like the Clarendon Dry Pile. This is much less than the 160 μW for 100 GB I estimated in Dercuano.

Topics

- Contrivances (p. 1143) (45 notes)
- Electronics (p. 1145) (39 notes)
- Memory hardware (p. 1250) (4 notes)
- Archival (p. 1389) (2 notes)

Variable length unaligned bytecode

Kragen Javier Sitaker, 02021-03-02 (updated 02021-03-03) (4 minutes)

Instruction set design for hardware has a different value system from instruction set design for software emulation; sometimes the tradeoffs that improve an instruction set for hardware implementation worsen it for software.

For example, for software emulation, instructions trailed with variable-length operands, like the 8086, are no problem, and inconsistency from one instruction encoding to the next is no problem, but bitfields are a terrible problem. For hardware, bitfields are just wires, but layout inconsistency adds layers of multiplexor delay; for software, bitfields require bit shifting, which requires a loop if your CPU lacks a barrel shifter or has registers too short for the numbers you're shifting.

I was prototyping a virtual machine in C today on my phone, using an unsigned byte array as the VM's memory. I was pleased to see that clang supports GCC's pointers-to-labels extension and generates decent code for it; my `goto dispatch;` at the end of each case even resulted in duplicating the code `dispatch: goto *tbl[mem[pc++]];` in every case, which is one or another variant of

```
ldrb r2, [r4, r6]
add r1, r6, #1
ldr r3, [r5, r2, lsl #1]
bx r3
```

Duplicating this code allows each case to have its own BTB entry, which is helpful for branch prediction, and saves a wasteful unconditional jump. So, the virtual machine bytecode dispatch overhead is only about a factor of 5, and all seemed good in the world. But then I looked at unaligned memory access.

My code to store a 32-bit value in the memory was:

```
#define byt(x) ((x) & 255)
#define store_tet(val, p) do { \
        mem[p] = byt(val); \
        mem[(p)+1] = byt((val) >> 8); \
        mem[(p)+2] = byt((val) >> 16); \
        mem[(p)+3] = byt((val) >> 24); \
} while (0)
```

I figured that this was a portable, #ifdef-free way of marshalling a little-endian 32-bit value, but the phone's ARM processor (a Qualcomm MSM8939 ARMv7l) is running in little-endian mode, so Clang would recognize this and convert it to a simple store. Imagine my surprise at seeing the assembly:

```
strb r7, [r4, #3]
strb r7, [r4, #2]
strb r7, [r4, #1]
strb r7, [r4]
```

ARMv7 supports unaligned access but possibly the compiler is not generating ARMv7 code; it's not using Thumb, for example, although /proc/cpuinfo says it's supported. (`-march=armv7` doesn't help.) (Older ARM versions mostly ignored the low-order bits, but had strange behavior in `ldr` and unpredictable behavior in `ldrh` and `ldrd`. ARMv11 has an even larger set of possibilities.) But there's an option in ARMv7 to trap on unaligned accesses! So you can't rely on it. Also, as a side note, it doesn't seem to be doing the specified bit shifts; in other cases it says things like

```
lsr r0, r3, #24
...
strb r0, [r1, #-1]
lsr r0, r3, #16
strb r0, [r1, #-2]
lsr r0, r3, #8
strb r0, [r1, #-3]
```

which is sort of reasonable, although I'm not sure what happened to r3's least significant byte. Except it's *not* reasonable: a single virtual machine instruction to store a 32-bit value in RAM is going to take 7 instructions on the underlying machine this way, plus dispatch overhead.

So I was happily replacing my byte array with an array of `uint32_t` (it's a prototype! I can do things like that!) and started using `mem[p/4]` and `byt(mem[p/4] >> (p & 3))` before I realized that I'd just replicated the whole unaligned-memory-access problem. If an instruction contains an unaligned 4-byte immediate operand, rounding down `p/4` is not going to fetch that immediate successfully!

So, unless I'm going to require the instructions to be aligned, which probably means mostly not using variable-length instructions, the multiple shifts and loads are going to be happening. Even if you have "hardware support" for unaligned accesses, that doesn't mean you aren't paying a speed penalty, or that it's less than the above code — it might be implemented by trapping to the kernel, making it hundreds of times slower.

So this is a case where I thought hardware and software were more different than they actually are.

Topics

- Performance (p. 1155) (22 notes)
- Assembly-language programming (p. 1175) (11 notes)
- Virtual machines (p. 1182) (9 notes)
- Instruction sets (p. 1214) (6 notes)
- The ARM Acorn RISC Machine

A survey of imperative programming operations' prevalence

Kragen Javier Sitaker, 02021-03-02 (updated 02021-09-11) (61 minutes)

Veskeno needs to be able to fluently (and, above all, deterministically) express the ingredients of software. But what are those ingredients? What are the things we want to be able to express?

What does normal code consist of?

According to our data, a typical procedure consists of 8 or 9 assignment statements, 4 calls to other procedures, 3 IF statements, 1 loop, and 1 escape (RETURN or EXITLOOP). Two of the assignment statements simply assign a constant to a scalar variable, one assigns one scalar variable to another, and 3 or 4 more involve only one operand on the right hand size. The entire procedure probably contains only 2 arithmetic operators. Two of the three conditions in the IF statements involve only a single relational operator, probably = or ≠.

(Tanenbaum, 01978, "Implications of Structured Programming for Machine Architecture"; note how no records, arrays, or other data structures are mentioned)

I thought I'd go and look at a variety of random code in different languages to get an idea of what random code looks like. After all, maybe my own personal preferences and practices aren't typical! Maybe I tend to focus on some small niche area of the wide world of programming. After all, I do write code mostly in Python nowadays. For better or worse, I'm focusing on languages I do know pretty well. I looked at random bits of code in 12 languages.

Modulecounts says npm (JS) is growing at 940 modules a day, Maven Central (Java) 234, PyPI 217, nuget (.NET) 181, Packagist (PHP) 102, crates.io (Rust) 58, and RubyGems 19. Everything else is below 10 new modules per day. I sampled the top three of these, plus another 9 that aren't hip. TIOBE gives C, Java, Python, C++, C# (.NET), Visual Basic (.NET), JS, PHP, SQL, and assembly as its top 10; I included five of these including the top four; GitHub ranks JS, Python, Java, TypeScript, C#, PHP, C++, C, shell, and Ruby, and I included six of these, including the top three. (PHP and C# are the big omissions.) Stack Overflow lists JS, Python, Java, bash, C#, PHP, TypeScript, C++, C, and Golang.

So, hopefully I'm getting a pretty broad range of programming styles, although all these languages except m4 are pretty similar.

Scheme

First consider this chunk of code from Ur-Scheme:

```
(define interned-symbol-list '())
(define (intern symbol)
  (interning symbol interned-symbol-list))
```

```
(define (interning symbol symlist)
  (cond ((null? symlist)
         ;; XXX isn't this kind of duplicative with the global variables stuff?
         (set! interned-symbol-list
               (cons (list symbol (new-label)) interned-symbol-list))
         (car interned-symbol-list))
        ((eq? symbol (caar symlist)) (car symlist))
        (else (interning symbol (cdr symlist)))))
(define (symbol-value symbol) (cadr (intern symbol)))
```

I think this is more or less normal code. It has some definitions, an assignment, a loop, a mutable variable, some immutable arguments, some comparisons, some conditionals, some invocations of some primitive functionality (mostly to access container objects), and some calls to high-level functions. It contains an unusually small amount of mutation (one assignment), an unusually small number of constants (just one, nil), and an unusually small amount of sequencing (just the sequence of two consequents of `null? symlist` and the top-level sequence of definitions). You can't tell from looking, but it invokes two macros: both `define` and `cond` are defined as macros in Ur-Scheme.

Python

Here's some more code that is also pretty normal, from bzrlib.plugins.gtk:

```
class LockDialog(Gtk.Dialog):

    def __init__(self, branch):
        super(LockDialog, self).__init__()

        self.branch = branch

        self.set_title('Lock Not Held')

        self.get_content_area().add(
            Gtk.Label(label=(
                'This operation cannot be completed as '
                'another application has locked the branch.')))

        self.add_button('Break Lock', RESPONSE_BREAK)
        self.add_button(Gtk.STOCK_CANCEL, RESPONSE_CANCEL)

        self.get_content_area().show_all()
```

This has a class definition with inheritance and a constructor, a superclass constructor invocation, some arguments, some references to free variables (imported from other modules in this case), some constant strings, a sequence of method calls, and even an object instantiation and a named argument. The only primitives used, other than those mentioned above, are attribute access.

C

Here's some slightly less normal code, from gpsd/json.c:

```
json_debug_trace((1, "JSON parse of '%s' begins.\n", cp));

/* parse input JSON */
for (; *cp != '\0'; cp++) {
json_debug_trace((2, "State %-14s, looking at '%c' (%p)\n",
          statenames[state], *cp, cp));
switch (state) {
case init:
    if (isspace(*cp))
    continue;
    else if (*cp == '{')
    state = await_attr;
    else {
    json_debug_trace((1,
            "Non-WS when expecting object start.\n"));
    return JSON_ERR_OBSTART;
    }
    break;
case await_attr:
    if (isspace(*cp))
    continue;
```

I would have included the whole function, but it's 454 lines long. This includes three calls to a C macro, which invokes a variadic function called json_trace; a couple of character constants; a loop; a couple of integer constants; three string constants; an assignment; a passel of sequencing; pointer arithmetic, including mutation by postincrement; indexing into an array by the enum; a switch statement on an enum; some early exits; lots of primitive operations; and a couple of invocations of isspace, from the standard library, which could be a subroutine but is probably a macro.

JS

Here's some more fairly normal code, this time from a JS version of Python's bisect binary search library, included in a package called "crossfilter". I've removed the comments but they already didn't say who the author was:

```
function bisect_by(f) {
  function bisectLeft(a, x, lo, hi) {
    while (lo < hi) {
      var mid = lo + hi >>> 1;
      if (f(a[mid]) < x) lo = mid + 1;
      else hi = mid;
    }
    return lo;
  }

  function bisectRight(a, x, lo, hi) {
    while (lo < hi) {
      var mid = lo + hi >>> 1;
      if (x < f(a[mid])) hi = mid;
      else lo = mid + 1;
    }
```

```
    return lo;
  }

  bisectRight.right = bisectRight;
  bisectRight.left = bisectLeft;
  return bisectRight;
}
```

This has some functions with parameters; conditionals; a couple of loops; comparisons; a few assignments; a little bit of arithmetic (unlike the others!), including a bit shift; a higher-order function taking a functional argument and returning closures; some assignments to attributes; and indexing into arrays.

Java

Here's some more code that isn't very normal at all, from Mako, "a simple stack-based virtual game console, designed to be as simple as possible to implement", but mostly another copy of Forth:

```
public void tick() {
    int o = m[m[PC]++];
    int a, b;

    switch(o) {
        case OP_CONST  :  push(m[m[PC]++]);                          break;
        case OP_CALL   : rpush(m[PC]+1); m[PC] = m[m[PC]];           break;
        case OP_JUMP   :                 m[PC] = m[m[PC]];           break;
        case OP_JUMPZ  : m[PC] = pop()==0 ? m[m[PC]] : m[PC]+1;      break;
        case OP_JUMPIF : m[PC] = pop()!=0 ? m[m[PC]] : m[PC]+1;      break;
        case OP_LOAD   : push(load(pop()));                          break;
        case OP_STOR   : stor(pop(),pop());                          break;
        case OP_RETURN : m[PC] = rpop();                             break;
        case OP_DROP   : pop();                                      break;
        case OP_SWAP   : a = pop(); b = pop(); push(a); push(b);     break;
        case OP_DUP    : push(m[m[DP]-1]);                           break;
        case OP_OVER   : push(m[m[DP]-2]);                           break;
        case OP_STR    : rpush(pop());                               break;
        case OP_RTS    : push(rpop());                               break;
        case OP_ADD    : a = pop(); b = pop(); push(b+a);            break;
        case OP_SUB    : a = pop(); b = pop(); push(b-a);            break;
        case OP_MUL    : a = pop(); b = pop(); push(b*a);            break;
        case OP_DIV    : a = pop(); b = pop(); push(b/a);            break;
        case OP_MOD    : a = pop(); b = pop(); push(mod(b,a));       break;
        case OP_AND    : a = pop(); b = pop(); push(b&a);            break;
        case OP_OR     : a = pop(); b = pop(); push(b|a);            break;
        case OP_XOR    : a = pop(); b = pop(); push(b^a);            break;
        case OP_NOT    : push(~pop());                               break;
        case OP_SGT    : a = pop(); b = pop(); push(b>a ? -1:0);     break;
        case OP_SLT    : a = pop(); b = pop(); push(b<a ? -1:0);     break;
        case OP_NEXT   : m[PC] = --m[m[RP]-1]<0?m[PC]+1:m[m[PC]]; break;
    }
}
```

This has a method definition; a switch statement; some constants;

a bunch of method calls; a bunch of arithmetic primitives (including bitwise operators, which we haven't seen before); indexing into arrays; conditionals; lots of assignment and other mutation; and a number of numeric constants.

Here's some more normal code, also in Java, reformatted from the OpenJDK; this code is compilation output from some kind of macro processing, and it implements a FIFO of double-precision floating-point numbers:

```
public DoubleBuffer asReadOnlyBuffer() {
    return new HeapDoubleBufferR(hb, this.markValue(), this.position(),
                                 this.limit(), this.capacity(), offset);
}

protected int ix(int i) { return i + offset; }
public double get() { return hb[ix(nextGetIndex())]; }
public double get(int i) { return hb[ix(checkIndex(i))]; }

public DoubleBuffer get(double[] dst, int offset, int length) {
    checkBounds(offset, length, dst.length);
    if (length > remaining())
        throw new BufferUnderflowException();
    System.arraycopy(hb, ix(position()), dst, offset, length);
    position(position() + length);
    return this;
}
```

This contains a number of method calls with varying protection, lots of type declarations (including methods overridden by type signature), lots of method calls, a little attribute access, some implicit self-instance-variable access, object instantiation, array indexing, a little bit of arithmetic (two additions and a comparison), a conditional, some mutation, and an exception.

Perl

Here's some more pretty normal code, this time in Perl, from Net::DBus::Binding::Message::Error:

```
=item my $error = Net::DBus::Binding::Message::Error->new(
      replyto => $method_call, name => $name, description => $description);

Creates a new message, representing an error which occurred during
the handling of the method call object passed in as the C<replyto>
parameter. The C<name> parameter is the formal name of the error
condition, while the C<description> is a short piece of text giving
more specific information on the error.

=cut

sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my %params = @_;
```

```
    my $replyto = exists $params{replyto} ? $params{replyto} : die "replyto param
eter is required";

    my $msg = exists $params{message} ? $params{message} :
        Net::DBus::Binding::Message::Error::_create
        (
         $replyto->{message},
         ($params{name} ? $params{name} : die "name parameter is required"),

         ($params{description} ? $params{description} : die "description paramete
r is required"));

    my $self = $class->SUPER::new(message => $msg);

    bless $self, $class;

    return $self;
}
```

This is a method definition including properly marked-up documentation. It contains five conditionals, six local variables, no mutation, three exceptions, a bunch of hash table lookups by string (like Python, Perl uses string-indexed hash tables instead of record types), a method call (on the superclass), deeply nested namespaces, and lots and lots of invocations of primitives, including `shift`, `ref`, `exists`, `@_`, and `bless`. It has a lot of string literals if we count the parameter/attribute names and the package names, but otherwise the only constants are some string literals for error messages.

Elisp

Here's some more pretty normal Lisp code, this time in Elisp, from Eric Ludlam's Speedbar package:

```
(defun speedbar-add-supported-extension (extension)
  "Add EXTENSION as a new supported extension for speedbar tagging.
This should start with a `.' if it is not a complete file name, and
the dot should NOT be quoted in with \\.  Other regular expression
matchers are allowed however.  EXTENSION may be a single string or a
list of strings."
  (interactive "sExtension: ")
  (if (not (listp extension)) (setq extension (list extension)))
  (while extension
    (if (member (car extension) speedbar-supported-extension-expressions)
        nil
      (setq speedbar-supported-extension-expressions
            (cons (car extension) speedbar-supported-extension-expressions)))
    (setq extension (cdr extension)))
  (setq speedbar-file-regexp (speedbar-extension-list-to-regex
                              speedbar-supported-extension-expressions)))
```

This function contains two conditionals, four assignments, a loop, properly formatted documentation, one to three constants (nil and two strings) depending on how you count, lots of calls to primitives,

and a call to another function in the same package.

Lua

Here's some more pretty normal code, this time in Lua from NMap, slightly reformatted. This is from Patrik Karlsson's interface to get packets from WinPcap:

```
-- Holds the two supported authentication mechanisms PWD and NULL
Authentication = {
  PWD = {
    new = function(self, username, password)
      local o = {
        type = 1,
        username = username,
        password = password,
      }
      setmetatable(o, self)
      self.__index = self
      return o
    end,

    __tostring = function(self)
      local DUMMY = 0
      return bin.pack(">SSSSAA", self.type, DUMMY, #self.username,
                      #self.password, self.username, self.password)
    end,
  },
```

Her we have three levels of nesting of Lua tables (dictionaries), `...Authentication.PWD.new`. There are a couple of functions (which are methods on a metatable object; one is a constructor), a couple of local variable declarations, an assignment to the `__index` attribute, a few accesses to attributes, invocation of the `#` length primitive, a call to a function from another module, and the construction of a new table `o`. There are three constants, one of which is a string in a binary serialization little language. There is a sequence of four statements in `new`.

C++

Here's some more much less normal code, this time in C++, from OpenSCAD's ColorModule (slightly reformatted):

```
#include "colormap.h"
AbstractNode *ColorModule::instantiate(const Context *ctx,
   const ModuleInstantiation *inst,
   const EvalContext *evalctx)
const
{
   ColorNode *node = new ColorNode(inst);

   node->color[0] = node->color[1] = node->color[2] = -1.0;
   node->color[3] = 1.0;

   AssignmentList args;
```

```
  args += Assignment("c", NULL), Assignment("alpha", NULL);

  Context c(ctx);
  c.setVariables(args, evalctx);

  Value v = c.lookup_variable("c");
  if (v.type() == Value::VECTOR) {
    for (size_t i = 0; i < 4; i++) {
      node->color[i] = i < v.toVector().size()
        ? v.toVector()[i].toDouble()
        : 1.0;
      if (node->color[i] > 1)
        PRINTB_NOCACHE("WARNING: color() expects numbers between"
                       " 0.0 and 1.0. Value of %.1f is too large.",
                        node->color[i]);
    }
  } else if (v.type() == Value::STRING) {
    std::string colorname = v.toString();
```

Here we see an `#include`, (the beginning of) a method definition, lots of parameters and other local variables, lots of type declarations, some object instantiations, lots of accesses to attributes ("instance variables" or "fields"), constness, lots of constants (numeric, string, and enum), arithmetic (using postincrement mutation) to step through an array and check numeric ranges, a loop, four conditionals, pseudo-RAII (`Context`'s constructors and destructors maintain a context stack for `lookup_variable` — it isn't so much that they acquire resources so much as that they automatically release them), operator overloading, lots of method calls, array indexing, macros, and floating point.

Bourne shell

Here's a shell script from eglibc's test suite:

```
common_objpfx=$1
run_program_prefix=$2
objpfx=$3

LC_ALL=C
export LC_ALL

# Create the domain directories.
mkdir -p ${objpfx}domaindir/de_DE/LC_MESSAGES
mkdir -p ${objpfx}domaindir/fr_FR/LC_MESSAGES
# Populate them.
msgfmt -o ${objpfx}domaindir/de_DE/LC_MESSAGES/multithread.mo tst-gettext4-de.po
msgfmt -o ${objpfx}domaindir/fr_FR/LC_MESSAGES/multithread.mo tst-gettext4-fr.po

GCONV_PATH=${common_objpfx}iconvdata
export GCONV_PATH
LOCPATH=${common_objpfx}localedata
export LOCPATH

${run_program_prefix} ${objpfx}tst-gettext4 > ${objpfx}tst-gettext4.out

exit $?
```

This has very little in common with the other examples, although we can identify parameters (`$1`, `$2`), variables, sequencing, string constants, primitives (`export`, `exit`), invocation of library functionality analogous to library functions (`mkdir`, `msgfmt`), containers of data (directories), and nested namespaces.

It's not entirely coincidental that this shell script lacks conditionals, loops, and subroutines (other than the whole script). It's pretty common for shell scripts to be just straight sequences like that: just a sequence of mutations, slightly parameterized.

Tcl

Here's some fairly normal Tcl code from the ArsDigita Community System, somewhat reformatted:

```
# calendar-defs.tcl
# by philg@mit.edu late 1998
# for the /calendar system documented at /doc/calendar.html

proc calendar_system_owner {} {
    return [ad_parameter SystemOwner calendar [ad_system_owner]]
}

proc calendar_footer {} {
    return [ad_footer [calendar_system_owner]]
}

ns_share ad_user_contributions_summary_proc_list

if { ![info exists ad_user_contributions_summary_proc_list]

   || [util_search_list_of_lists $ad_user_contributions_summary_proc_list "/calendar postings" 0]
       == -1 } {
    lappend ad_user_contributions_summary_proc_list \
        [list "/calendar postings" calendar_user_contributions 0]
}
```

Again we see invocations of "primitives" (`if`, `lappend`, `list`); two subroutine definitions (which could have had parameters but don't); variables; invocations of non-primitive functionality like `util_search_list_of_lists` and `ad_parameter`; string constants (all over the place), some of which are also integer constants; sequencing; and a couple of conditionals.

Tcl is kind of close to shell scripts in a lot of ways — its only data type is ostensibly strings — though it's imported some aspects of Lisp. The ACS codebase is probably less shell-scripty than most Tcl codebases. I think of this script from the OpenTitan project as being more typical of Tcl:

```
# Copyright lowRISC contributors.
# Licensed under the Apache License, Version 2.0, see LICENSE for details.
# SPDX-License-Identifier: Apache-2.0
```

```
source ./tcl/sta_common.tcl

set overall_rpt_file "${lr_synth_rpt_out}/timing/overall"
timing_report $lr_synth_clk_input $overall_rpt_file $lr_synth_sta_overall_paths

set lr_synth_path_group_list [list]

setup_path_groups $lr_synth_inputs $lr_synth_outputs lr_synth_path_group_list

foreach path_group $lr_synth_path_group_list {
  puts $path_group
  set path_group_rpt_file "${lr_synth_rpt_out}/timing/$path_group"
  timing_report $path_group $path_group_rpt_file $lr_synth_sta_paths_per_group
}

exit
```

Here we have variables (which are in some sense parameters, since they occur free), a loop, and sequencing, but no conditionals. There's technically an assignment in the loop but the mutation to that variable is kind of nonessential. However, since it's almost purely a sequence, mutation is the only way for it to do anything useful.

Tcl and shell are both very easy to get started with, like keyboard macros, but very bug-prone and kind of hard to understand. Part of the problem is that much of their semantics is based on string interpolation.

m4

Speaking of which, what does typical m4 look like? Dennis Ritchie's m4 is a Turing-complete macro language, in which (unlike in bash and Tcl, like in Make) the results of macro substitution are subject to further macro substitution, which allows you to write a loop by writing a macro that conditionally expands to invoke itself. For example, although it has a built-in `len` operation that gives the length of a string, we can also define a new one recursively in terms of its built-in `ifelse`, `incr`, and `substr` operations:

```
define(`length',`ifelse(,$1,0,`incr(length(substr($1,1)))')')
```

I adapted this from this example found on the Softpanorama page about m4:

```
define(len,`ifelse($1,,0,`eval(1+len(substr($1,2)))')')
```

This definition, however, has a bug in it: the `2` should be `1`, a bug introduced in an earlier version of this example in Kernighan and Plauger's *Software Tools* in 01976 (p. 280). It took me quite a while to debug it because the definition line doesn't quote `len`, so my attempts to redefine it were (apparently) silently ignored; I was instead defining a macro named `0`:

```
$ m4
define(len,`ifelse($1,,0,`eval(1+len(substr($1,2)))')')len(wotcha)
3
```

```
len(half)
2
define(len,`ifelse($1,,0,`eval(1+len(substr($1,1)))')')len(wotcha)
3
len(huh)
2
len(why does nothing make sense)
13
define(`len',`ifelse($1,,0,`eval(1+len(substr($1,1)))')')len(wotcha)
6
```

The built-in `len` macro, by contrast, doesn't get substituted unless you offer it arguments (though this behavior is a GNU extension), which permitted the first definition to succeed.

This accidentally-defined macro `0` can't be invoked by normal means because its name isn't "a word", but it does exist:

```
0(wibbling)
0(wibbling)
indir(0,wibbling)
5
```

The output can build up macro names through concatenation, either intentionally or unintentionally, which means that both the input† and the output of the macro are subject to macro expansion. I think that is actually sufficient to construct conditionals without the `ifelse` builtin, but I haven't figured out how.

```
define(foo,l$1)foo(en)(something)
9
foo(e)n(something)
9
```

In *Software Tools* (p. 281) Kernighan and Plauger warn:

> As you can see this is not the most transparent programming language in the world. ...you get the hang of it. But beware of becoming too clever with macros. In principle, `macro` [the early version of m4 presented in the book] is capable of performing any computing task, but it is all too easy to write unreadable macros that cause more trouble than they save work.

Hopefully this gives some flavor of both m4's capabilities and its nightmarish bug-proneness.

Most current use of m4 is by way of autoconf. Here's part of the autoconf script for an old version of libart:

```
dnl AM_PATH_LIBART([MINIMUM-VERSION, [ACTION-IF-FOUND [, ACTION-IF-NOT-FOUND]]])
dnl Test for LIBART, and define LIBART_CFLAGS and LIBART_LIBS
dnl
AC_DEFUN([AM_PATH_LIBART],
[dnl
dnl Get the cflags and libraries from the libart-config script
dnl

AC_ARG_WITH(libart-prefix,[  --with-libart-prefix=PFX   Prefix where LIBART is in
stalled (optional)],
```

```
          libart_prefix="$withval", libart_prefix="")

AC_ARG_WITH(libart-exec-prefix,[  --with-libart-exec-prefix=PFX Exec prefix where
 LIBART is installed (optional)],
          libart_exec_prefix="$withval", libart_exec_prefix="")

AC_ARG_ENABLE(libarttest, [  --disable-libarttest       Do not try to compile and
 run a test LIBART program],
                 , enable_libarttest=yes)
```

m4 is a templating language with dangerous delusions of grandeur, capable of not only Turing-complete computation but higher-order programming; but, as you can see, autoconf has managed to build its own castle on that swamp, turning m4 into an entirely separate programming language. Unfortunately I don't know enough about autoconf to know what those ugly names mean and how they work together.

Here's an excerpt from what claims to be a typical 02005 use of m4 for configuring Sendmail, which is a lot closer to vanilla m4. You'll note that it has a lot of `dnl` invocations; these are to prevent spurious newlines from being emitted, but they are completely unnecessary in this case because of the leading `divert(-1)`; it's just cargo-cult code:

```
divert(-1)
include(`/usr/share/sendmail-cf/m4/cf.m4')
VERSIONID(`linux setup for my Linux dist')dnl
OSTYPE(`linux')
define(`confDEF_USER_ID',``8:12'')dnl
undefine(`UUCP_RELAY')dnl
undefine(`BITNET_RELAY')dnl
define(`PROCMAIL_MAILER_PATH',`/usr/bin/procmail')dnl
define(`ALIAS_FILE', `/etc/aliases')dnl
define(`UUCP_MAILER_MAX', `2000000')dnl
```

D. Robert Adams gives this motivating example in his introduction to m4, sometime prior to 02006:

```
define(`PAGE_HEADER',
`<table border="0" background="steel.jpg" width="100%">
  <tr>
    <td align="left">$1</td>
    <td align="right">$2</td>
  </tr>
</table>
<div align=right>
  Last Modified: esyscmd(`date')
</div>
')
```

I probably *should* have included PHP, another templating language with dangerous delusions of grandeur, but I don't have anything that I think is "typical PHP" code handy.

† Macro argument are macro-expanded by default, but you can

quote them. Kernighan and Plauger make this change in the middle of their chapter in *Software Tools* about m4, saying, "for common uses like replacing symbolic parameters, the two methods produce the same result," and it contributes considerably to m4's already impressive bug-proneness.

Forth

Unfortunately I don't have a lot of confidence that this represents "typical" Forth, but it's one of the few Forth programs I've actually used and didn't write:

```
\ tt.pfe        Tetris for terminals, redone in ANSI-Forth.
\               Written 05Apr94 by Dirk Uwe Zoller,
\                       e-mail duz@roxi.rz.fht-mannheim.de.
\               Look&feel stolen from Mike Taylor's "TETRIS FOR TERMINALS"
\
\               Please copy and share this program, modify it for your system
\               and improve it as you like. But don't remove this notice.
\ ...

: draw-pit      \ --- ; draw the contents of the pit
                deep 0 do  i draw-line  loop ;

: show-key      \ char --- ; visualization of that character
                dup bl <
                if  [char] @ or  [char] ^ emit  emit  space
                else  [char] ` emit  emit  [char] ' emit
                then ;

: show-help     \ --- ; display some explanations
                30  1 at-xy ." ***** T E T R I S *****"
                30  2 at-xy ." ======================="
                30  4 at-xy ." Use keys:"
                32  5 at-xy left-key    show-key ."  Move left"
\ ...
                ;

: update-score  \ --- ; display current score
                38 16 at-xy score @ 3 .r
                38 17 at-xy pieces @ 3 .r
                38 18 at-xy levels @ 3 .r ;
```

Here we have three subroutines, one of which takes a parameter; they contain a conditional and a loop; they invoke primitives like `@`, `or` , `[char]`, and `emit`. This excerpt barely uses variables, `score`, `pieces`, and `levels`; `i` is not really a variable, though it's variable-like. The parameter to `show-key` is not a variable. `deep` and `left-key` are constants. There's a little arithmetic (on ASCII codes for keys). There are lots of places where one subroutine invokes another; `draw-line` and `show-key` are parts of the Tetris game. `at-xy` comes with PFE.

I think this program uses less variables and longer subroutines than is typical for Forth, but it shows how you can use high-level Forth words to script at whatever level is comfortable for your application.

What do these things look like compiled?

The execution model for all of the above, except m4, are actually fairly similar. They're eager imperative languages, equipped with closed subroutines and primitive facilities for arithmetic and constructing and accessing composite data structures. They're all equipped with textual namespaces to use for describing dataflow connections by connascence of name. Though not all the control flow shown above is single-entry single-exit, it *is* all "structured" in the truer sense. None of them have pattern matching or strong static type checking, much less dependent types. None of them (except m4) have an outlandish execution model like Prolog, Erlang, Haskell, Miranda, SNOBOL, or Icon, although Python and Lua do have one or another kind of coroutine, and most of them can use threads.

There are some major practical differences. Python lists don't support efficient FP-persistent incremental construction like my initial example from Ur-Scheme used, and we lack garbage collection entirely in C, C++, and Forth. JS supports higher-order functions with closures, a feature pioneered in Scheme but missing from C, C++, Forth, and Elisp, and much less used in the other languages than in JS and perhaps Lua. Elisp, C, C++, Forth, Java, and Scheme don't support the effortless digraph-of-dictionaries-referencing-each-other memory structure that characterizes Python, Perl, Lua, and JS, so I had to write `(caar symlist)` instead of the more transparent `sym.name`. Tcl, m4, and the Bourne shell don't support references at all, except by name.

Their scoping differs — Elisp has a single global namespace where it temporarily binds local variables; Scheme has a single global namespace and then lexically-nested block scopes within it; C has three nested kinds of namespace (extern, file-static, and block scope, which nests arbitrarily); Forth word scopes extend from declarations forward until something hides the declaration, such as a change of wordlist; and there are nested hierarchical namespaces in JS, Lua, Java, Perl, and Python. (I forget how scoping works in Tcl and m4.) But for the most part these scoping differences disappear before execution time.

Elisp, Forth, Scheme, Tcl, the Bourne shell, m4, and C don't have a concept of "object scope", "class scope", or "methods", while Java, Python, C++, JS, Lua, and Perl all have varying concepts. All of these languages have some way to invoke a function whose identity is not known until runtime, although in Tcl and the Bourne shell this is done with a string referencing a function in the global namespace, and in C you have to do extra work to invoke a *closure*. Java and C++ mostly treat objects as primary and methods as mere details of objects, leading to things like the `Callable` interface, while functions are first-class objects in Elisp, Scheme, Python, JS, Lua, and Perl.

But most of these differences are sort of details.

The Elisp implementation in more depth

Consider that initial example:

```
(define interned-symbol-list '())
(define (intern symbol)
  (interning symbol interned-symbol-list))
```

```
(define (interning symbol symlist)
  (cond ((null? symlist)
         ;; XXX isn't this kind of duplicative with the global variables stuff?
         (set! interned-symbol-list
               (cons (list symbol (new-label)) interned-symbol-list))
         (car interned-symbol-list))
        ((eq? symbol (caar symlist)) (car symlist))
        (else (interning symbol (cdr symlist)))))
(define (symbol-value symbol) (cadr (intern symbol)))
```

We can translate this into Elisp as follows:

```
(defvar interned-symbol-list '())

(defun ur-intern (symbol)
  (interning symbol interned-symbol-list))

(defun new-label ()
  (gensym))

(defun interning (symbol symlist)
  (cond ((null symlist)
         (setq interned-symbol-list
               (cons (list symbol (new-label)) interned-symbol-list))
         (car interned-symbol-list))
        ((eq symbol (caar symlist))
         (car symlist))
        (t
         (interning symbol (cdr symlist)))))

(defun ur-symbol-value (symbol)
  (cadr (ur-intern symbol)))
```

This byte-compiles as follows, with the unprintable bytes removed:

```
(defalias 'ur-intern #[(symbol) "(redacted)"
                       [symbol interned-symbol-list interning] 3])
(defalias 'new-label #[nil "(redacted)" [gensym] 1])
(defalias 'interning
  #[(symbol symlist) "(redacted)"
    [symlist symbol interned-symbol-list x new-label interning]
    4])
(defalias 'ur-symbol-value #[(symbol) "(redacted)"
                              [symbol x ur-intern] 3])
```

The redacted bytecodes disassemble as follows:

```
byte code for ur-intern:
  args: (symbol)
0       constant  interning
1       varref    symbol
2       varref    interned-symbol-list
3       call      2
```

```
4       return

byte code for new-label:
  args: nil
0       constant  gensym
1       call      0
2       return
```

Those two are very simple, just some eager nested expressions. But `interning` contains conditionals, and a recursive loop, although its tail-call nature is not visible in the bytecode:

```
byte code for interning:
  args: (symbol symlist)
0       varref    symlist

1       goto-if-not-nil 1                 ; forward jump past the first cond case; note the double negation
4       varref    symbol
5       constant  new-label
6       call      0
7       list2                             ; build a 2-item list
8       varref    interned-symbol-list
9       cons

10      dup                               ; one reference for the variable, the other for return value
11      varset    interned-symbol-list
12      car
13      return
14:1    varref    symbol                  ; second case:
15      varref    symlist
16      dup

17      varbind   x                       ; note, not varset; this creates a local binding

18      car                               ; `caar` compiles into two successive `car` bytecodes
19      car
20      unbind    1                       ; discard useless x binding
21      eq
22      goto-if-nil 2                     ; jump to the else
25      varref    symlist
26      car
27      return
28:2    constant  interning               ; set up the recursive call
29      varref    symbol
30      varref    symlist
31      cdr                               ; reduce toward the base case
32      call      2
33      return
```

`max-lisp-eval-depth` defaults to 500, and it does kill functions like this

in my version of Emacs when they tail-recurse too deeply, whether byte-compiled or not. That is, Elisp doesn't have tail-call elimination as Ur-Scheme does.

If you want a loop in Elisp, you need to use an explicit looping construct, such as `while`, which gets compiled to `goto-if-not-nil` bytecodes like the ones above, just backwards instead of forwards.

```
byte code for ur-symbol-value:
  args: (symbol)
0       constant  ur-intern
1       varref    symbol
2       call      1
3       dup
4       varbind   x
5       cdr
6       car
7       unbind    1
8       return
```

Except for the special forms `define`, `cond`, and `setq`, the Lisp source makes no distinction between primitive operations like `cdr` and `null`, one one hand, and invocations of ordinary functions like `interning` and `new-label`, on the other. But the bytecode compiler certainly does, as an efficiency hack; it has special bytecodes for the basic Lisp functions (in this case, `list`, `cons`, `car`, `eq`, `null` (merged with the `cond`), and `cdr`), as well as lots of Emacs-specific operations like `save-excursion`, `forward-char`, `insert`, and `forward-word`, which don't occur here. This makes the bytecode very compact, though it still carries around a "constant vector" for opcodes like `constant` and `varref` to index into. It just doesn't have to indirect through the constant vector and look up a function binding every time you invoke `car`, so it runs, reportedly, about four times faster.

(In the case of `interning`, the constant vector is 6 items, so presumably 56 more bytes including a count field; there's also a "bytecode object" that ties the bytecode to the arguments, constant vector, etc. It doesn't seem to have been designed for minimum memory usage, despite the admirable compactness of the bytecode itself.)

The `exec_byte_code` function in the Emacs source code that executes a bytecode-compiled function like the above is about 1500 lines of C.

The `interning` bytecode above is pretty short, 34 bytes. Here's the implementation of the `Bvarref` bytecode that begins the `interning` function, as well as its cousins:

```
CASE (Bvarref7):
  op = FETCH2;
  goto varref;

CASE (Bvarref):
CASE (Bvarref1):
CASE (Bvarref2):
CASE (Bvarref3):
CASE (Bvarref4):
```

```
CASE (Bvarref5):
  op = op - Bvarref;
  goto varref;

/* This seems to be the most frequently executed byte-code
   among the Bvarref's, so avoid a goto here.  */
CASE (Bvarref6):
  op = FETCH;
varref:
  {
    Lisp_Object v1, v2;

    v1 = vectorp[op];
    if (SYMBOLP (v1))
      {
    if (XSYMBOL (v1)->redirect != SYMBOL_PLAINVAL
        || (v2 = SYMBOL_VAL (XSYMBOL (v1)),
        EQ (v2, Qunbound)))
      {
        BEFORE_POTENTIAL_GC ();
        v2 = Fsymbol_value (v1);
        AFTER_POTENTIAL_GC ();
      }
      }
    else
      {
    BEFORE_POTENTIAL_GC ();
    v2 = Fsymbol_value (v1);
    AFTER_POTENTIAL_GC ();
      }
    PUSH (v2);
    NEXT;
  }
```

I guess this means that the Elisp definition of "the binding of a variable" is sort of complicated.

FETCH isn't a constant; it's defined as `*stack.pc++`, and FETCH2 is similar but fetches two bytes. But in this case none of that comes into play.

There are several other such groups of 8 opcodes: 6 with the operand packed into the low three bits of the byte, and two "breakouts" that consume one or two immediate bytes to get the real operand. `stack-ref`, `varref`, `varset`, `varbind`, `call`, and `unbind` all work this way; there are also four fixed-arity versions of the `list` function plus a variable-arity version, three fixed-arity versions of the `concat` function plus a variable-arity version, and some others. There are 173 bytecodes defined in all, so it's about 9 lines of code per bytecode, but 48 of the bytecodes are these groups of 8.

In SBCL

The same Elisp code is valid as Common Lisp code, and SBCL (1.0.57.x) compiles `interning` as follows for amd64. You will note that it is noticeably more than 34+56 = 90 bytes, more like 439 bytes; our sweet innocent 9-line Lisp function has exploded into 131 assembly

instructions:

```
* (compile 'interning)

INTERNING
NIL
NIL
* (disassemble 'interning)

; disassembly for INTERNING

; 02A2FBF7:       4881FE17001020   CMP RSI, 537919511         ; no-arg-parsing en⚬
⚬try point
;      BFE:       744B             JEQ L2
;      C00:       8BC6             MOV EAX, ESI
;      C02:       240F             AND AL, 15
;      C04:       3C07             CMP AL, 7
;      C06:       0F8559010000     JNE L11
;      C0C:       488BC6           MOV RAX, RSI
;      C0F:       488B48F9         MOV RCX, [RAX-7]
;      C13:       8BC1             MOV EAX, ECX
;      C15:       240F             AND AL, 15
;      C17:       3C07             CMP AL, 7
;      C19:       0F854D010000     JNE L12
;      C1F:       488B49F9         MOV RCX, [RCX-7]
;      C23:       4939C8           CMP R8, RCX
;      C26:       750A             JNE L1
;      C28:       488B56F9         MOV RDX, [RSI-7]
;      C2C: L0:   488BE5           MOV RSP, RBP
;      C2F:       F8               CLC
;      C30:       5D               POP RBP
;      C31:       C3               RET
;      C32: L1:   488B7E01         MOV RDI, [RSI+1]
;      C36:       498BD0           MOV RDX, R8

;      C39:       488B0540FFFFFF   MOV RAX, [RIP-192]         ; #<FDEFINITION obj⚬
⚬ect for INTERNING>
;      C40:       B904000000       MOV ECX, 4
;      C45:       FF7508           PUSH QWORD PTR [RBP+8]
;      C48:       FF6009           JMP QWORD PTR [RAX+9]
;      C4B: L2:   488D5424F0       LEA RDX, [RSP-16]
;      C50:       4883EC18         SUB RSP, 24

;      C54:       488B052DFFFFFF   MOV RAX, [RIP-211]         ; #<FDEFINITION obj⚬
⚬ect for NEW-LABEL>
;      C5B:       31C9             XOR ECX, ECX
;      C5D:       48892A           MOV [RDX], RBP
;      C60:       488BEA           MOV RBP, RDX
;      C63:       FF5009           CALL QWORD PTR [RAX+9]
;      C66:       480F42E3         CMOVB RSP, RBX
;      C6A:       488B75F0         MOV RSI, [RBP-16]
;      C6E:       4C8B45F8         MOV R8, [RBP-8]
;      C72:       488BFA           MOV RDI, RDX
;      C75:       498BD0           MOV RDX, R8
;      C78:       49896C2440       MOV [R12+64], RBP
```

```
;       C7D:        4D8B5C2418       MOV R11, [R12+24]
;       C82:        498D5B20         LEA RBX, [R11+32]
;       C86:        49395C2420       CMP [R12+32], RBX
;       C8B:        0F86E0000000     JBE L13
;       C91:        49895C2418       MOV [R12+24], RBX
;       C96:        498D5B07         LEA RBX, [R11+7]
;       C9A: L3:    488BC3           MOV RAX, RBX
;       C9D:        488950F9         MOV [RAX-7], RDX
;       CA1:        4883C010         ADD RAX, 16
;       CA5:        488940F1         MOV [RAX-15], RAX
;       CA9:        488978F9         MOV [RAX-7], RDI
;       CAD:        48C7400117001020 MOV QWORD PTR [RAX+1], 537919511
;       CB5:        49316C2440       XOR [R12+64], RBP
;       CBA:        7402             JEQ L4

;       CBC:        CC09             BREAK 9                      ; pending interrupt
 trap

;       CBE: L4:    488B05CBFEFFFF   MOV RAX, [RIP-309]           ; 'INTERNED-SYMBOL-
LIST
;       CC5:        488B5021         MOV RDX, [RAX+33]
;       CC9:        498B1414         MOV RDX, [R12+RDX]
;       CCD:        4883FA61         CMP RDX, 97
;       CD1:        7504             JNE L5
;       CD3:        488B50F9         MOV RDX, [RAX-7]
;       CD7: L5:    4883FA51         CMP RDX, 81
;       CDB:        0F84A7000000     JEQ L14
;       CE1:        49896C2440       MOV [R12+64], RBP
;       CE6:        4D8B5C2418       MOV R11, [R12+24]
;       CEB:        498D4B10         LEA RCX, [R11+16]
;       CEF:        49394C2420       CMP [R12+32], RCX
;       CF4:        0F8693000000     JBE L15
;       CFA:        49894C2418       MOV [R12+24], RCX
;       CFF:        498D4B07         LEA RCX, [R11+7]
;       D03: L6:    49316C2440       XOR [R12+64], RBP
;       D08:        7402             JEQ L7

;       D0A:        CC09             BREAK 9                      ; pending interrupt
 trap
;       D0C: L7:    488959F9         MOV [RCX-7], RBX
;       D10:        48895101         MOV [RCX+1], RDX

;       D14:        488B1575FEFFFF   MOV RDX, [RIP-395]           ; 'INTERNED-SYMBOL-
LIST
;       D1B:        488B4221         MOV RAX, [RDX+33]
;       D1F:        49833C0461       CMP QWORD PTR [R12+RAX], 97
;       D24:        7406             JEQ L8
;       D26:        49890C04         MOV [R12+RAX], RCX
;       D2A:        EB04             JMP L9
;       D2C: L8:    48894AF9         MOV [RDX-7], RCX

;       D30: L9:    488B0559FEFFFF   MOV RAX, [RIP-423]           ; 'INTERNED-SYMBOL-
LIST
;       D37:        488B4821         MOV RCX, [RAX+33]
;       D3B:        498B0C0C         MOV RCX, [R12+RCX]
```

```
;       D3F:       4883F961         CMP RCX, 97
;       D43:       7504             JNE L10
;       D45:       488B48F9         MOV RCX, [RAX-7]
;       D49: L10:  4883F951         CMP RCX, 81
;       D4D:       7455             JEQ L16
;       D4F:       8BC1             MOV EAX, ECX
;       D51:       240F             AND AL, 15
;       D53:       3C07             CMP AL, 7
;       D55:       7552             JNE L17
;       D57:       488B51F9         MOV RDX, [RCX-7]
;       D5B:       E9CCFEFFFF       JMP L0
;       D60:       CC0A             BREAK 10                         ; error trap
;       D62:       02               BYTE #X02

;       D63:       18               BYTE #X18                        ; INVALID-ARG-COUNT⚬
⚬-ERROR
;       D64:       54               BYTE #X54                        ; RCX
;       D65: L11:  CC0A             BREAK 10                         ; error trap
;       D67:       04               BYTE #X04

;       D68:       02               BYTE #X02                        ; OBJECT-NOT-LIST-E⚬
⚬RROR
;       D69:       FE9501           BYTE #XFE, #X95, #X01            ; RSI
;       D6C: L12:  CC0A             BREAK 10                         ; error trap
;       D6E:       02               BYTE #X02

;       D6F:       02               BYTE #X02                        ; OBJECT-NOT-LIST-E⚬
⚬RROR
;       D70:       55               BYTE #X55                        ; RCX
;       D71: L13:  6A20             PUSH 32
;       D73:       4C8D1C2570724200 LEA R11, [#x427270]              ; alloc_tramp
;       D7B:       41FFD3           CALL R11
;       D7E:       5B               POP RBX
;       D7F:       488D5B07         LEA RBX, [RBX+7]
;       D83:       E912FFFFFF       JMP L3
;       D88: L14:  CC0A             BREAK 10                         ; error trap
;       D8A:       02               BYTE #X02

;       D8B:       1A               BYTE #X1A                        ; UNBOUND-SYMBOL-ER⚬
⚬ROR
;       D8C:       15               BYTE #X15                        ; RAX
;       D8D: L15:  6A10             PUSH 16
;       D8F:       4C8D1C2570724200 LEA R11, [#x427270]              ; alloc_tramp
;       D97:       41FFD3           CALL R11
;       D9A:       59               POP RCX
;       D9B:       488D4907         LEA RCX, [RCX+7]
;       D9F:       E95FFFFFFF       JMP L6
;       DA4: L16:  CC0A             BREAK 10                         ; error trap
;       DA6:       02               BYTE #X02

;       DA7:       1A               BYTE #X1A                        ; UNBOUND-SYMBOL-ER⚬
⚬ROR
;       DA8:       15               BYTE #X15                        ; RAX
;       DA9: L17:  CC0A             BREAK 10                         ; error trap
;       DAB:       02               BYTE #X02
```

```
;      DAC:       02                 BYTE #X02                      ; OBJECT-NOT-LIST-E
RROR
;      DAD:       55                 BYTE #X55                      ; RCX
```

(See Open coded primitives (p. 283) for a more complete dissection of some SBCL output.)

I'm not quite sure where to start with this. This looks like a type test, with the type tag 7 in the low four bits of a pointer, and the place it's jumping to claims to signal an "OBJECT-NOT-LIST-ERROR":

```
;      C00:       8BC6               MOV EAX, ESI
;      C02:       240F               AND AL, 15
;      C04:       3C07               CMP AL, 7
;      C06:       0F8559010000       JNE L11
```

So I guess SBCL is hoisting a type test from the various conditional branches, all of which demand that the symlist be either null or a cons, up to the top of the function. Actually the null test may be the thing above, so this might just be a pair test:

```
; 02A2FBF7:       4881FE17001020     CMP RSI, 537919511             ; no-arg-parsing en
try point
;      BFE:       744B               JEQ L2
```

Maybe 537919511 (0x20100017) is SBCL's representation of NIL, and RSI is the second argument. If so, it looks like it would pass that LISTP test too, ending in a 7 as it does. The consequent of that test is this horrendous basic block, which looks like it's doing the right thing as it starts by allocating some stack space and loading in NEW-LABEL in RAX. Why SBCL opted to put this down near the end of the function I'm not sure; maybe it decided that `symlist` (RSI) was almost never going to be `NIL`.

```
;      C4B: L2:   488D5424F0         LEA RDX, [RSP-16]
;      C50:       4883EC18           SUB RSP, 24

;      C54:       488B052DFFFFFF     MOV RAX, [RIP-211]             ; #<FDEFINITION obj
ect for NEW-LABEL>
;      C5B:       31C9               XOR ECX, ECX
```

I guess that means ECX (not RCX!) has the argument count (0) we're going to pass to NEW-LABEL? I did something similar in Ur-Scheme, but it's nice to not have to pass and check argument counts at run time. (Notice that `interning` doesn't in fact check its argument count!)

```
;      C5D:       48892A             MOV [RDX], RBP
```

That's *storing* RBP into the stack; this is Intel operand order.

```
;      C60:       488BEA             MOV RBP, RDX
```

That way it can overwrite it with the new frame pointer. Why it's allocating a new frame in the middle of a function I don't know.

```
;       C63:       FF5009           CALL QWORD PTR [RAX+9]
```

Presumably this is loading the pointer to NEW-LABEL's code from 9 bytes past the beginning of its FDEFINITION object, not actually jumping there.

```
;       C66:       480F42E3         CMOVB RSP, RBX
```

Now, I'm not sure what's going on *here*. We expect NEW-LABEL to have passed us back some kind of return status in the flags? So that we can decide whether or not to ... clobber the stack pointer with RBX? Maybe this is how SBCL handles errors? Anyway, then we set up some registers from things stored in the stack frame, one of which is presumably NEW-LABEL's return value:

```
;       C6A:       488B75F0         MOV RSI, [RBP-16]
;       C6E:       4C8B45F8         MOV R8, [RBP-8]
;       C72:       488BFA           MOV RDI, RDX
;       C75:       498BD0           MOV RDX, R8
;       C78:       49896C2440       MOV [R12+64], RBP
```

Okay, now we're saving our frame pointer at an offset from... R12?

```
;       C7D:       4D8B5C2418       MOV R11, [R12+24]
;       C82:       498D5B20         LEA RBX, [R11+32]
;       C86:       49395C2420       CMP [R12+32], RBX
;       C8B:       0F86E0000000     JBE L13
```

Okay, at this point I have no idea what's going on.

```
;       D71: L13:  6A20             PUSH 32
;       D73:       4C8D1C2570724200 LEA R11, [#x427270]         ; alloc_tramp
;       D7B:       41FFD3           CALL R11
;       D7E:       5B               POP RBX
;       D7F:       488D5B07         LEA RBX, [RBX+7]
;       D83:       E912FFFFFF       JMP L3
```

Ohhh, it was checking to see if the nursery was full. I guess R11 is the allocation limit, and R12 is the allocation pointer. Maybe 32 is how many bytes we're going to allocate. So if we ran out of space in the nursery we invoke a minor GC before continuing:

```
;       C8B:       0F86E0000000     JBE L13                     ; (duplicated conte
xt instruction above)
;       C91:       49895C2418       MOV [R12+24], RBX
;       C96:       498D5B07         LEA RBX, [R11+7]
```

So maybe it's actually RBX that's the allocation pointer? I don't

know. But I think we're in the middle of an open-coded implementation of two-argument LIST.

```
;       C9A: L3:   488BC3           MOV RAX, RBX
;       C9D:       488950F9         MOV [RAX-7], RDX
```

So now our newly allocated cons pointer is in RAX. 16 sounds like the size of a dotted pair on a 64-bit machine. So we stored… RDX into it? That originally came from `[RBP-8]`, so maybe it was new-label's return value.

```
;       CA1:       4883C010         ADD RAX, 16
;       CA5:       488940F1         MOV [RAX-15], RAX
;       CA9:       488978F9         MOV [RAX-7], RDI
```

Okay, so we're creating a second cons, and poking its address into the CDR of the previously allocated one, and now poking RDI (maybe our first argument, `symbol`?) into the car.

```
;       CAD:       48C7400117001020 MOV QWORD PTR [RAX+1], 537919511
```

And this is our magic number again that might mean NIL, and we're poking it into what looks like the CDR of our second cons. One problem with the above hypotheses: the putative `symbol` and putative new label are in the wrong order in the list.

```
;       CB5:       49316C2440       XOR [R12+64], RBP
;       CBA:       7402             JEQ L4

;       CBC:       CC09             BREAK 9                     ; pending interrupt○
○ trap
```

Aha, this is some runtime safety code to verify that RBP still (or again) has the same value it had when we saved it earlier … because if it doesn't, we have a pending interrupt? Anwyay, normally we'll jump over that `BREAK` and into the code that sets up to call `CONS`:

```
;       CBE: L4:   488B05CBFEFFFF   MOV RAX, [RIP-309]          ; 'INTERNED-SYMBOL-○
○LIST
;       CC5:       488B5021         MOV RDX, [RAX+33]
;       CC9:       498B1414         MOV RDX, [R12+RDX]
;       CCD:       4883FA61         CMP RDX, 97
;       CD1:       7504             JNE L5
```

So we're using PC-relative addressing to get to the value cell of this global variable, and then we … follow a couple of pointers in ways I don't understand, and expect to get 97. What is that? The ASCII code for `? If that's not what we got, we jump to L5. L5?

```
;       CD1:       7504             JNE L5
;       CD3:       488B50F9         MOV RDX, [RAX-7]
;       CD7: L5:   4883FA51         CMP RDX, 81
;       CDB:       0F84A7000000     JEQ L14
```

Well apparently if it wasn't 97, now we look someplace else to see if we have an 81 (ASCII `Q`)? I'm not sure what all this has to do with consing the new list onto `interned-symbol-list` and sticking it in `interned-symbol-list`?

```
;       D88: L14:  CC0A            BREAK 10                  ; error trap
;       D8A:       02              BYTE #X02

;       D8B:       1A              BYTE #X1A                 ; UNBOUND-SYMBOL-ER
ROR
;       D8C:       15              BYTE #X15                 ; RAX
```

Oh, apparently the value 81 in the word before a symbol's value cell signifies that the signal is unbound? Because that is a possibility, after all. And the next thing we need to do is get the actual value of the symbol to pass it to cons. Still, I'm not sure why it makes sense to jump from finding that RDX is 97 to checking to see whether it's 81. So, what does the happy path look like?

```
;       CDB:       0F84A7000000    JEQ L14
;       CE1:       49896C2440      MOV [R12+64], RBP
;       CE6:       4D8B5C2418      MOV R11, [R12+24]
;       CEB:       498D4B10        LEA RCX, [R11+16]
;       CEF:       49394C2420      CMP [R12+32], RCX
;       CF4:       0F8693000000    JBE L15
```

This looks like another open-coded allocation with R11 and R12, and saving off a copy of RBP again to check later if it's been scroggled. But now we're using RCX, so evidently the allocation pointers aren't kept persistently in registers; they live at these weird offsets from R11 and R12. Actually maybe just from R12, since that's where we loaded R11 from. (R12 remains sacrosanct throughout.)

```
;       D8D: L15:  6A10            PUSH 16
;       D8F:       4C8D1C2570724200 LEA R11, [#x427270]      ; alloc_tramp
;       D97:       41FFD3          CALL R11
;       D9A:       59              POP RCX
;       D9B:       488D4907        LEA RCX, [RCX+7]
;       D9F:       E95FFFFFFF      JMP L6
```

That seems right, although it seems kind of goofy to have all this duplicated machine code hanging around, differing only in the registers used and the address we jump back to. So, what if allocation succeeded?

```
;       CF4:       0F8693000000    JBE L15
;       CFA:       49894C2418      MOV [R12+24], RCX
;       CFF:       498D4B07        LEA RCX, [R11+7]
;       D03: L6:   49316C2440      XOR [R12+64], RBP
;       D08:       7402            JEQ L7

;       D0A:       CC09            BREAK 9                   ; pending interrupt
 trap
```

In the happy path we've stored [R12+24]+16 back into [R12+24], thus allocating an additional cons cell. We're gonna open-code CONS! And now we overwrite RCX with, I guess, an address somewhere in the middle of the cons. And check again to see if our base pointer has gotten cabbaged.

(I think the weird +7 stuff is because 7 is the type tag for dotted pairs and NIL (and maybe value cells for global variables too?), so SBCL just uses immediate offsets of +1 and -7 to access the CDR and CAR respectively.)

I'm guessing that what comes next is that we're going to store interned-symbol-list's value into the CDR of our new cell, and the LIST we constructed earlier (or maybe its reversal?) into the CAR?

```
;       D0C: L7:   488959F9         MOV [RCX-7], RBX
;       D10:       48895101         MOV [RCX+1], RDX

;       D14:       488B1575FEFFFF   MOV RDX, [RIP-395]          ; 'INTERNED-SYMBOL-
LIST
;       D1B:       488B4221         MOV RAX, [RDX+33]
;       D1F:       49833C0461       CMP QWORD PTR [R12+RAX], 97
;       D24:       7406             JEQ L8
```

Bingo! RBX, remember, was set to R11+7 when we were allocating the list. And RDX was the thing we fetched from INTERNED-SYMBOL-LIST previously, which we were concerned might be 81 or 97. (It wasn't 81.) Now we're checking again to see if something or other related to interned-symbol-list is 97 for some reason. Why? We already have its value! It's already in the CDR!

```
;       D26:       49890C04         MOV [R12+RAX], RCX
;       D2A:       EB04             JMP L9
;       D2C: L8:   48894AF9         MOV [RDX-7], RCX

;       D30: L9:   488B0559FEFFFF   MOV RAX, [RIP-423]          ; 'INTERNED-SYMBOL-
LIST
;       D37:       488B4821         MOV RCX, [RAX+33]
;       D3B:       498B0C0C         MOV RCX, [R12+RCX]
;       D3F:       4883F961         CMP RCX, 97
;       D43:       7504             JNE L10
```

Well, I guess if [R12+RAX] wasn't 97, we're going to store our new CONS pointer there. But if it was, we're going to store the new list pointer back at interned-symbol-list.

Hmm, maybe this 97 thing is some sort of garbage collector write barrier?

Anyway, then we *again* check to see if something indexed off RCX related to interned-symbol-list is 97.

```
;       D43:       7504             JNE L10
;       D45:       488B48F9         MOV RCX, [RAX-7]
;       D49: L10:  4883F951         CMP RCX, 81
;       D4D:       7455             JEQ L16
```

I feel like I'm in a loop. This is Groundhog Day. Time has no meaning. Now we're loading the value of `interned-symbol-list` *again* to see if it's 81, *again*. Even though we just set it. To an address that probably ends in a 7.

```
;       DA4: L16:  CC0A            BREAK 10                   ; error trap
;       DA6:       02              BYTE #X02

;       DA7:       1A              BYTE #X1A                  ; UNBOUND-SYMBOL-ER◦
◦ROR
;       DA8:       15              BYTE #X15                  ; RAX
```

And if it was 81, we again have an `UNBOUND-SYMBOL-ERROR`. But how does that make any sense? Ohhh, now we're past the mutation! Now we're in the next line of code, where we return its `CAR`!

Well, if so, we should fetch from `[RCX-7]` next, then return.

```
;       D4F:       8BC1            MOV EAX, ECX
;       D51:       240F            AND AL, 15
;       D53:       3C07            CMP AL, 7
;       D55:       7552            JNE L17
;       D57:       488B51F9        MOV RDX, [RCX-7]
;       D5B:       E9CCFEFFFF      JMP L0
```

Yes! We fetched the `CAR`! And we put it in `RDX`! After checking the type tag to make sure it was a list. That is what we were checking, right?

```
;       DA9: L17:  CC0A            BREAK 10                   ; error trap
;       DAB:       02              BYTE #X02

;       DAC:       02              BYTE #X02                  ; OBJECT-NOT-LIST-E◦
◦RROR
;       DAD:       55              BYTE #X55                  ; RCX
```

Yup. Okay, what's L0? Are we going to clean up our stack frame and return?

```
;       C2C: L0:   488BE5          MOV RSP, RBP
;       C2F:       F8              CLC
;       C30:       5D              POP RBP
;       C31:       C3              RET
```

Yup, exactly. Also we carefully clear the carry flag first, in case our caller is going to pull the same kind of `CMOVB` trick we did when we called `NEW-LABEL`. Apparently leaving the carry flag set means some kind of exceptional condition or something in SBCL's calling convention. So that was the compilation of these four lines of code:

```
((null symlist)
 (setq interned-symbol-list
       (cons (list symbol (new-label)) interned-symbol-list))
 (car interned-symbol-list))
```

So, how about if it *wasn't* nil? Back to the top:

```
; 02A2FBF7:       4881FE17001020   CMP RSI, 537919511         ; no-arg-parsing en⤶
⤷try point
;      BFE:       744B             JEQ L2
;      C00:       8BC6             MOV EAX, ESI
;      C02:       240F             AND AL, 15
;      C04:       3C07             CMP AL, 7
;      C06:       0F8559010000     JNE L11
```

So, once we get past the `JNE L11` guardian of the bridge, we know RSI is a list, or at any rate NIL or a dotted pair. Our next case is the name-match case:

```
((eq symbol (caar symlist))
 (car symlist))
```

I suppose now we're going to fetch `[RSI-7]` (the `CAR`), do the same type-test on it, and then return its `CAR`?

```
;      C0C:       488BC6           MOV RAX, RSI
;      C0F:       488B48F9         MOV RCX, [RAX-7]
;      C13:       8BC1             MOV EAX, ECX
;      C15:       240F             AND AL, 15
;      C17:       3C07             CMP AL, 7
;      C19:       0F854D010000     JNE L12
;      C1F:       488B49F9         MOV RCX, [RCX-7]
;      C23:       4939C8           CMP R8, RCX
;      C26:       750A             JNE L1
```

Looks like it, although there's an extra apparently useless register move in there. (`L12` is indeed another `OBJECT-NOT-LIST-ERROR` trap). And now we're using an open-coded `eq` to check that `caar` against... R8? Could it be that `symbol`, our first argument, gets passed in R8? But in the other branch we overwrote R8 without checking, and used the value in RDX... hmm, well, what happens next if they *are* equal?

```
;      C28:       488B56F9         MOV RDX, [RSI-7]
;      C2C: L0:   488BE5           MOV RSP, RBP
;      C2F:       F8               CLC
;      C30:       5D               POP RBP
;      C31:       C3               RET
```

We load the `CAR` from `RSI` (`symlist`), which we've already verified was a list so we don't need to check again, and fall into the return path, returning `RDX` as before.

So that's what SBCL has decided is the "happy path" for the function; it's only 20 instructions. Unfortunately SBCL is wrong about this; this is just the second of the two base cases for the recursion (search failure, then search success). The inner recursive loop is the part we haven't seen yet:

```
    (t
     (interning symbol (cdr symlist)))))
```

Starting from L1:

```
;      C32: L1:   488B7E01         MOV RDI, [RSI+1]
;      C36:       498BD0           MOV RDX, R8

;      C39:       488B0540FFFFFF   MOV RAX, [RIP-192]         ; #<FDEFINITION obj⚬
⚬ect for INTERNING>
;      C40:       B904000000       MOV ECX, 4
;      C45:       FF7508           PUSH QWORD PTR [RBP+8]
;      C48:       FF6009           JMP QWORD PTR [RAX+9]
```

...that's... it? We load an argument count of 4 (‽ Maybe it's doubled?) into ECX, set up the function object in RAX in case the function is a closure, set RDI to be the CDR of RSI, set RDX to be R8 (which apparently is somehow symbol, our first argument). Then we push something on the stack (‽) and do a tail call by making a jump. So apparently whatever lurks at that address is expecting its arguments in (RDX, RDI), not (RSI, R8) as we received them. But this whole tail call is only 6 instructions.

So the inner recursive loop is 21 instructions, and the likely initial misprediction due to SBCL getting the cases in the wrong order probably only costs a single branch prediction error per program restart, before the CPU learns to guess the branch correctly.

Of the total of 131 assembly instructions, in fact, only 92 are on any of the three control paths that are present in the source; the other 39 instructions and pseudo-instructions are exception handlers tagged onto the end of the function in case of type errors or running out of memory.

Wait! That's not all! It also included this bit of dead code in case the arg count is wrong. (Maybe some code somewhere else jumps to it.)

```
;      D60:       CC0A             BREAK 10                   ; error trap
;      D62:       02               BYTE #X02

;      D63:       18               BYTE #X18                  ; INVALID-ARG-COUNT⚬
⚬-ERROR
;      D64:       54               BYTE #X54                  ; RCX
```

Upon reading on the web, it seems that SBCL doesn't have user-level write barriers, so I'm not sure what this 97 stuff is. The SBCL Internals Manual has a Calling Convention section which probably answers some of the questions I had above; in particular Full Calls explains the calling convention:

> Basically, we use caller-allocated frames, pass an fdefinition, function, or closure in EAX, argcount in ECX, and first three args in EDX, EDI, and ESI. EBP points to just past the start of the frame (the first frame slot is at [EBP-4], not the traditional [EBP], due in part to how the frame allocation works). The caller stores the link for the old frame at [EBP-4] and reserved space for a return address at [EBP-8]. [EBP-12] appears to be an empty slot that conveniently makes just enough space for the first three multiple return values (returned in the argument passing registers) to be written

over the beginning of the frame by the receiver. The first stack argument is at [EBP-16]. The callee then reallocates the frame to include sufficient space for its local variables, after possibly converting any &rest arguments to a proper list. ... The above scheme was changed in 1.0.27 on x86 and x86-64 by swapping the old frame pointer with the return address and making EBP point two words later:

No idea what's going on with R8 still.

Also, "Unknown-Values Returns" explains the carry-flag thing:

For a single-value return, we load the return value in the first argument-passing register (A0, or EDI), reload the old frame pointer, burn the stack frame, and return. The old convention was to increment the return address by two before returning, typically via a JMP, which was guaranteed to screw up branch-prediction hardware. The new convention is to return with the carry flag clear.

For a multiple-value return, we pass the first three values in the argument-passing registers, and the remainder on the stack. ECX contains the total number of values as a fixnum, EBX points to where the callee frame was, EBP has been restored to point to the caller frame, and the first of the values on the stack (the fourth overall) is at [EBP-16]. The old convention was just to jump to the return address at this point. The newer one has us setting the carry flag first.

The code at the call site for accepting some number of unknown- values is fairly well boilerplated. If we are expecting zero or one values, then we need to reset the stack pointer if we are in a multiple-value return. In the old convention we just encoded a MOV ESP, EBX instruction, which neatly fit in the two byte gap that was skipped by a single-value return. In the new convention we have to explicitly check the carry flag with a conditional jump around the MOV ESP, EBX instruction.

But I suppose the CMOVB is a better alternative.

Since interned-symbol-list is essentially a private variable for interning, I added a declaration to tell SBCL to not worry so much about type checking, although maybe truly-the is the real ticket:

```
(defvar interned-symbol-list '())

(defun new-label ()
  (gensym))

(defun interning (symbol symlist)
  (declare (optimize (safety 0)))
  (cond ((null symlist)
         (setq interned-symbol-list
               (cons (list symbol (new-label)) interned-symbol-list))
         (car interned-symbol-list))
        ((eq symbol (caar symlist))
         (car symlist))
        (t
         (interning symbol (cdr symlist)))))
```

Now the whole function is only 90 instructions, and the short path (success) is down to 11 instructions, and the inner loop (the recursive path) 12 instructions:

```
; disassembly for INTERNING

; 029F972D:       4881FE17001020   CMP RSI, 537919511          ; no-arg-parsing en
try point
;       734:       7430             JEQ L2

;       736:       488B46F9         MOV RAX, [RSI-7]
```

```
;      73A:       488B48F9         MOV RCX, [RAX-7]
;      73E:       4939C8           CMP R8, RCX
;      741:       750A             JNE L1

;      743:       488B56F9         MOV RDX, [RSI-7]

;      747: L0:   488BE5           MOV RSP, RBP
;      74A:       F8               CLC
;      74B:       5D               POP RBP
;      74C:       C3               RET

;      74D: L1:   488B7E01         MOV RDI, [RSI+1]
;      751:       498BD0           MOV RDX, R8

;      754:       488B0565FFFFFF   MOV RAX, [RIP-155]           ; #<FDEFINITION obj⚬
⚬ect for INTERNING>
;      75B:       B904000000       MOV ECX, 4
;      760:       FF7508           PUSH QWORD PTR [RBP+8]
;      763:       FF6009           JMP QWORD PTR [RAX+9]

;      766: L2:   488D5424F0       LEA RDX, [RSP-16]
;      76B:       4883EC18         SUB RSP, 24

;      76F:       488B0552FFFFFF   MOV RAX, [RIP-174]           ; #<FDEFINITION obj⚬
⚬ect for NEW-LABEL>
;      776:       31C9             XOR ECX, ECX
;      778:       48892A           MOV [RDX], RBP
;      77B:       488BEA           MOV RBP, RDX
;      77E:       FF5009           CALL QWORD PTR [RAX+9]

;      781:       480F42E3         CMOVB RSP, RBX
;      785:       488B75F0         MOV RSI, [RBP-16]
;      789:       4C8B45F8         MOV R8, [RBP-8]
;      78D:       488BFA           MOV RDI, RDX
;      790:       498BD0           MOV RDX, R8
;      793:       49896C2440       MOV [R12+64], RBP
;      798:       4D8B5C2418       MOV R11, [R12+24]
;      79D:       498D5B20         LEA RBX, [R11+32]
;      7A1:       49395C2420       CMP [R12+32], RBX
;      7A6:       0F86B3000000     JBE L11

;      7AC:       49895C2418       MOV [R12+24], RBX
;      7B1:       498D5B07         LEA RBX, [R11+7]

;      7B5: L3:   488BC3           MOV RAX, RBX
;      7B8:       488950F9         MOV [RAX-7], RDX
;      7BC:       4883C010         ADD RAX, 16
;      7C0:       488940F1         MOV [RAX-15], RAX
;      7C4:       488978F9         MOV [RAX-7], RDI
;      7C8:       48C7400117001020 MOV QWORD PTR [RAX+1], 537919511
;      7D0:       49316C2440       XOR [R12+64], RBP
;      7D5:       7402             JEQ L4


;      7D7:       CC09             BREAK 9                      ; pending interrupt⚬
```

```
⚬ trap

;      7D9: L4:   488B05F0FEFFFF   MOV RAX, [RIP-272]          ; 'INTERNED-SYMBOL-⚬
⚬LIST
;      7E0:       488B5021         MOV RDX, [RAX+33]
;      7E4:       498B1414         MOV RDX, [R12+RDX]
;      7E8:       4883FA61         CMP RDX, 97
;      7EC:       7504             JNE L5

;      7EE:       488B50F9         MOV RDX, [RAX-7]

;      7F2: L5:   49896C2440       MOV [R12+64], RBP
;      7F7:       4D8B5C2418       MOV R11, [R12+24]
;      7FC:       498D4B10         LEA RCX, [R11+16]
;      800:       49394C2420       CMP [R12+32], RCX
;      805:       766F             JBE L12
;      807:       49894C2418       MOV [R12+24], RCX
;      80C:       498D4B07         LEA RCX, [R11+7]

;      810: L6:   49316C2440       XOR [R12+64], RBP
;      815:       7402             JEQ L7


;      817:       CC09             BREAK 9                     ; pending interrupt⚬
⚬ trap

;      819: L7:   488959F9         MOV [RCX-7], RBX
;      81D:       48895101         MOV [RCX+1], RDX

;      821:       488B15A8FEFFFF   MOV RDX, [RIP-344]          ; 'INTERNED-SYMBOL-⚬
⚬LIST
;      828:       488B4221         MOV RAX, [RDX+33]
;      82C:       49833C0461       CMP QWORD PTR [R12+RAX], 97
;      831:       7406             JEQ L8

;      833:       49890C04         MOV [R12+RAX], RCX
;      837:       EB04             JMP L9

;      839: L8:   48894AF9         MOV [RDX-7], RCX


;      83D: L9:   488B0D8CFEFFFF   MOV RCX, [RIP-372]          ; 'INTERNED-SYMBOL-⚬
⚬LIST
;      844:       488B4121         MOV RAX, [RCX+33]
;      848:       498B0404         MOV RAX, [R12+RAX]
;      84C:       4883F861         CMP RAX, 97
;      850:       7504             JNE L10

;      852:       488B41F9         MOV RAX, [RCX-7]

;      856: L10:  488B50F9         MOV RDX, [RAX-7]
;      85A:       E9E8FEFFFF       JMP L0

;      85F: L11:  6A20             PUSH 32
```

```
;       861:        4C8D1C2570724200 LEA R11, [#x427270]       ; alloc_tramp
;       869:        41FFD3           CALL R11

;       86C:        5B               POP RBX
;       86D:        488D5B07         LEA RBX, [RBX+7]
;       871:        E93FFFFFFF       JMP L3

;       876: L12:   6A10             PUSH 16
;       878:        4C8D1C2570724200 LEA R11, [#x427270]       ; alloc_tramp
;       880:        41FFD3           CALL R11

;       883:        59               POP RCX
;       884:        488D4907         LEA RCX, [RCX+7]
;       888:        EB86             JMP L6
```

I think this is mostly the same except for all the elided type checks.

Topics

- Programming (p. 1141) (49 notes)
- Performance (p. 1155) (22 notes)
- Experiment report (p. 1162) (14 notes)
- Python (p. 1166) (12 notes)
- Safe programming languages (p. 1172) (11 notes)
- Virtual machines (p. 1182) (9 notes)
- C (p. 1194) (8 notes)
- Instruction sets (p. 1214) (6 notes)
- FORTH (p. 1231) (5 notes)
- Bytecode (p. 1236) (5 notes)
- Scheme (p. 1274) (3 notes)
- The Veskeno virtual machine (p. 1313) (2 notes)
- Tcl (p. 1318) (2 notes)
- Perl (p. 1342) (2 notes)
- m4 (p. 1352) (2 notes)
- Lua (p. 1354) (2 notes)
- The JS programming language (p. 1359) (2 notes)
- The Bourne shell
- Java
- Elisp

Vaughan Pratt and Henry Baker's COMFY control-flow combinators

Kragen Javier Sitaker, 02021-03-04 (updated 02021-03-20) (8 minutes)

I read Henry Baker's columns about COMFY-65 and COMFY-80 and found them very interesting. He credits the approach to Pratt in the early 01970s, which is pretty interesting; combined with top-down operator precedence — not PEG or TDPL, but a different linear-time top-down parsing algorithm with better space and time bounds — it seems like Pratt totally reinvented compilers in a different way in the 01970s and nobody noticed!

The COMFY-65 approach is "structured control flow" in the sense that larger pieces of code are built up from smaller pieces of code using a fixed set of forms of composition, but they aren't the same forms as the Kleene set of sequence, alternation, and closure used in the Böhm-Jacopini theorem. The key difference is that Pratt's/Baker's pieces of code have one entry and two exits ("win" and "lose") instead of one, so they can directly represent arbitrary control-flow graphs. In COMFY-65 his forms of composition are `seq`, `alt`, `if`, `while`, `not`, and `loop`, which are similar to the usual constructs but not the same.

Re-expressing the COMFY approach with constraint satisfaction

I suspect that constraint satisfaction is a productive way to formulate compilation problems in general, so here's an attempt to formulate Baker's constructs logically as sets of constraints in a hypothetical hierarchical constraint language supporting the definition of infix operators, using `yes` and `no` rather than `win` and `lose`. Starting with `loop`:

```
(loop X):
    entry = X.entry = X.yes
    no = X.no
```

This is an abbreviation for

```
(loop X).entry = X.entry = X.yes
(loop X).no = X.no
```

The remaining constructs can be defined as follows:

```
(A; B):                 # sequence
    entry = A.entry
    A.yes = B.entry
    yes = B.yes
    no = A.no = B.no
```

```
(A || B) = !(!A; !B)   # alternation

(!X):                  # negation
    entry = X.entry
    yes = X.no
    no = X.yes

(A ? B : C):           # general conditional
    entry = A.entry
    A.yes = B.entry
    A.no = C.entry
    no = B.no = C.no

(while A: B):
    entry = A.entry
    A.yes = B.entry
    B.yes = A.entry
    yes = A.no
    no = B.no
```

Baker's compiler compiles these constructs recursively, providing `yes` and `no` (`win` and `lose`) as concrete numbers when `compile` is invoked. It does this by filling memory with instructions starting from the end, such that each of these exit points (the `yes` and `no` items) generally refer to things that have previously been compiled and will follow them in memory — possibly immediately, in which case manifesting the required control flow doesn't require a jump instruction. In that context, though, I'm not totally sure how he handles loops, which necessarily need a jump back to the beginning of the loop, which is at a location unknown when the loop body is being compiled. I need to look at his code.

But not right now!

An alternate definition of `alt`:

```
(A || B):
    entry = A.entry
    A.no = B.entry
    no = B.no
    yes = A.yes = B.yes
```

Given a program `pass` which does nothing and just passes control to its `yes`, we could define `loop` as:

```
loop X = (while pass: X)
```

We could define `fail`, which does nothing and just passes control to its `no`, as `!pass`, we could define sequencing as a conditional:

```
(A; B) = (A ? B : fail)
```

Here's a totally revised, terser version, using some symbols from Baker's 1976 note, the above constructions, a Python-like respelling

of the conditional, and C-like blocks instead of Python indentation. I'm also changing "entry" to "go" (on your mark, get set,...) and defining ¬X in terms of the conditional, as Darius did in his "Language of Choice". Actually, it occurs to me that COMFY is very closely related to binary decision diagrams, but its control graph isn't constrained to be acyclic...

```
pass            { go = yes }
fail            { go = no }    # do not pass go
(B if A else C) { go = A.go, A.yes = B.go, A.no = C.go, no = B.no = C.no }
(while A: B)    { go = A.go, A.yes = B.go, B.yes = A.go, yes = A.no, no = B.no }

(¬X)      = (fail if X else pass)
(A; B)    = (B    if A else fail)
(A / B)   = (pass if A else B)
(X∞)      = (while pass: X)
```

I was going to try to unpack a small program in this notation into a flat pile of constraints, but I'm getting confused about the class/instance distinction when I try.

A particular instruction might fit into this framework in a form like the following:

```
pushq_rbp { mem[go] = 0x55, yes = go + 1 }
```

(Since `pushq %rbp` doesn't use its `no` exit, it imposes no constraints on `no`.)

More generally, I think straight-line code as cons-lists of bytes can be emitted as follows:

```
straight([])   = pass
straight(A::B) { mem[go] = A, T = straight(B), T.go = go + 1, yes = T.yes }
```

There's a semantic lacuna in the above, though; something like `pushq_rbp / pushq_rbp` will fail, because there's no place to insert the necessary jumps. (Also, because `pushq_rbp` can't fail, the second alternative could be entirely optimized out.)

Dissection of Baker's Elisp

Here's the part of Baker's Elisp code concerned with compiling loops. It turns out to work more or less the way you'd expect if you know it compiles things recursively, starting from the end of memory:

```
(defun compile (e win lose)
  (cond ((eq (car e) 'loop)
         (let* ((l (genbr 0)) (r (compile (cadr e) l lose)))
           (ra l r)
           r))
        ((eq (car e) 'while)              ; do-while.
         (let* ((l (genbr 0))
                (r (compile (cadr e)
                            (compile (caddr e) l lose)
```

```
                              win)))
          (ra l r)
          r))))
```

So, it begins by generating a branch to 0 at the end of the loop, saving the address of the jump instruction in `l`; then it compiles the contents of the loop (possibly including jumps to `lose` and, in the `while` case, `win`). It saves the address of the beginning of the loop in `r`, and then it runs `(ra l r)`:

```
(defun ra (b a)
  ;;; replace the absolute address at the instruction "b"
  ;;; by the address "a".
  (let* ((ha (lsh a -8)) (la (logand a 255)))
    (aset mem (1+ b) la)
    (aset mem (+ b 2) ha))
  b)
```

So this just backpatches the jump; `ha` and `la` are the high and low bytes of the address, and it stores them two bytes after and one byte after the address of the jump instruction, which is generated as follows:

```
(defun genbr (win)
  ;;; generate an unconditional jump to "win".
  (gen 0) (gen 0) (gen jmp) (ra f win))
```

And that's in terms of `gen`, which is:

```
(defun gen (obj)
  ;;; place one byte "obj" into the stream.
  (setq f (1- f))
  (aset mem f obj)
  f)
```

Modifying `mem` and `f`:

```
(defvar mem (make-vector 10 0)
  "Vector where the compiled code is placed.")

(setq mem (make-vector 100 0))

(defvar f (length mem)
  "Compiled code array pointer; it works its way down from the top.")
```

The semantic lacuna mentioned above is filled in this case with an explicit `genbr` call followed by the backpatching. But the `win` and `lose` addresses (which Baker calls continuations) are passed in to compile the snippets of code within the loop, which in many cases bottom out in `emit`, which begins by inserting a call to the `genbr` above if necessary:

```
(defun emit (i win)
  ;;; place the unconditional instruction "i" into the stream with
  ;;; success continuation "win".
```

```
(cond ((not (= win f)) (emit i (genbr win)))
      ...))
```

That is, if the next address to execute is not the address immediately following where we're going to be compiling, then first we `genbr` a jump so that it is.

Naming

What should you call something derived from Baker's system, but different? Antonyms for "comfy" include cold, cool, disagreeable, dissatisfied, hard, strict, troubled, uncomfortable, unfriendly, unhappy, unpleasant, unsuited, destitute, poor, discontented, needy, hopeless, miserable, neglected, pitiable, upset, and wretched, according to Roget's. Synonyms include snug, cozy, cushy, homey, and soft. Lojban has "kufra" for "foo is comfortable with bar".

Topics

- Programming (p. 1141) (49 notes)
- History (p. 1153) (24 notes)
- Lisp (p. 1174) (11 notes)
- Compilers (p. 1178) (10 notes)
- Program calculator (p. 1246) (4 notes)
- Control flow (p. 1299) (3 notes)
- COMFY-* (p. 1300) (3 notes)
- Constraints

Generating novel unique pronounceable identifiers with letter frequency data

Kragen Javier Sitaker, 02021-03-10 (updated 02021-03-22) (11 minutes)

From Project Gutenberg's edition of "War and Peace", we get etaonihsrdlucmwfgypbvkxqzj, which is pretty close to the etaonrishdlfcmug...qz I remember from Zim's "Codes and Secret Writing" and the etaoinshrdlu of the Linotype. Interestingly, capital letters are ITAPHNBMRSWECDFOGYKVLXJUZQ, a quite different distribution.

```
perl -lne '$f{$&}++ while /./g; END {for (sort {$f{$b} <=> $f{$a}} keys %f) { print "$_ $f{$_}" }}' war-and-peace-2600.txt
  514911
e 312990
t 219591
a 199239
o 191245
n 180561
i 166351
h 163027
s 159906
r 145373
d 116274
l 95814
u 65180
c 59520
m 58395
w 56319
f 52950
g 50024
y 45000
, 39891
p 39014
b 31052
. 30805
v 25970
k 19230
" 17970
I 7933
' 7529
T 6817
A 6575
P 6519
- 6308
H 4378
! 3923
x 3711
N 3614
```

```
B 3606
M 3251
? 3137
R 3057
S 2987
W 2888
q 2295
z 2280
j 2266
E 2259
C 2107
D 2017
F 1946
O 1635
G 1303
Y 1265
K 1201
; 1145
V 1116
: 1015
L 713
X 673
) 670
( 670
1 392
J 308
* 300
U 254
8 193
0 179
2 147
Z 108
3 61
6 57
5 55
7 40
Q 35
9 35
/ 29
4 23
$ 2
= 2
@ 2
[ 1
% 1
# 1
] 1
```

The first version of this Perl used `for` instead of `while` and consequently got the wrong answer.

It occurs to me that you could represent a pretty reasonable language model by dividing a suffix array of War and Peace into some 1024 to 16384 equal bins and listing the boundary elements of

the bins.

If you break the letters down by binary order of magnitude in frequency, you get etaonihs rdl ucmwfgy pbv k ' x qzj. Lower-case letters in etaonihs are about 3–4 bits of entropy; letters in rdl are about 4–5; ucmwfgy, 5–6; pbv, 6–7; k, 7–8; x, 9–10; qzj, 10–11. For capital letters, it's ITAPH NBMRSWECD FOGYKV LX JU O Z Q, I presumably being frequent because of its use in Roman numerals. The intersection of the least common 13 in both groups is fgyvkxqzj.

Generating novel strings for identifiers

This suggests that if you want to generate a sequence of letters that rarely occurs in English, letters like X and K buy you almost twice as much entropy as letters like C and D. However, it isn't enough to just say "qqqq", because although that's very unlikely to occur in English, it's fairly likely under other language models. And, for example, in base64-encoded gzipped data, "xzqj" occurs in one out of every 2^{24} positions, just like every other four-byte string consisting of valid base64 characters, rather than once every 2^{40} positions as the above naïve character frequency model would predict, or the even lower frequency a more accurate English model would predict.

Including digits and punctuation is a time-honored way of increasing apparent randomness, especially higher digits — though "8" and "0" in this corpus occur more often, because it has a lot of dates in the early 1800s, both "7" and "9" occur nearly an order of magnitude less commonly than "1", which is itself six times less common than any lowercase letter. Digits, though, also occur in base64 with frequency equal to that of letters. Including a punctuation character (other than + and /, the final two base64 digits, or , as in RFC3501, or - and _ as in RFC4648 §5) can avoid collisions in these cases, even if search engines aren't good at picking them up. The least objectionable candidates would seem to be ., : and !, though all of these are used by uuencode.

If there were 10 billion monkeys in the world constantly banging away on typewriters at 10 keystrokes per second each, evenly distributed over a 32-letter alphabet including our chosen glyphs, how long would it take them on average to produce a chosen string of any given length? That's 10 picoseconds per character:

length	probability	occurs every	example	existing meaning
1	2^{-5}	320 ps	y	years, a combinator, etc.
2	2^{-10}	10240 ps	yg	rapper, Korean record label, etc.
3	2^{-15}	0.0003 ms	ygf	graph format, record label, Yamaha guitar, etc.
4	2^{-20}	0.01 ms	ygf6	"yellow swim stormy flower" beach bag
5	2^{-25}	0.34 ms	ygf6v	occurs in uuencoded EDGAR filings and leaked email
6	2^{-30}	10.7 ms	ygf6vq	occurs in one EDGAR filing in 02018 and some URLs
7	2^{-35}	340 ms	ygf6vq6	an autogenerated email address
8	2^{-40}	11.0 s	ygf6vq62	a spammer domain
9	2^{-45}	6 minutes	ygf6vq624	no results
10	2^{-50}	3.1 hours	ygf6vq6244	
12	2^{-60}	130 days	ygf6vq624483	

14	2^{-70}	370 years	ygf6vq624483.v
15	2^{-75}	12000 years	ygf6vq624483.v2

In practice, we seem to escape from the existing human universe in this case around three or four characters, so we ought to be able to generate unique identifiers pretty reliably with four letters chosen from, say, "dfogykvlxjuzq", a digit that isn't 1, and a punctuation character chosen from ".:!". We have to be careful not to generate too many digits, because although digits occur at less than one in 8192 in my sample text above, they occur at 1 in 10 in other contexts, and so any five-digit number will occur in many places. In Python:

```
[m[:q] + random.choice('234567890') + m[q:]
 for _ in range(8)
 for m in [l[:p] + random.choice('.:!') + l[p:]
           for l in [''.join(random.choice('dfogykvlxjuzq')
                     for _ in range(5))]
           for p in [random.randrange(1, len(l))]]
 for q in [random.randrange(1, len(m))]]
```

This generates for example ['uly2q:k', 'k:9jzok', 'j!xjx9y', 'z9l!qjy', 'uoj5:dq', 'kdo:quy', 'xg!k6ou', 'gf!x2xj']. These do indeed seem to be unique, but search engines like Google and Startpage.com find false hits for them because they treat the punctuation as a word separator and find documents that spuriously contain each of the two "words". So a better approach for search engines is to put the punctuation always at the beginning:

```
[m[:q] + random.choice('234567890') + m[q:]
 for _ in range(8)
 for m in [random.choice('.:!') + l
           for l in [''.join(random.choice('dfogykvlxjuzq')
                     for _ in range(5))]]
 for q in [random.randrange(1, len(m))]]
```

This yields [':dxgouj', '.2glqud', '!2qkvdf', '.koyokg', '!xv8gdl', '.g3llgl', '.yd4uxu', ':vg2zqy']. Some of these yield no search-engine results, while others yield a small number of obviously spurious results; none of them can occur in base64 with the punctuation, and none of the search results have the punctuation either. The code as written can only produce $3 \times 5 \times 9 \times 13^5 = 50\,124\,555$ different identifiers, so it's not particularly random! You sure wouldn't want to use this to choose passwords. And if you use it to choose six or seven thousand identifiers, it'll probably produce a collision, thanks to the birthday paradox.

But evidently even four such letters and a digit are enough to escape from the universe of already-chosen global names:

```
[m[:q] + random.choice('234567890') + m[q:]
 for _ in range(4)
 for m in [random.choice('.:!') + l
           for l in [''.join(random.choice('dfogykvlxjuzq')
                     for _ in range(4))]]
```

```
 for q in [random.randrange(1, len(m)+1)]]
```

This yields ['.k5zyy', '!f2zxx', '.fyf9d', ':guu6x'] none of which seem to have an existing meaning.

If you want *pronounceable* random strings, the easiest approach is probably CV syllables, and you probably need a third vowel, such as `u`, and you probably want to eliminate `k` and `g` as possibly sounding the same as `q` and `j`:

```
[''.join(c+v for _ in range(n)
        for c in [random.choice('dfyvlxjzq')]
        for v in [random.choice('uoi')])
 for n in range(5)]
```

This has only 30 possible syllables, but yields 'zi' (master philosopher, ZoomInfo, zero infrastructure, a dozen different Wikipedia articles, etc.), 'jiji' (a Spanish laugh, a Nigerian classifieds site, a Taiwanese hostel system), 'vuqojo' (no occurrences, though there's a guy in Uganda named Pius Vukojo), 'qivufozo' (no occurrences found), and 'qiyilifizi' (no occurrences found). This suggests that three or four syllables of this form is enough to generate a unique name on most runs of this approach.

A slight improvement here might be to expand the consonants slightly and use `g` rather than `j`:

```
[''.join(c+v for _ in range(n)
        for c in [random.choice('wdfgypvlxzq')]
                  for v in [random.choice('uoi')])
 for n in range(5)]
```

This yields 'yu' (arrows, English second-person singular pronoun, and dozens of other meanings), 'qufo' (an Italian food retailer; also, there's an Albanian TV channel named Çufo), 'yiyogu' (the nickname of a person on Facebook, Yiyo Gu), 'qupizuqi' (apparently unique), and 'qilixopoqi' (apparently unique). This approach easily generates many somewhat-pronounceable apparently-unique names like `quzodupogu`, `zofopu`, `voxopidi`, `poyodipi`, `gopiwixu`, and `xiwoyifo`. Many of the three-syllable names it generates are taken, but few of the four-syllable names.

If we add an easy nasal syllable coda in some cases, we might get more unique names at three syllables:

```
[''.join(c+v+(random.choice('nm') if not random.randrange(3) else '')
        for _ in range(3)
        for c in [random.choice('wdfgypvlxzq')]
          for v in [random.choice('uoi')])
 for _ in range(5)]
```

This yields 'qozuqu' (apparently a Tajik and Georgian word),

'xinfowo' (unique), 'donxowom' (unique), 'lolomyom' (unique, though there's a tumor called a "lolomyoma", which I imagine is less amusing than it sounds), and 'dolonwu' (unique). (Some of these do appear in lists of randomly or exhaustively generated words similar to the above.)

Adding V syllables (with no onset) gives us:

```
[''.join(c+v+(random.choice('nm') if not random.randrange(3) else '')
         for _ in range(3)
         for c in [random.choice([''] + list('wdfgypvlxzq'))]
           for v in [random.choice('uoi')])
 for _ in range(5)]
```

This produces 'impufu' (a hill in South Africa and also a YouTuber), 'xonvulu' (unique), 'quqifin' (a village in Syria), 'zoliom' ("zona libre ordenanzas municipales"), and 'zinvixi' (unique). So this change seems to be counterproductive, perhaps unsurprising as it not only shortens the words but also increases the relative frequency of vowels in them, which are generally higher-frequency letters.

If we use a somewhat more difficult set of codas, we get

```
[''.join(random.choice('wdfgypvlxzq')
        +random.choice('uoi')
        +(random.choice('mfz') if not random.randrange(3) else '')
         for _ in range(3))
 for _ in range(5)]
```

This gives us 'xiqilo' (a brand of blinking sneakers), 'lofwili' (unique), 'yimlugu' (unique), 'figimzu' (unique), and 'gizxiwu' (unique but unpronounceable).

Topics

- Programming (p. 1141) (49 notes)
- Experiment report (p. 1162) (14 notes)
- Python (p. 1166) (12 notes)
- Natural-language processing (p. 1284) (3 notes)
- Randomness (p. 1336) (2 notes)
- Perl (p. 1342) (2 notes)

Garbage-collected allocation performance on current computers

Kragen Javier Sitaker, 02021-03-13 (updated 02021-04-08)
(4 minutes)

Originally posted at https://news.ycombinator.com/reply?id=26441466&goto=threads%3Fid%3Dkragen%2326441466

I guess you aren't trolling; you're just confusing the part of programming that you know about with the whole field. But there are more things in heaven and earth, Horatio, than are dreamt of in your philosophy…

You said, "If you allocate a a few bytes at a time, it will top out in the ballpark of 10 million per second per core." In my link above, I demonstrated a one-line program which, allocating a few bytes at a time, tops out at 150 million allocations per second per core; if the memory stays in use long enough to survive a minor GC, that drops to 100 million allocations per second per core. It's using the same allocator SBCL uses for everything (except large blocks), and these performance numbers include the time for deallocation. It takes into account all of the things that make memory allocation problematic for performance.† But it does an order of magnitude more allocations per second than you're saying is possible. Even LuaJIT 2.0.5 on this same laptop manages 42 ns per allocation, 23 million per second:

```
function nlist(n)
    local rv = nil
    for i = 1, n do
        rv = {i, rv}
    end
    return rv
end

function mnlist(m, n)
    print('m='..m..' n='..n)
    for i = 1, m do nlist(n) end
end

mnlist(500000, 2000)
```

It's true, as you say, that the way it works is similar to "adding [small numbers] to the size of a C++ vector with a large capacity on each iteration of a loop." (It's not "adding 1" because allocations of all sizes are served from the same nursery; interspersing different open-coded allocation sizes affects performance only a little.) But just doing that doesn't save you from writing a garbage collector and implementing write barriers, or alternatively doing MLKit-style static reasoning about lifetimes the way you do to allocate things on a per-frame heap in C++.

It's *also* true that, as you say, "memory allocation is often huge low

hanging fruit for optimization.” You aren’t going to get this kind of performance out of HotSpot or a malloc implementation, not even mimalloc. So if you’re using one of those systems, memory allocation is an order of magnitude more expensive. And, if you’re concerned about *worst-case* performance—latency, rather than throughput, as HFT people and AAA game programmers are—probably *no* kind of garbage collection is a good idea, and you may even need to avoid allocation altogether, although recent versions of HotSpot make some remarkable claims about worst-case latency, claims which may be true for all I know.

Of course, even when it comes to throughput, there is no free lunch. An allocator design so heavily optimized for fast allocation necessarily makes mutation more expensive—SBCL notoriously uses segfaults for its write barrier so that writes into the nursery are as fast as possible, a cost that would be intolerable for more mutation-oriented languages like Java or C++; and heavy mutation is generally a necessary evil if latency is important. (Also, I think there are algorithms, especially in numerical computation, where the best known mutation-based algorithms have a logarithmic speedup over the best known pure functional algorithms.)

You can find a more detailed discussion of some of the issues in https://archive.fo/itW87 (Martin Cracauer’s comparison of implementing memory allocation in LLVM and SBCL, including years of experience running SBCL in production and extensive discussion of the latency–throughput tradeoff I touch on above) and the mimalloc technical report, https://www.microsoft.com/en-us/research/publication/mimalloc-free-list-sharding-in-action/. The mimalloc report, among other things explains how they found that, for mimalloc, BBN-LISP-style per-page free-lists (Bobrow & Murphy 1966, AD647601, AFCRL-66-774) were faster than pointer-bumping allocation!

† This build of SBCL does have multithreading enabled, and the allocation benchmark takes the same amount of time running in a separate thread, but on my machine it doesn’t get a very good speedup if run in multiple threads, presumably due to some kind of allocator contention.

Topics

- Programming (p. 1141) (49 notes)
- Performance (p. 1155) (22 notes)
- Lisp (p. 1174) (11 notes)
- Real time (p. 1195) (7 notes)
- Facepalm (p. 1199) (7 notes)
- OCaml (p. 1249) (4 notes)
- Garbage collection (p. 1255) (4 notes)
- Allocation performance (p. 1308) (3 notes)
- Steel Bank Common Lisp (p. 1330) (2 notes)
- LuaJIT (p. 1353) (2 notes)

How Lao Yuxi painted a cock

Kragen Javier Sitaker, 02021-03-19 (updated 02021-04-14)
(7 minutes)

At first I did not understand this folk tale at all. I cannot trace its origin, though there are many tellings on the web, so I will add another version, which is of course a lie from beginning to end.

In the days of the Ming, after the barbarian invaders lost the Mandate of Heaven, in the twenty-third year of the Hongwu Emperor, there lived an artist named Lao Yuxi. Lao had studied with the nameless masters in Fujian, but now he lived alone in Suzhou, on the shore of Lake Tai, near the canals. There he cooked xiaolong bao for himself, and painted, speaking to few. In his garden he cultivated the fragrant suzi plant which gives Suzhou its name, which is called सलिाम or silam. Once a year he made the long trek across the enormous, ancient city to the tall stone Yunyan Temple on top of Tiger Hill, which was not as tall in those days, but had already begun to lean. The painter Lao was said to be a greater painter than the Hanlin Yuan painters, greater even than Shangguan Boda's painters at the Palace of Virtue and Knowledge, who wore the golden girdle — even though the painter Lao did not have to pay a fine for each error in his paintings.

The Hongwu Emperor aspired to restore the artistic glory the Song had achieved before the barbarian invasion. That summer, when the magnolias were in bloom, hearing of the painter Lao, he called him to his court at the Capital of the South (which is pronounced "Nanjing" in Chinese) and commissioned him to paint a cock crowing at dawn. Lao took a fortune in gold and bronze back to his home in Suzhou, on the shore of Lake Tai, near the canals, where he painted.

A month later, the Hongwu Emperor sent for the painting of the cock, but the painter Lao was eating xiaolong bao, which burst in your mouth when you bite them. He sent back word that the painting was not done yet. So the Emperor waited.

Summer gave way to autumn, and when the tree limbs were bare against the sky, again the Hongwu Emperor sent for the painting of the cock. The painter Lao was weeding among the fragrant suzi in his garden, and he sent back word that the painting was not done yet. So, still, the Emperor waited.

Autumn became winter and then spring, and in the summer, when the lotus began to bloom, the mighty Hongwu Emperor sent for the painting of the cock a third time. When the Emperor's messenger arrived at the house of the painter Lao in Suzhou, on the shore of Lake Tai, near the canals, Lao was not there. The messenger waited impatiently until it was dark, and finally the painter Lao returned from the tall stone Yunyan Temple on top of Tiger Hill. He gave the hungry messenger xiaolong bao, which burst in his mouth when he bit it. But, again, the painter Lao sent back word that the painting was not done yet. The Emperor knew patience from his days as a monk, and he knew the value of mercy from the years he had wandered as a beggar, and so, still, the Hongwu Emperor waited.

That autumn, the leaves turned yellow and orange, then deep red. The Hongwu Emperor had given the land of many nobles to peasants, and those nobles went to speak with the general Lan Yu, the Emperor's oldest friend, the personal tutor of the Crown Prince. When the Hongwu Emperor found ten thousand of the finest Japanese swords secreted in Lan's house in Anhui, he knew Lan had betrayed him, so he put to death his oldest friend, Lan Yu. Then, the Emperor put to death fifteen thousand more men the Embroidered Uniform Guard said were loyal to Lan.

But the painter Lao was at his house in Suzhou, on the shore of Lake Tai, near the canals. He cooked xiaolong bao for himself, and they burst in his mouth when he bit them.

That spring, when the mimosas bloomed, the Hongwu Emperor punished the Embroidered Uniform Guard for abusing their authority during the investigation. Times were very difficult. The Emperor put to death tens of thousands more people, including three of his favorite concubines.

It was two more years before the Hongwu Emperor paid attention to artistic matters, and then he was furious. He did not want to wait any longer. He went to the house of the painter Lao in Suzhou, on the shore of Lake Tai, near the canals. As the sun set over Lake Tai, he burst through Lao's door and demanded the painting he had paid such a fortune for.

The painter Lao put down his plate of xiaolong bao. He picked up his brush and his inkstone, and with a few deft strokes, he created the most beautiful painting of a cock the Hongwu Emperor had ever seen, so lifelike it was impossible to believe human hands had painted it. Not a single brushstroke was wasted, and the cock looked ready to walk off the paper and peck the Hongwu Emperor. Somehow the dawn in the painting seemed to glow. The Emperor felt he could not breathe when the painter Lao handed him the painting.

But this evidence of the incomparable excellence of the painter Lao did not appease the Hongwu Emperor. After he recovered from his astonishment, he was angrier than before. "You have kept me waiting three years while you ate xiaolong bao," he roared, "for a painting you could paint in twenty minutes?" His bodyguard stirred uneasily outside the door in his embroidered uniform.

The painter Lao showed no fear and spoke no word. He slid open his closet door. Ten thousand paintings of cocks avalanched onto the floor. Many of them were terrible, even ridiculous. Some were good. But the best painting of all was the one in the Emperor's hand.

The Hongwu Emperor was ashamed. Silent, he left with the twenty-minute painting on which the painter Lao had spent three years of his life, and he returned to Nanjing. For the rest of his life the painting was his most precious possession.

Topics

- Art (p. 1306) (3 notes)
- Fiction (p. 1368) (2 notes)

- Learning

Bench supply

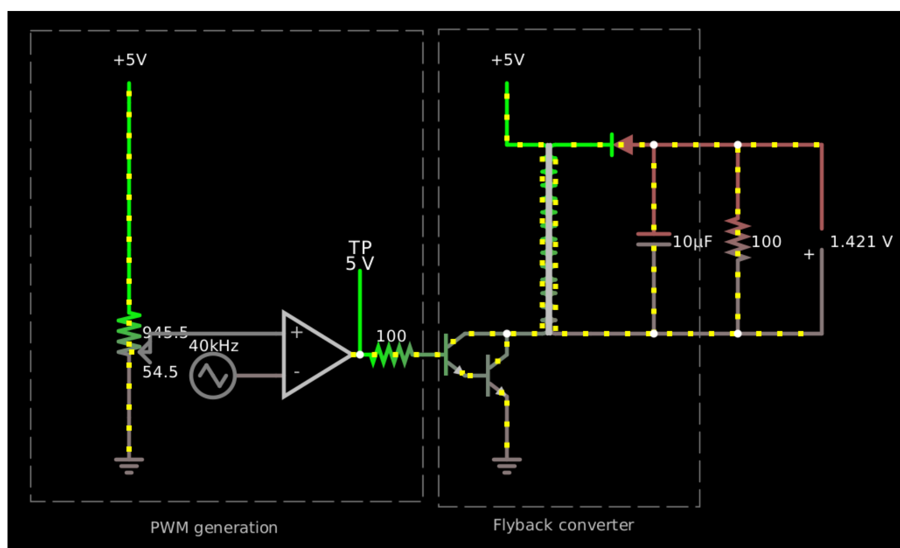
Kragen Javier Sitaker, 02021-03-19 (updated 02021-12-30)
(25 minutes)

One thing I've been procrastinating on for years is building a proper bench power supply. Minimally it needs a galvanically isolated single-ended DC output voltage adjustable from 0 V to 15 V with an adjustable current limit of up to at least a few hundred milliamps, some kind of readout that tells you what it's delivering, and precision of ±10% or better over most of that range. Bipolarity supplies, larger voltages, very precise regulation, linear regulation, foldback current limiting, high efficiency, and thermal protection would be nice but are less essential.

A super ghetto flyback design

The standard design for this kind of thing involves a lot of analog components, but computers are cheaper than transistors now, so it might be better to use an analog circuit that's as simple as possible and pushes as much complexity as possible into software.

So ideally you'd like some kind of transformer-isolated switching buck–boost converter. Like a flyback converter, which I've drawn here being driven from a discrete-component PWM generation circuit instead of a microcontroller:




```
$ 1 1e-8 16.817414165184545 50 5 43 5e-11
R 208 160 160 160 0 3 40000 2.5 2.5 0 0.5
R 96 -48 96 -80 0 0 40 5 0 0 0.5
a 208 144 272 144 9 5 0 1000000 0.027999984128133537 0.025000000000000005 100000
174 96 192 112 64 1 1000 0.005 Resistance
g 96 192 96 224 0 0
w 112 128 208 128 0
368 272 144 272 64 0 0
t 320 144 352 144 0 1 -12.72087652079897 -0.20438741318712572 100 default
r 272 144 320 144 0 100
R 384 -16 384 -80 0 0 40 5 0 0 0.5
T 384 128 448 -16 0 0.000004 1 -3.0191582567340447e-10 -0.12340456006778311 0.999
```

```
d 496 -16 448 -16 2 1N5712
c 496 -16 496 128 0 0.000009999999999999999 -0.40177526518551265 0.001
w 496 128 448 128 0
w 496 -16 560 -16 0
w 496 128 560 128 0
r 560 -16 560 128 0 100
w 560 -16 624 -16 0
w 560 128 624 128 0
p 624 128 624 -16 1 0 0
w 96 64 96 -48 0
t 352 160 384 160 0 1 -12.516489107611845 0.2043871144711009 10 default
w 384 144 384 128 0
w 352 128 384 128 0
g 384 176 384 224 0 0
b 320 256 21 -103 0
x 112 280 210 283 4 12 PWM\sgeneration
b 332 -104 531 260 0
x 373 278 481 281 4 12 Flyback\sconverter
o 6 128 0 4099 10 6.4 0 2 6 3
o 19 128 0 4098 20 0.1 0 1
```

The actual flyback converter itself is four components: an npn darlington grounding one end of the primary of a transformer, a schottky, and a capacitor. Well, and a base resistor for the darlington if the input signal is voltage-mode PWM instead of current-mode PWM, so that's five components. (Horowitz & Hill suggest a zener snubber across the primary to limit the voltage spike from the primary leakage inductance: a rectifier and a TVS zener, in opposite directions, so it's more like seven. But I didn't simulate that. In their example the leakage inductance is about 5% of the total.)

Here's the values I chose in the above simulation:

- a 5-volt supply;
- 40 kHz;
- a 100Ω base resistor;
- β=100 for the first transistor and β=10 for the second;
- a 1:1 transformer with a 4 μH primary inductance and a coupling coefficient of 0.999;
- a 1N5712 schottky with a forward voltage around 290 mV;
- a 10μF output smoothing cap;
- a 100Ω load.

With these values I get about:

- ±1% output ripple with a high output voltage, around 17 V and 170 mA (2.9 W);
- ±2% output ripple near the middle of the range, around 13 V and 130 mA,;
- ±1% output ripple near the bottom of the range (around 1.4 V and 14 mA);
- ±1% output ripple at the bottom of the range (0.5% duty cycle) (around 80 mV and 0.8 mA).

At the top of the range (99.5% duty cycle) the output voltage is actually *lower* with this load.

It can drive thirstier loads like 10Ω at the cost of more ripple, for example 260–330 mV (±6%) but then it tops out at around 2.5–3.2 V (±6%), at which point it's pushing 300 mA (≈1 W). So you'd probably want a bigger output cap. τ = 10Ω·10μF = 100 μs, and the PWM period here is only 25 μs.

This design has a serious problem, though. At high output voltages, and especially when increasing the output voltage rapidly, the power transistor sucks up a lot of power, like tens to hundreds of watts. It seems that it's not getting enough base current. At one point, for example, its Vce is 3.2 volts and its Ic is a rather alarming 33.6 amps, which wouldn't be so bad if it had a nice saturated Vce of 0.2 volts or so, but noo, it's forward active! It's "only" getting 3.4 amps of base current, you see, because the other transistor is *also* forward active, because it's only getting 33 mA of base current and so it's "only" passing 3.4 amps of Ic. This is in part because both transistors have a relatively high Vbe at this point of almost 900 mV, reducing the voltage on the base resistor to only 3.3 V.

Reducing the base resistor value to 47 Ω helps a bit with the problem, but that's demanding a lot of current on the PWM input. Even the 38 mA or so this circuit can demand is kind of a lot. Really probably what we need is an additional amplifier stage. Also it's probably hard to find a transistor that can actually handle 3.4 amps of Ic but has β=100. A better alternative might be to use a pnp signal preamp (β=100) to drive an npn power darlington (β=10 × β=10); that way we avoid stacking more Vbes and can also use a beefy power transistor there. But using a power MOSFET, and maybe driving the whole circuit from a 12-volt supply instead of 5 volts so it can use lower currents, is maybe a better idea.

Doing the pnp-preamp thing driven from the PWM voltage input signal through a 470-Ω base resistor does indeed help keep the switching transistor's Vce from going above a volt or so, so it would dramatically reduce the power-dissipation problems. (It also inverts the sense of the PWM signal.)

I think the transformer's inductance imposes an inherent limit on the power this circuit can deliver at a given frequency, which in this simulation is about 3 watts, as we can see above. The energy per cycle is $\frac{1}{2}LI^2$. But what determines the maximum current? It's not clear to me — the current's derivative clearly is inversely proportional to the inductance, but it need not fall to 0 each cycle. Mindlessly banging on the simulator didn't yield any ready answers; altering the transformer inductance, PWM frequency, and transformer coupling losses, even over orders of magnitude, didn't get my output voltage over 20 volts, although there are higher voltages floating around the circuit, sometimes over 100 volts. I said I only needed 15 volts, but I'd like to understand what the invisible limit is here.

OH, interesting — switching the diode model to a 1N4004 did the trick, and now I can get 120V output. Maybe the Schottky's reverse leakage was the limit! Changing just the diode model gives us potentially much higher output voltages, like over 100VDC, because now our reverse leakage is like 18 nA instead of ...0.7 nA with the 1N5712? No, that can't be it.

Oh, yes, it is — the simulation says that once it has about 20.8 volts

of reverse bias it starts passing 60 mA in reverse (or probably any quantity at all, whatever's needed to keep it from going below -21.)

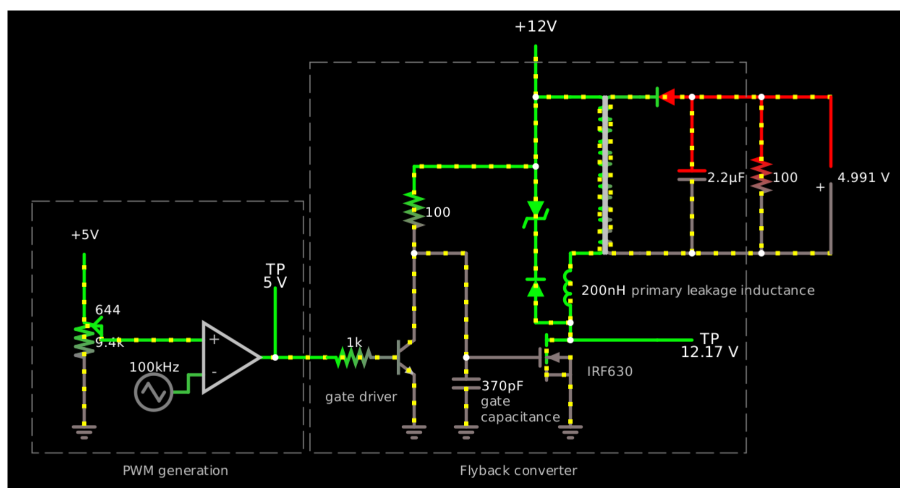
So suppose you do build such a device; how do you regulate the output? You need some way to measure the output voltage and current so you can react to them. One way to deal with this is to try to move those analog quantities across the galvanic barrier, for example with analog optocouplers or pulse transformers, putting the microcontroller on the non-floating part of the circuit. A different approach is to float the microcontroller, although you probably don't want to try to run the microcontroller off 80 mV; you need a separate isolated power supply for the microcontroller, probably using a separate transformer. Then you can hook its analog inputs up directly to the output circuit with relatively little fear, and use a single pulse transformer or ordinary on/off optocoupler to transmit the information back over to the non-isolated side. That sounds simple!

Probably you'll want a sense resistor, say 1Ω, one side of which is connected to the microcontroller's ground. At 300mA it would produce 300mV, which is a very reasonable amount to measure, about 25% of full scale (256 counts) on an AVR's 1.1V internal reference. You'll need to divide down the actual output voltage, and maybe not suck more than 100μA into the bargain; if it's 20V that means a 200kΩ divider, which is pretty reasonable, and you can divide it down, say, with 220kΩ and 6.8kΩ. Then 20V on the power supply output works out to 600mV on the microcontroller's input. The smallest change you can reliably measure with a raw AVR is probably about 5 mV, which works out to about 170 mV at the output — not ideal but not terrible. Multiple sensing ranges attached to multiple input pins may be a good idea here.

The output doesn't need to be regulated for a constant DC voltage. With a center-tapped primary and an additional transistor switch, you can generate an AC voltage on the output instead, with an arbitrary waveform limited only by the PWM frequency. (You could just use a DC-blocking capacitor on the output, but then you could *only* use it as an AC supply, instead of changing mode under software control.)

A slightly better flyback design

Here's a revised version with 12V, 100kHz, leakage inductance simulation and protection, a higher-breakdown Schottky, and a MOSFET switch driven through an npn level-shifter:



```
$ 1 3.0000000000000004e-9 2.008553692318767 46 5 43 5e-11
R -96 112 -96 80 0 0 40 5 0 0 0.5
a 0 192 80 192 9 5 0 1000000 2.986054403507636 4.678 100000
174 -96 112 -96 224 1 10000 0.06440000000000001 Voltage knob
w -80 176 0 176 0
368 80 192 80 112 0 0
R 320 -48 320 -112 0 0 40 12 0 0 0.5
T 352 96 416 -48 0 0.000004 1 -2.3850875119357795e-7 6.800005003526621e-10 0.999
34 power\sschottky 0 6.8e-10 12 1.003 150 0
d 464 -48 416 -48 2 power\sschottky
c 464 -48 464 96 0 0.0000022 -15.440326076997223 0.001
w 464 96 416 96 0
w 464 -48 528 -48 0
w 464 96 528 96 0
r 528 -48 528 96 0 100
w 528 -48 592 -48 0
w 528 96 592 96 0
p 592 96 592 -48 1 0 0
w 352 176 352 160 0
w 320 160 352 160 0
g 352 208 352 256 0 0
x -61 300 37 303 4 12 PWM\sgeneration
x 250 300 358 303 4 12 Flyback\sconverter
l 352 160 352 96 0 2.0000000000000002e-7 -2.38508751092964e-7 0
d 320 160 320 96 2 power\sschottky
34 fwdrop\q0.806 1 1.7143528192810002e-7 0 2.0000000000000084 50 1
z 320 16 320 96 2 fwdrop\q0.806
w 320 16 320 -48 0
w 80 192 128 192 0
x 408 134 576 137 4 12 primary\sleakage\sinductance
368 432 176 480 176 0 0
w 352 176 432 176 0
w 352 -48 320 -48 0
t 176 192 208 192 0 1 0.6242259399470127 0.7204676364540711 100 default
r 128 192 176 192 0 1000
w 208 96 256 96 0
w 256 96 256 192 0
f 256 192 352 192 32 3 0.4
r 208 16 208 96 0 100
c 256 192 256 240 0 3.7e-10 0.09624169650705836 0.001
g 256 240 256 256 0 0
x 269 236 296 239 4 12 gate
x 269 251 342 254 4 12 capacitance
x 367 209 410 212 4 12 IRF630
g 208 208 208 256 0 0
x 126 232 192 235 4 12 gate\sdriver
w 208 16 320 16 0
w 208 96 208 176 0
g -96 240 -96 256 0 0
w 0 208 0 224 0
R 0 224 -32 224 0 3 100000 2.5 2.5 0 0.5
b -144 48 106 280 0
b 112 -80 514 280 0
o 4 128 0 4099 20 12.8 0 2 4 3
o 15 128 0 4354 40 0.1 0 1
```

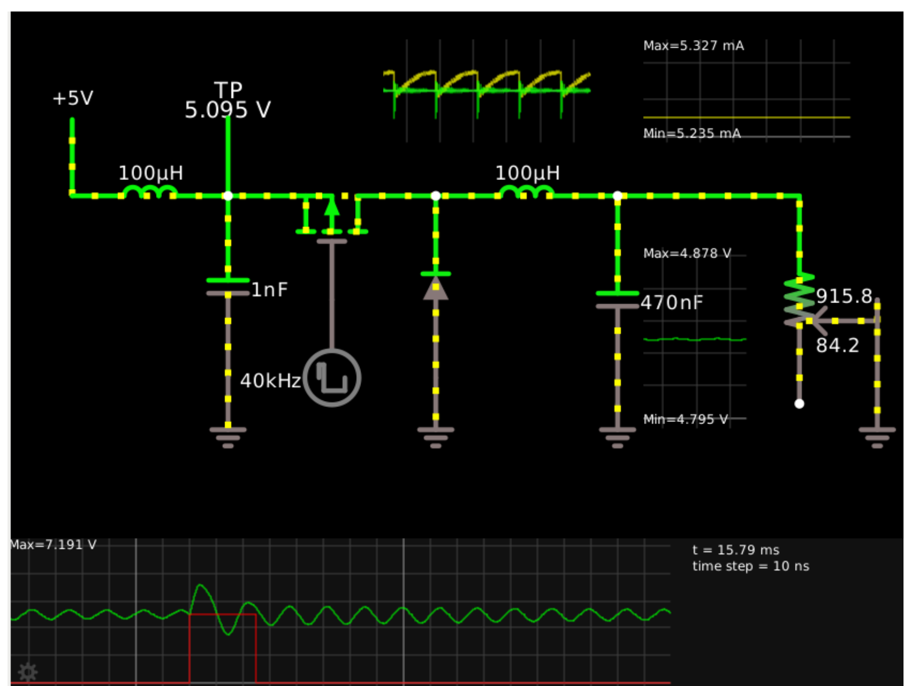
```
o 27 32 0 4099 320 204.8 1 2 27 3
o 34 32 7 xa1013 80 0.4 1 2 640 20 0 23 7 640 20 0
```

Pulling up the IRF630's gate with a resistor makes it turn on somewhat slowly, but this is not really important, because its drain current ramps up slowly from zero anyway. You could probably use a slower resistor and save some power. The strong npn pulldown slams it off quickly, and that *may* be important, because when it turns off, it may be carrying 15 amps! However, the IRF630 doesn't have excellent on-resistance — even when fully turned on, at 15 amps, the ST datasheet says its Vds is about 7 volts, about half an ohm. Beefier parts like the IRF540 would make that part of the circuit cooler, more efficient, and perhaps more reliable; so too would a GaN part like the EPC2036.

Inductor current feedback

I was thinking about the current-sensing problem today, stimulated by Chris Glaser's note on high-side current sensing for LED lighting using the voltage drop across the switching power MOSFET. I was driven to revisit a thought I'd had previously about high-side current sensing: if you're using a microcontroller, maybe you could sense the current intermittently by using an inductor to hold the current constant while you divert it?

Here's an example of this current-feedback scheme, where the switching P-MOSFET of a standard buck converter is fed from an LC low-pass filter, so its source experiences a significant positive voltage excursion every time it switches off, whose initial slope and peak voltage value should tell us rather precisely how much current the buck converter is drawing.




```
$ 1 1e-8 0.9487735836358526 50 5 43 5e-11
R -112 -16 -112 -80 0 0 40 5 0 0 0.5
l -112 -16 -16 -16 0 0.00009999999999999999 0.009476973841339323 0
l 112 -16 224 -16 0 0.00009999999999999999 0.007763234883748598 0
```

```
f 48 48 48 -16 41 1.5 0.02
w -16 -16 32 -16 0
c -16 -16 -16 96 0 1e-9 4.82585135926864 0.001
g -16 96 -16 128 0 0
R 48 48 48 96 0 2 40000 2.5 2.5 0 0.1
174 336 -16 384 112 1 1000 0.9158000000000001 Resistance
c 224 -16 224 112 0 4.7000000000000005e-7 4.827958335480461 0.001
g 224 112 224 128 0 0
g 384 48 384 128 0 0
w 224 -16 336 -16 0
368 -16 -16 -16 -80 0 0
w 112 -16 64 -16 0
d 112 96 112 -16 2 1N5711
g 112 96 112 128 0 0
403 80 -112 208 -48 0 2_64_0_4103_10_0.1_0_2_2_3
403 240 -112 368 -48 0 12_64_0_4353_5_0.1_0_2_12_3
403 240 16 304 128 0 9_64_0_4354_10_0.05_0_2_9_3
o 13 4 0 4098 10 6.4 0 3 7 0 7 3
```

In simulation this sort of works, but not as well as I had hoped. When the load is drawing 5.3 mA, the input voltage jounces up to 7.191 V, because at the point the transistor cuts off, the input 100μH inductor is carrying 6.34 mA. But when the load is drawing 50.0 mA, the input voltage jounces up to 9.109 V, because the input inductor was carrying 54.1 mA. Physically this seems wrong; in the 6.34 mA case the inductor holds 2.0 nJ of energy, but the input 1 nF capacitor would contain 25.9 nJ at 7.191 V, while at the nominal 5 V it would contain 12.5 nJ, a gain of a lot more than 2.0 nJ; this could be explained if the voltage on the capacitor was already 6.91 V at the point where the voltage starts climbing. In the 54.1 mA case the 100 μH inductor contains 146 nJ, and at 9.109 V the 1 nF cap contains 41.5 nJ, which is a mystery in the other direction.

This is partly explained by my inept circuit design: I was trying to cut off the P-MOSFET (whose Vth is simulated as -1.5V) by putting its gate at 5 V, but of course as soon as the inductor drives its source up past 6.5 V it starts conducting again. Nevertheless I think it proves out the underlying principle. (This mechanism might actually be a useful way to limit the peak voltage to something safe, if you're measuring the current using the rate of voltage rise rather than the peak voltage.)

The appeal of this complicated mechanism over a sense resistor (or using the MOSFET on-resistance) is that in theory you should be able to get more precise measurements at lower energy loss and regulation instability for a given precision of voltage measurement.

Suppose your switching MOSFET is a 2N7000: can handle 200 mA (and dissipate up to 400 mW) and block 60 V with 1.9 Ω of on-resistance, switched with 2 nC of gate charge with about a 3 V threshold voltage. (According to Can you get JLCPCB to fabricate a CPU for you affordably from "basic" parts? (p. 347) JLCPCB will solder a 2N7002, a lower-power version, onto your surface-mount project, for 1.23¢ for the part plus 0.45¢ for the three terminals. According to file `jellybeans` in Dercuano, Digi-Key sells the 2N7002 for 2.958¢ in quantity 1000, and local merchants here in Argentina sell

the 2N7000 for 12¢.) And suppose you're building a tiny adjustable power supply with one or more of these, supplying up to 200 mA and up to 12 volts, working from a 12-volt supply, and you're using a single-ended ADC with ±0.2% error over the range 0 to 1.1 volts. (For now I'll assume your voltage reference is better than that.)

The 2N7000 has low enough on-resistance that, if it's being operated in the fully-on or fully-off states, it will never come close to its 400 mW limit. The surface-mount 2N7002 has higher on-resistance (7 Ω) and max current (115 mA) but I think is actually *more* able to dissipate heat than its through-hole progenitor, so I think the max current is the only limit here.

Suppose you're doing high-side sensing with a 1 Ω sense resistor, just using the ADC directly rather than some kind of differential amplifier or flying-capacitor setup. You need to use some kind of resistive dividers to divide the 12 V signal down into the 1.1 V range, say 100kΩ and 10kΩ (GND-10kΩ-t-100kΩ-x-1Ω-y-100kΩ-u-10kΩ-GND, and the ADC measures the voltages at points t and u in order to measure the current between points x and y). You'll have some gain error between the two ends of the sense resistor, but that's easy enough to calibrate out. Then your 200 mA maximum current generates 200 mV, which gets divided down to a difference of 18.2 mV (1.091 V on the high end, 1.073 V on the low end). ±0.2% error on your 1.1 V signal would be ±2.2 mV, which is ±24 mA, or ±12% error on your current reading even at max scale. If you actually care about whether your circuit is using 10 mA or 30 mA, it's hopeless.

If you use a 4.7Ω sense resistor instead, those 200 mA becomes a 85 mV signal (1.005 V vs. 1.091 V), and your ±0.2% error now translates to ±5.1 mA, which is ±2.5% on your full-range current measurement, which is not good but not completely useless. But such a large sense resistor means that at 200 mA output the sense resistor is eating almost a whole volt; not only is your 12 V circuit running at 11.06 V, if you're feeding the load 200 mA at 1 V, the high side of the resistor has to be at 1.94 V (and burning 200 mW, which is not good for precision). That's potentially pretty unhealthy for the load if it suddenly drops from 200 mA to 10 mA — output capacitance upstream of the sense resistor could raise the load voltage up from 1 V to 1.89 V instantly. Capacitance across the load, downstream of the sense resistor, could prevent this, at the cost of creating similar uncontrolled and unmeasured *current* excursions when the load impedance suddenly drops instead of suddenly rising.

If you use a low-side sense resistor instead, you avoid the need for the 11× divider, and your current-measuring precision correspondingly improves by a factor of 11. This also doubles your data rate and eliminates the need to calibrate out the gain difference between the dividers on the high side and low side of the sense resistor. But now the load isn't grounded; it's floating above ground, potentially by as much as 0.94 V in the above 4.7Ω scenario. And the sense resistor is still burning 184 mW at maximum output current. For an isolated supply maybe this doesn't matter. But this would give us ±0.2% error on the maximum current, i.e. ±400 μA.

Consider instead the situation where we use an LC circuit

upstream of the switching transistor. We should be able to arrange for the full-scale C voltage at 12V output to be 48 V or so, and an 0.2% error in measuring that voltage should be 96 mV, which would be an 0.27% error in the 36V ΔV, which corresponds to an 0.54% error on the energy, which corresponds to an 0.27% error in the 200 mA current, ±540 μA. We can take multiple data points on the voltage curve as it rises, which allows us to eliminate any offset error in the ADC, and we don't have to deal with calibrating out gain differences between different dividers, because we have only one divider to worry about. We could quite plausibly take 20 data points for our measurement with a 1Msps ADC like the one on the STM32 series in 20 μs...

Ugh, but that means that if we want to only be doing this measurement 10% of the time, we can only run the buck converter at 5kHz, which is horrible. I was hoping to come up with a happy story here like "so, you see, we can divert an arbitrarily small amount of our power to measurement, and still get a very precise measurement!" but that's not going to be true with an ADC. So it looks like we'd have to do some analog RF electronics to measure the peak current level. Maybe a flyback winding on the input inductor or something.

Topics

• Electronics (p. 1145) (39 notes)
• Ghettobotics (p. 1169) (12 notes)
• Power supplies (p. 1176) (10 notes)
• Falstad's circuit simulator (p. 1198) (7 notes)

Recursive residue number systems?

Kragen Javier Sitaker, 02021-03-20 (updated 02021-03-22)
(8 minutes)

As I wrote in https://news.ycombinator.com/item?id=26525837 residue number systems with many small bases are not well suited to day-to-day human use; they're more confusing than systems with a smaller number of bases. So, for example, the Maya tzolk'in residue number system calendar uses bases 13 and 20, necessitating at least 20 digits (though in fact the Maya used different digits for the two cycles, one being an ordinary numeral and the other being the name of one of 20 natural phenomena). Small bases also tend to vary a lot from digit to digit; in that post I suggested using bases 5, 6, 8, 9, and 11, but that means that you need at least 11 digits, but can only use 5 of them in the fastest-cycling digit, wasting more than a whole bit of encoding space.

The conventional way to solve this problem is to use another number system for the individual digits. For example, the 13 day numbers of the Maya were
which are written in base 5 (and 1-origin — Dijkstra would not approve), and the Babylonians used up to 14 cuneiform strokes to form their sexagesimal digits: up to five "ten" strokes, and up to nine "one" strokes. And Wilhelm Fliess wrote his cocaine-fueled biorhythm numbers, counting days modulo 23 and 28, pairs of Western Arabic digits, as do modern biorhythm scammers, adding a 33-day cycle. So day 10000 of a person's life might be represented as 18(P), 4(E), 1(I), because 10000 % 23 = 18, 10000 % 28 = 4, and 10000 % 33 = 1, although it's more common for biorhythm scammers to just plot some sine waves.

Applying the RNS principle recursively

This is a fascinating alternative. For example, suppose we have 16_{10} digits, like the corners of a tesseract, or conventional sexadecimal symbols. With two of them we can represent a number mod 240_{10} with its moduli relative to 15_{10} and to 16_{10}:

```
00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee
f0 01 12 23 34 45 56 67 78 89 9a ab bc cd de
e0 f1 02 13 24 35 46 57 68 79 8a 9b ac bd ce
d0 e1 f2 03 14 25 36 47 58 69 7a 8b 9c ad be
c0 d1 e2 f3 04 15 26 37 48 59 6a 7b 8c 9d ae
b0 c1 d2 e3 f4 05 16 27 38 49 5a 6b 7c 8d 9e
a0 b1 c2 d3 e4 f5 06 17 28 39 4a 5b 6c 7d 8e
90 a1 b2 c3 d4 e5 f6 07 18 29 3a 4b 5c 6d 7e
80 91 a2 b3 c4 d5 e6 f7 08 19 2a 3b 4c 5d 6e
70 81 92 a3 b4 c5 d6 e7 f8 09 1a 2b 3c 4d 5e
60 71 82 93 a4 b5 c6 d7 e8 f9 0a 1b 2c 3d 4e
50 61 72 83 94 a5 b6 c7 d8 e9 fa 0b 1c 2d 3e
40 51 62 73 84 95 a6 b7 c8 d9 ea fb 0c 1d 2e
30 41 52 63 74 85 96 a7 b8 c9 da eb fc 0d 1e
20 31 42 53 64 75 86 97 a8 b9 ca db ec fd 0e
10 21 32 43 54 65 76 87 98 a9 ba cb dc ed fe
```

Now we apply the recursive step, bottom up. An additional such pair can represent the number mod 239: 00 00, 11 11, 22 22, … dc dc, ed ed, fe 00, 00 11, 11 22, … ed dc, fe ed, 00 00. This lets us represent $57'360_{10}$ numbers in four digits. A second recursive step represents a number in this way directly and also modulo $57'359_{10}$, so that following ed dc ed dc, we have not fe ed fe ed but fe ed 00 00. This can represent numbers up to $3'290'112'240_{10}$ in eight digits: 31.6 bits of information, only 1.2% less than a binary place-value notation.

Extension to multiple precision

It also has a straightforward rule for abbreviating small numbers: to extend a small positive number to a longer version, the longer version of the number just has more copies of the same number, in a way analogous to zero-extending or sign-extending in conventional place-value systems. So 3 can be validly represented, for example, as 33, 33 33, or 33 33 33 33, and a4 is the same as a4 a4 or a4 a4 a4 a4. In the notation used in the above counts, I am extending numbers to the right rather than, as is conventional, on the left, but the extension algorithm is the same either way.

Sign testing is straightforward and efficient

The rule to determine the *sign* of a number, which is an enormous problem with residue number systems in general, is easy but not totally trivial. You subtract the two halves of the number from one another and use the sign of the result. So, for example, ed dc is negative, because dc - ed = fe, which is negative because e - f = f, which is negative because I said so. You can choose where to put the window where positive numbers wrap around to negative, but as long as you have at least *some* negative numbers this approach will give you the right subtraction sign if you do the arithmetic with sufficient precision.

I’m not sure how to do ordinary arithmetic though!

I thought the usual digit-by-digit methods of residue number systems would apply in a trivial way here because of the simple recursive construction, but they don’t. 05 a8 5a fd multiplied by a4 3e a4 3e, for example, is 05 15 c2 ca: $320'000_{10} \times 6154_{10} = 1'969'280'000_{10}$. For the first two digits we can just multiply corresponding digits: $0 \cdot a = 0$, $5 \cdot 4 = 5$ (mod 15_{10}). But to compute that a8·3e ($218_{10} \cdot 179_{10}$) should give us ‘15’ (mod 239; 65_{10}), we cannot just multiply a·3 mod 16_{10} and 8·e mod 15_{10}, which would give us e7, the wildly different number 142_{10}. That’s because $218_{10} \cdot 179_{10} = 39'022_{10}$, which is $163_{10} \cdot 239_{10} + 65_{10}$ or $162_{10} \cdot 240_{10} + 142_{10}$.

To put it another way, a8 a8 · 3e 3e = e7 15. But how do we calculate that “15”? It happens that 15 - e7 = $65_{10} - 142_{10}$ (+ 239) = 162_{10}, because each lap around the racetrack, the mod-239_{10} number gained one on the mod-240_{10}. So if we could somehow calculate that it took 162_{10} = 2c laps to get to e7, we could add those differences back in and get the right answer: 2c + e7 = 15, where the first digits

add mod 16_{10} and the second digits add mod 15_{10}.

But how on Earth do you calculate that within the RNS without falling into an infinite regress of four-digit multiplications?

For *addition*, I think the problem may be easier. Let's divide positive from negative numbers as equally as possible, rather than taking advantage of the window-choosing freedom I described above. Then, if we add two numbers of the same sign and get a number of the opposite sign, I think we can conclude that we've lapped around the racetrack, going one way or the other — but only once.

Topics

- Algorithms (p. 1163) (14 notes)
- Math (p. 1173) (11 notes)
- Facepalm (p. 1199) (7 notes)

Brute force speech

Kragen Javier Sitaker, 02021-03-21 (updated 02021-03-22)
(7 minutes)

I just read the 100 most popular words from the British National Corpus out loud, one at a time, with pauses in between:

the of and to a in it is was that i for on you he be with by at have are this not but had his they from as she which or we an there her were do been all their has would will what if one can so no who said more about them some could him into its then up two time my out like did only me your now other may just these new also people any know first see well very should than how get most over back way our much think between years go er

This took 114 seconds, or 1.14 seconds per word, so probably the first 1000 words would take me about 20 minutes, and the first 10000 words about three or four hours.

These 100 words comprise 50.56% of the words in the BNC ("the" being about 6.87%, and "er" being about 0.1%), or at any rate the part of it that is represented in my frequency table, so if you had recordings of them (as LPC-10 or whatever) you could only synthesize about half of the words in typical text. Here's an overview of how that number changes:

Word	Rank	Cumulative %	Self %	Time to record this many	To miss one word in
much	95	50.05%	0.10%	2 minutes	2
er	100	50.56%	0.10%	2 minutes	2
married	1261	75.005%	0.0083%	24 minutes	4
colonel	4407	87.5004%	0.0020%	1½ hours	8
cdna?	10445	93.7500%	0.0006%	3½ hours	16
capitals	10446	93.7506%	0.0006%	3½ hours	16
stabilise	20104	96.87494%	0.00019%	6½ hours	32
pasha	33275	98.43748%	0.00007%	11 hours	64
lessens	43350	99%	0.00004%	14 hours	100
pizzicato	60386	99.5%	0.00002%	19 hours	200
superfields	76770	99.75%	0.000011%	24 hours	400

After the first 8000 words or so, the ordering starts to be somewhat dubious, as reverse alphabetization sets in due to small corpus size.

Brute-force recording the whole English lexicon seems like a surprisingly approachable, if boring, project, from this point of view. In a few days of work you could compile enough recordings that your computer could read most text understandably, if not naturally.

I analyzed (an earlier version of) this note using these frequencies. It contained two known words less common than #65536: "BNC" and "superfields"; four words less common than #32768: "synthesize", "pasha", "lessens", and "pizzicato"; three words less common than #16384: "pauses", "stabilise", and "approachable", plus some HTML tags; and 8 words less common than #8192: "corpus", "comprise", "cumulative", "cdna", "capitals", "brute", "lexicon", and "compile".

Here's the full analysis:

100 114 1 14 1000 20 10000 100 50 56 6 87 0 1 LPC 10 95 50 05 0 10 2 2 100 50 56 0 10 2 2 1261 75 005 0 0083 24 4 4407 87 5004 0 0020 1 8 10445 93 7500 0 0006 3 16 10446 93 7506 0 0006 3 16 20104 96 87494 0 00019 6 32 33275 98 43748 0 00007 11 64 43350 99 0 00004 14 100 60386 99 5 0 00002 19 200 76770 99 75 0 000011 24 400

1

the the the the the the the the the the the

2

of and and of and of of of of of of

4

a in to a in it in it in in to To in a In a

8

I is was that i for on you he that is you you that you that

16

from at with be with by at have are this not but had his they from as she which This at had as this from this

32

one or we an there her were do been all their has would will what if one can so no who said more about them some could him into its then up two or so would about about or about about or so if them or could about an one if could

64

just most out time between time my out like did only me your now other may just these new also people any know first see well very should than how get most over back way our much think between years go er first take me first three These being er being any my only Here how Time many much er like work

128

British National took four part number point days

256

read words word probably words minutes words hours words words rate table half words table Word word minutes minutes minutes hours hours hours hours hours hours hours hours table whole English seems view few enough

512

popular per whatever s changes record miss force project

1024

typical text married

2048

seconds seconds represented frequency th th th th Self th th recording surprisingly

4096

loud recordings overview Rank br br colonel boring recordings

8192

Corpus comprise Cumulative cdna capitals Brute lexicon compile

16384

pauses tr tr td td td td td td tr td td td td td td tr td td td td td td tr td td td td td td tr td td td td td td tr td td td td td td tr td stabilise td td td td td tr td td td td td td tr td td td td td td tr td td td td td td tr td td td td td td approachable

32768

synthesize pasha lessens pizzicato

65536

BNC superfields

So, with 16384 words, which could be recorded in a day or two,

plus numbers and initialisms, only “pauses” and “approachable” would have been missed in a text-to-speech of a note like this one, if we leave out the words deliberately chosen to be uncommon. Even with only a 512-word vocabulary, the note would be comprehensible.

Other notes in Dernocua are not so fortunate. The 7500-word note on energy-autonomous computing (p. 143), for example, includes 1790 words not in the BNC at all; about half of these are real English words such as Github, Kobo, joule, touchscreens, ebook, and 80Mbps, plus many Spanish words and initialisms. The most common 512 words would have covered a bit over 3000 of its 7500 words.

Topics

- Programming (p. 1141) (49 notes)
- Experiment report (p. 1162) (14 notes)
- Audio (p. 1304) (3 notes)
- Speech synthesis (p. 1322) (2 notes)

When is it better to compute by moving atoms rather than electrons?

Kragen Javier Sitaker, 02021-03-21 (updated 02021-03-22) (5 minutes)

Electrons move fast because they are very light and very strongly charged.

A classroom Van de Graaff generator might charge its sphere to 100 kilovolts. The capacitance to infinite space of a sphere of radius *r* is $4\pi\varepsilon_0 r$, about 11 pF for a 10-cm-radius sphere, so this voltage would be about 1.1 μC of charge, about 6.9 trillion electrons. Because an electron weighs about 5.5×10^{-4} atomic mass units, which is 9.1×10^{-31} kg, this quantity of electrons weighs 6.3×10^{-18} kg, 6.3 femtograms. If such a mass were to fall a meter off the Van de Graaff generator onto the table under the force of gravity, it would gain 62 attojoules by falling, dissipating it in the impact (or, more likely, from air resistance). But if these electrons instead "fall" through this 100-kilovolt potential, they gain 111 **milli**joules, 111 quintillion attojoules, about 2 quintillion (2×10^{18}) times as much. So, in the absence of air resistance, they would tend to impact going about a billion times as fast.

The acceleration due to gravity is a pretty normal acceleration in our world, although there are stronger accelerations like hitting things with hammers (100 gees or more) and weaker ones like things rolling down slopes. And 100 kilovolts is a pretty reasonable kind of voltage for electostatic machines, though a bit on the high side for electromagnetic machines and especially for semiconductor devices. So, in general, electrons tend to move around a few thousand times to a billion times faster than macroscopic objects.

This is why electronics are so useful for computing.

The advantage becomes smaller when we're only limited by energy. With a joule, you can accelerate 10 trillion carbon atoms (200 picograms) up to about 0.01 of *c*, 3200 km/s. If you apply that joule over a micron, then you will move them over that micron in 0.6 picoseconds. But if you're accelerating just their electrons, well, those are only 0.11 picograms (110 femtograms), so classically you'd expect to be able to accelerate them to 0.45 of *c*, 135'000 km/s, so classically you could cross that micron in 0.0074 picoseconds, 7.4 femtoseconds. (Relativistic effects increase this by a few femtoseconds.)

Of course you can't normally apply a joule to 200 picograms of anything, much less 110 femtograms — not without the thing ceasing to be a *thing*. 4.184 joules per gram, a calorie per gram, heats up water by a kelvin. So a joule per 200 picograms ends up being 1.2 gigakelvins, about six orders of magnitude hotter than temperatures at which solid matter exists, even solid matter with a somewhat higher specific heat than water. Duly derating the above numbers by six orders of magnitude of energy and thus three of velocity, it seems that

you can move groups of atoms micron-scale distances at nanosecond timescales, or you can move groups of electrons micron-scale distances at picosecond timescales. If you must remain near room temperature, it takes several nanoseconds or several picoseconds.

However, Drexler has suggested that if you're computing with solids, you may not need to move them as far, because Heisenberg's uncertainty principle $\sigma_x \sigma_p \geq ½\hbar$ means that their location can be defined to higher resolution. Here p is the momentum, x is the position, and σ is the standard deviation; so increasing the mass 2000× with a given uncertainty of velocity would increase the uncertainty of momentum by the same 2000×, so decreasing the (possible) uncertainty of position by the same 2000×. So perhaps you have to move some electrons by 1 nm to resolve the result with a given certainty, which seems to be what chip manufacturers do these days, but if you were moving some atoms to encode the same result with the same certainty, you could move them 2000 times less distance, 500 fm.

This seems rather challenging since, for example, the lattice spacing of silicon atoms is around 200 picometers, so you'd be deforming the lattice by about 0.3% of a single atom spacing. LIGO successfully measures such small displacements every day, but it still seems daunting.

Still, if that approach works out, then instead of comparing moving some atoms by a micron, to moving some electrons by a micron 45 times as fast, we'd be comparing moving some atoms by 0.5 picometers, to moving some electrons 2000 times as far, 1000 picometers, 45 times as fast. This suggests that in fact computing by moving around atoms should be about 45 (= 2000 ÷ 45) times as fast at a given level of uncertainty, at least if you can bring similar energies to bear rather than similar electric field strengths.

Topics

- Physics (p. 1157) (18 notes)
- Mechanical (p. 1159) (17 notes)
- Physical computation (p. 1208) (6 notes)

Veskeno is a “fantasy platform” like TIC-80

Kragen Javier Sitaker, 02021-03-21 (updated 02021-03-22) (3 minutes)

Writing about Veskeno, it occurred to me that the nearest equivalent might not be Chifir, the Universal Machine of the Cult of the Bound Variable, Lorie’s UVM, or even Brainfuck, but rather TIC-80 and CHIP-8, or in general video game consoles:

> TIC-80 is a **FREE** and **OPEN SOURCE** fantasy computer for making, playing and sharing tiny games.
>
> CHIP-8 programs are run on a CHIP-8 virtual machine. It was made to allow video games to be more easily programmed for [01970s 8-bit RCA 1802] computers. … In 1990, a CHIP-8 interpreter called CHIP-48 was made for HP-48 graphing calculators so games could be programmed more easily.

And Cowgod say:

> Chip-8 is a simple, interpreted, programming language which was first used on some do-it-yourself computer systems in the late 1970s and early 1980s.

PICO-8, 2:

> PICO-8 is a fantasy console for making, sharing and playing tiny games and other computer programs. … A fantasy console is like a regular console, but without the inconvenience of actual hardware. PICO-8 has everything else that makes a console a console: machine specifications and display format, development tools, design culture, distribution platform, community and playership. It is similar to a retro game emulator, but for a machine that never existed.

The most crucial thing here is the “cartridge format”, an image file format that includes everything a game needs to run, and that the peripherals are fully specified, which is the case for Chifir but not for the UM or Brainfuck. This is true of consoles like the Nintendo, too: you can write a game and, if it works on one Nintendo, you can be reasonably sure it will work on other Nintendos too. You don’t have to worry that maybe some Nintendo can’t handle as many sprites, or runs some instructions faster and screws up the game’s timing, or has less RAM, or has a TSR installed that’s stealing cycles, or has a garbage collection that’s too conservative and causes your game to run out of memory and crash halfway through, or doesn’t interpret the cartridge format in the same way.

The challenge for Veskeno is achieving this, for a “console emulator” written by someone born after this body dies, who doesn’t have access to a working Veskeno implementation, while enabling the format to support applications that are sufficiently powerful to do things like run Linux and Windows at a speed sufficient for artifact preservation, if not everyday use.

To do this, it needs to specify the behavior of the peripherals used for the user interface at a level of detail sufficient to permit real usage; CHIP-8, for example, specifies the keyboard layout used by the COSMAC VIP, designed by CHIP-8’s author, and Norbert Landsteiner explains that the most difficult part of getting the PDP-1 emulation for Spacewar! usable was emulating the afterglow of the P7 phosphor used on the CRT, an aspect Steve Russell specifically called

out for praise in his comments.

Topics

- History (p. 1153) (24 notes)
- Safe programming languages (p. 1172) (11 notes)
- Virtual machines (p. 1182) (9 notes)
- Instruction sets (p. 1214) (6 notes)
- File formats (p. 1233) (5 notes)
- Reproducibility (p. 1277) (3 notes)
- The Veskeno virtual machine (p. 1313) (2 notes)
- Emulation
- CHIP-8
- Chifir
- Brainfuck

Some notes on reading Chris Seaton's TruffleRuby dissertation

Kragen Javier Sitaker, 02021-03-21 (updated 02021-03-22) (16 minutes)

I recently learned about Chris Seaton's dissertation on TruffleRuby, an efficient implementation of Ruby he wrote — 90% of the performance of C in some cases. It's built on top of Graal, which is now free software, and I thought taking some notes would be worthwhile.

Really great stuff

The dissertation itself is not a great pleasure to read, but it has a lot of really first-class information in it.

He mentions "value profiling" where TruffleRuby observes that a particular edge in the dataflow graph is usually a constant value and so partially evaluates the rest of the graph with respect to it. And he has a generalized "inline cache" mechanism ("dispatch chains", introduced p. 101, ch. 5) that caches the results of expensive reflection mechanisms, such as calling a method determined by a selector argument, and checks to see if they are still valid. Both of these seem highly relevant to Bicicleta.

He mentions SubstrateVM, a feature of Graal which I hadn't heard of, but which aims at ahead-of-time compilation for, mostly, JVM applications.

On p. 90 he mentions the `ExactMath` class from Graal, which throws an exception on arithmetic overflow of integers, allowing the TruffleRuby implementation to transparently fall back to bignums upon overflow. This seems like an interesting technique; in a COMFY-style

If we suppose that the diagram of "a conventional PIC" on p. 104 is literally correct, it may explain why Ur-Scheme got such (relatively speaking) reasonable performance with such simple techniques: the PIC diagrammed here doesn't inline Fixnum#div or Double#div, but instead invokes them through a conventional call-return mechanism! (Though he does say "or possibly inlined" beneath it.)

The PSD.rb he mentions in his blogpost was also one of his early evaluation benchmarks (p. 108).

His name for "specialization" seems to be "splitting", or perhaps splitting is a certain kind of specialization.

Rails support

On p. 93 of the dissertation, he says TruffleRuby doesn't run Rails yet, even though a lot of his motivating examples about what makes Ruby difficult to implement efficiently come from Rails. But https://github.com/oracle/truffleruby says TruffleRuby *does* run Rails now:

TruffleRuby can run Rails and is compatible with many gems, including C

extensions. However, TruffleRuby is not 100% compatible with MRI 2.7 yet. Please report any compatibility issues you might find. TruffleRuby passes around 97% of ruby/spec, more than any other alternative Ruby implementation.

TruffleRuby might not be fast yet on Rails applications and large programs. Notably, large programs currently take a long time to warmup on TruffleRuby and this is something the TruffleRuby team is currently working on. Large programs often involve more performance-critical code so there is a higher chance of hitting an area of TruffleRuby which has not been optimized yet.

Apparently he got hired by Shopify, which uses Rails, but not to make this happen. There he confesses:

Baseline memory is often pretty high, and it takes memory to run our optimisations, but TruffleRuby when it's running then has optimisations to reduce memory used for each request, such as removing object allocations, zero-copy strings, and so on. Realistically TruffleRuby is designed for larger deployments serving many users, and probably isn't suite for a 500 MB $5 instance, this is true.

Disappointments

He kind of begs off evaluation of warmup time, startup performance, and memory usage, saying that he doesn't know how to evaluate them, that they aren't important for his purposes, and anyway he wants to trade them off for higher peak performance. It would seem to me that in order to trade something off you would need to know how much of it you have and how much of it you're paying in order to know if it's a good tradeoff. Overall I think this is sort of a problem with the JVM in general.

A disappointing thing about the dissertation is that it scales all the reported performance results by an opaque fudge factor, so none of them are individually falsifiable, are comparable to other published results, and none provide even an order-of-magnitude estimate of performance that could be used for other purposes. For example, perhaps an alpha-blending algorithm is reported, and in TruffleRuby it can alpha-blend 2.7 times as many pixels per second as in MRI and 2.3 times as many as in Rubinius. Does that mean it is alpha-blending 2700 pixels per second, 2.7 million, or 2.7 billion? This information is nowhere to be found, not in Figure 5.4 or anywhere else in the dissertation.

It's profoundly disappointing to find this statement on p. 94:

Another advantage is that [microbenchmarks] are typically very well understood by researchers which allows new implementations to be quickly tuned to run them well, but they are highly unrepresentative of real Ruby code. For example, nobody is making money by running a web service in Ruby to provide solutions to n-body problems.

This implicit equation of "real Ruby code" with "making money" perhaps explains some of the previous disappointments: he seems to consider code to solve physics problems to be 'fake code' because what he values is making money, not whatever else you might be able to achieve by solving physics problems, such as understanding things more deeply. And I guess nobody was paying him to polish this dissertation... but I wouldn't say that made it a "fake dissertation". But maybe that's why it says things like this (pp. 75–76):

This means that adding additional cores will not reduce the response time for a single customer. [sic] Increasing the memory capacity of a server is more easy as the limit [sic] on the quantity of memory that can be attached to a server is very high and a single sequential process [sic] can take advantage of all memory attached to a system.

Memory usage and performance

Overall the statistical performance evaluation that is the centerpiece of the dissertation's presented results is sort of disappointing. Not only does he give up on reporting any absolute performance numbers that would be comparable to other publications, but also he throws up his hands at statistical significance testing, taking independent sample measurements, and in general on quantifying the sources of error in his measurements. ("However, there are also natural reasons why there may be a visual pattern [in a Kalibera–Jones lag plot] — some benchmarks just appear to be cyclic in nature, no[] matter how long they are given to warm up." (p. 87), though he seems not to have actually included any lag plots of his own data in the dissertation; the plots on this page are from the Kalibera and Jones paper.)

This gives me more appreciation for why he didn't post any details of allocation cost in the HN thread at the level of detail I was looking at: he's considering systems that are so complex and poorly understood that it would be misleading to say, "this is the code the compiler generates in this case," because that is very dependent on an immense and delicate web of circumstances. Moreover, he was mostly chasing large and obvious speedups, not small and subtle ones, so even a very vague idea of what was going on would be sufficient.

He says it's "easier to scale memory than processor power", which is true in a sense — if your *request latency* is limited by memory usage, in 02021 or even in 02014 when he wrote the dissertation, it's much easier to get a machine you can jam twice as much RAM into as to get a machine running at twice the clock speed.

But "scaling" generally isn't about latency but about throughput: for a network service, cutting your latency from 100 ms to 10 ms is excellent, from 10 ms to 1 ms is useful, from 1 ms to 100 μs is sometimes good, from 100 μs to 10 μs is pointless, and from 10 μs to 1 μs is probably unobservable. *Scalability* of latency is a non-goal. (Seaton seems to be aiming lower than this, though: in one case (p. 86) he compares a hypothetical 1-second latency to a hypothetical 100-second latency, which is a dispiriting order of magnitude to be considering.)

By contrast, *throughput* is eminently scalable. If your Ruby web service serves 10 requests per second on one EC2 m5.xlarge instance, it will probably serve 1000 requests per second on 100 of them, and if you set up some readslaves (or let Amazon do it) you can probably get 100,000 requests per second on 10,000 m5.xlarge instances, and maybe you can scale to a million or ten million requests per second. There are some network services that do more, though usually not that inefficiently. In general, you can often usefully scale by several orders of magnitude more than you can scale latency.

So it makes sense to ask what the limiting resource is for scaling the throughput of a given application: CPU? Memory? Memory *bandwidth*? L1 cache? Disk bandwidth? Network bandwidth? Disk latency, or rather its reciprocal? There is *almost surely* one limiting resource at any given time, and increasing the other resources — or, equivalently, using them more efficiently — will not increase throughput. Seaton implicitly points this out when he says (p. 80):

…the application may create caches that are only cleared when there is memory pressure. This makes us consider whether optimising for low memory usage is even a desirable goal. If we have memory available in the system, why not use it?

This is a sensible question if the limiting resource is something other than memory, such as CPU time, or if your evaluation objective is latency rather than throughput. But a very common case in everyday web services is for memory rather than the other possibilities above to be the limiting resource. httpdito can handle some 25'000 HTTP requests per second on my laptop in part because it only uses one 4KiB page of memory per concurrent request. AWS Lambda usage is presently billed at seventeen nanodollars per gibibyte millisecond, starting at 128 MiB; a "lambda function" that uses one gibibyte costs eight times as much to run per millisecond as one that uses 128 mebibytes. (Though AWS does claim they scale CPU availability proportionally.)

A typical sort of situation is a frontend host dynamically generating a web page using a few dozen SQL queries sent to a remote database server, perhaps spending 64 ms in all waiting on SQL replies. During these 64 ms, the frontend host can spend its CPU time serving other concurrent web page requests, but it cannot spend its RAM on those other requests — the RAM is tied up until it gets the response back. Perhaps each request only take 2 ms of computation, so if it is CPU-bound, each core can do the computation for 32 other web pages while it's waiting for the SQL responses. If it has 4 cores, its response times will remain consistent until it's processing 128 concurrent requests, about 1900 requests per second.

But, if processing each of those requests is using 256 mebibytes of RAM, it needs 32 gibibytes of RAM to reach this level of concurrency. The m5.xlarge instance type has 4 (virtual) 3.1GHz Xeon Platinum 8175M cores but only 16 gibibytes of RAM, so at this performance level, its CPU couldn't be more than 50% utilized in this scenario. If the amount of time spent waiting on I/O is higher, even smaller memory usage becomes a problem; by contrast, faster I/O, like SSDs, reduces memory demands proportionally. Typically SSDs are around 128 times lower latency than spinning rust — typical NAND Flash chips take 16–128 μs to read a 2048-byte page rather than the 8192 μs typical of spinning rust (p. 143), so there is the opportunity to redesign the system architecture for higher throughput by getting better CPU efficiency at the expense of RAM efficiency.

Actually, this shift seems like maybe a big opportunity for things like TruffleRuby.

Conflating metaprogramming with reflection

Frequently the dissertation says "metaprogramming" when it means "reflection" or even specifically "reflection" or even more specifically "dynamic method invocation"; though, in places (§5.2, p. 102) it acknowledges that macro expansion and runtime code generation are also metaprogramming, in others it contradicts this:

> However metaprogramming has not received the same research attention and is often not optimised, even in mature language implementations. … In Ruby, probably more so than in other languages, metaprogramming should not be viewed by implementers as a side-channel that does not need to be optimised, but instead as just another form of dispatch. (p. 101)

This is a good example of *metaprogramming* being used to make the program

simpler (from the perspective of the Ruby community) but *the dynamism* not actually being needed in practice. (p. 108, emphasis mine)

Dispatch chains

Chapter 5 outlined how dispatch chains in TruffleRuby work, and presented measurements showing that in some cases they produce good performance, but it sort of handwaves about how this happens; all the magic remains behind Graal's curtain:

> Our second contribution in this chapter is to observe that the structures we have described above are trees, and so can be implemented using the same nodes as we use to implement the Ruby AST. Each condition in the chain is implemented as a node with a child node for the cache-miss case, and a child node for the cache-hit case, in the case of first level of caching, or a reference to the method to be called in the case of the second level of caching. … we rely entirely on the partial evaluation in Truffle's Graal backend to remove the degree of freedom in the method name if we are not using it. … we have confidence in the partial evaluation phase of Truffle's Graal backend to propagate that constant and entirely constant fold and remove the logic in the dispatch chain that handles varying method names. (pp. 106–7).

It would be useful to see the output of the partial evaluation phase of Truffle's Graal backend, or of the entire backend, in order to understand whether this confidence is justified, and if so, under what circumstances. Instead all we get are execution-time boxplots from which the absolute units of time have been carefully erased.

Topics

- Performance (p. 1155) (22 notes)
- Safe programming languages (p. 1172) (11 notes)
- Compilers (p. 1178) (10 notes)
- Virtual machines (p. 1182) (9 notes)
- Garbage collection (p. 1255) (4 notes)

.xosm: experimental obvious stack machine

Kragen Javier Sitaker, 02021-03-21 (updated 02021-03-24)
(20 minutes)

This is an unfinished twigman outline of a simple computer — more complex than its inspiration Calculus Vaporis, but perhaps more practical.

The foolish fill their coffee cups to the brim, in their greed unable to forgo a single drop unless the cup cannot hold it, and thus scald their hands when slightly jostled. The wise use slightly bigger cups.

The .xosm is a virtual machine design with byte-addressable RAM, eight 32-bit architectural registers (X Y Z T PC CP S D), and 16-bit instruction words. It is intended to be nearly as minimal as possible, but leave enough space at the top of the cup to avoid being penny-wise and pound-foolish, to mix a metaphor. Here are some of the pitfalls I'm hoping to steer this ship between, to add two more incompatible metaphors to this witch's brew of too many cooks, with attempted operationalizations:

- It should not be too hard to implement — it should take less than a day, or 132 source lines of C, working from the spec and test suite, for which purpose it should have less than 100 instructions;
- It should not be too slow to interpret — not more than 16 clock cycles per bytecode instruction on a modern superscalar CPU;
- It should not be too slow if compiled — not more than 2 clock cycles per bytecode instruction on a modern superscalar CPU;
- It should not be too large if implemented in hardware — more than 4096 logic gates;
- It should not be too awkward to program for by hand — not more than twice as much assembly code as for amd64, though probably more than twice as many instructions;
- It should not be too hard to compile C to, imposing neither an enormous performance penalty nor ridiculously complicated bytecode nor extremely complicated compilation tactics — no more than, say, four times worse than handwritten bytecode.
- It should not be too hard to debug programs for.

16-bit instruction words are a compromise. They occupy about 50% more space than 8-bit instructions (like Elisp or the 6502), but less than 32-bit instructions (like MIPS, SPARC, Lua, ARM, or RISC-V without the C extension). The instruction word consists of an opcode byte and an operand byte, but most opcodes do not use operands.

Compared to 8-bit encoding, 16-bit encoding reduces the number of cases where an instruction is followed by an immediate operand, and it allows the use of 16-bit-wide memory without alignment efficiency concerns; 32-bit immediates can be fetched in two memory operations rather than the 4 that would be needed with 8-bit alignment. The opcode byte can use a simpler encoding that simplifies instruction decode.

Compared to 32-bit encoding, 16-bit encoding uses a lot less space.

Operand registers

The .xosm has a four-register operand stack, whose registers are called X, Y, Z, and T (for time — introduced in the HP-35). X can be usefully thought of as the CPU's accumulator. Most instructions take implicit arguments on this stack and return results there; for example, the `x += y` instruction (0x2b) adds Y to X, and the `x -= y` instruction subtracts Y from X. Each of these instructions also pops the stack. Popping the stack consists of overwriting Y with Z and Z with T. Oddly, T, rather than retaining its value as you would expect, gets the old value of X, as explained below in the section about reversibility. There is a `;` instruction that just pops the stack without doing anything else.

Some instructions push the stack instead before whatever other actions they take. Pushing the stack consists of overwriting T with Z (losing the previous value of T), Z with Y, and Y with X. For example, the `x = *s` instruction (see below) pushes the stack before overwriting X with a value loaded from memory at the address in index register S. The `y = x` or `dup` instruction *only* pushes the stack without doing anything else. Immediate-load instructions like `x = 1` push the stack before setting X to a constant. There are two immediate-load opcodes, one which sets X to the operand byte, and one which is followed by a 32-bit immediate argument to set X to.

Single-operand ALU instructions like `x = ~x`, `x++`, and `x /= 2` neither push the stack nor pop it; they merely overwrite the X register with their result.

The four-level operand stack permits the evaluation of even relatively complex nested arithmetic expressions before having to fetch and store temporaries in memory, as well as providing a more convenient way to pass up to four parameters to subroutines than is common in assembly languages.

Here's a tentative full list of ALU/operand-stack instructions:

```
x += y
x -= y
x &= ~y
x &= y
x ^= y
x = ~x
x = y  # ;
y = x  # dup
x = 0
x++
x--
x += x  # x <<= 1
x <<= 3
x /= 2  # x >>= 1
x /= 8  # x >>= 3
x = k8  # 8-bit immediate
x = k32 # 32-bit immediate
```

These are 17 ALU instructions, which seems like a reasonable set

compared to 16 in Wirth-the-RISC, 6 in Chifir, 21 in LuaJIT, 4 in SWEET-16 if we categorize the comparisons as control-flow instructions, and 7 in the MuP21 or F21.

XXX maybe provide rotates instead of shifts?

Pointer registers

XXX from looking at the RTL this is still a little muxier on the hardware side than having a single architectural A register like the MuP21, and of course involves more instructions, although maybe it's better for software. Maybe you could have a "wielded pointer register" and an "alternate pointer register".

The .xosm has two 32-bit pointer or index registers, S and D. S is used for reading from memory (loads), while D is used for writing to memory (stores). Normal load instructions push the stack and store the result in operand register X, though there are two "leap" instructions that store it instead in S or D; all the store instructions write the contents of register X to memory before popping the operand stack. Commonplace address arithmetic can be done within the S and D registers rather than requiring the use of the operand stack; there are instructions for bumping them by small (8-bit) immediate constants ("creep"), adding them to large (32-bit) immediate constants, adding the program counter to them, shifting them left by 2 bits, and "leap"ping with the `d = *s` and `s = *s` instructions.

There are two instructions `x <=> d` and `x <=> s` to transfer the index registers to and from the operand stack. These instructions exchange X with, respectively, D and S, without pushing or popping the operand stack.

This segregation into fetch and store registers means that if you need a call stack (as C does!) you need to allocate a memory address to store your call stack pointer at. So it might be worthwhile to add an SP register and three instructions for it.

Tentatively here's the index-register instruction set:

```
*d = x
x = *(u32*)s  # if we go to 16-bit words then this can have an offset field
x = *(char*)s
d = *s  # leap d
s = *s  # leap s
s += k8  # immediate constant; 8-bit and 32-bit formats
s += k32
d += k8
d += k32
s += pc
d += pc
s <<= 2
d <<= 2
x <=> d
x <=> s
```

That's 15 opcodes.

Control flow

Like MIPS or RISC-V, there are no conditional flags, because the conditional instructions work on the contents of the operand stack; RISC-V chose this because it eases superscalar implementations, but for my purposes the big advantage is that software implementations don't have to bend over backwards to compute lots of data that's never used, which is a really bug-prone thing to do.

The .xosm has two architectural registers for control flow, the program counter PC and the continuation pointer CP, and a single control-flow operation `yield`, which swaps them, and can thus function as either a procedure call or return instruction. There are three `yield` instructions: unconditional (`else`), conditional on `x == 0` (`if (x)`), and conditional on `x >= 0` (`if (x < 0)`). The conditional instructions pop the operand stack. To enable control flow that goes beyond just two coroutines yielding back and forth, there's an `x <=> cp` instruction which exchanges CP and X, which simultaneously loads in a new continuation pointer (for example, pointing to another location within the same subroutine) and puts the old one in a location where you can save it to memory.

XXX do conventional short jumps too?

This approach is inspired by Henry Baker's COMFY-65 compiler and the Warren Abstract Machine, although it's also related to Calculus Vaporis. A very simple function like Forth's `: triple dup dup + + ;` might be implemented as nothing more than `dup dup + + else`; a nonrecursive function that calls other functions might save CP to a static memory location on entry and restore it before yielding on exit. I'd need more experience with the .xosm to really get a feel of what prologues and epilogues to use.

Instruction 0x00 is the "halt" instruction, because if you're executing uninitialized memory that's a bug. I don't know what it should do exactly.

Tentative control-flow instruction set:

```
if (x) ...
if (x < 0) ...
else
halt
```

4 opcodes.

Reversibility

For debugging, backwards execution and efficient tracing and checkpointing are obviously very desirable. So many of the .xosm's operations are defined to erase as little information as possible, reducing the volume of information that must be logged for a reverse-executable trace. The `yield` instructions erase only one bit of information — whether the previous instruction execution was a `yield` or not, and thus whether the previous program counter is in PC-1 or in CP-1 — and because the two-operand ALU instructions save both the result and one of the operands (the previous value of X, which is saved in T) they are fully reversible as well if the underlying ALU operation is. The various register-swap instructions are also fully

reversible. The non-reversible operations are:

• Everything that pushes the operand stack, including loads from memory and immediate-load instructions. Immediate-load instructions not only erase the previous T value but also, when the immediate is not embedded in the opcode byte, the previous value of PC — some suffix of the immediate constant might also be a valid instruction.
• Store instructions.
• Irreversible ALU operations such as `x &= y`.
• `dup`, which you could consider an "irreversible ALU operation".

This may also have implications for efficient hardware implementation, as the tsunami in advance of the Landauer-limit earthquake seems to be arriving already.

Twigman evaluation

This is 17+15+4 = 36 opcodes, which seems perhaps a bit more oversimplified than I would like, but will probably grow to the size I want when I get some experience with its deficiencies.

A couple of sample instruction implementations in an interpreter on a 64-bit machine might be:

```
xor:
    tmp = y;
    y = z;
    z = t;
    t = x;
    x ^= tmp;
    goto *opcodes[mem[pc++] & 0xff];
leap_d:
    d = mem[s];
    goto *opcodes[mem[pc++] & 0xff];
```

These probably work out to 9 and 6 instructions respectively, including a jump with a failed prediction, so I think I'm within my target performance zone for interpretation of 16 clock cycles per bytecode. It's also 5 lines of C per opcode, but some of that can and should be factored out into an inline function, and then it will probably be within my lines-of-code complexity budget.

```
xor:
    x ^= pop_operand_stack();
    goto dispath;
```

8 architectural registers is a good, practical, 8080ish size; we'd also need a non-architectural instruction register I, a memory-address register A, a memory-read register M, and some kind of microcycle state machine. 32 bits may be a bit excessive for simple hardware implementation; one flip-flop per bit means we need 256 flip-flops for just the architectural registers. If you implement it with an 8-bit ALU and 8-bit data paths you can probably hit the 4096-gate target I set at the top at the expense of a slowdown of 4× or so.

A rough sketch of the RTL:

```
X <= ALU-output if ALU-instruction else
     operand-byte if load-immediate-8 else
     M if fetching-into-X else
     S if s-swapping else
     D if d-swapping else
     CP if cp-swapping else
     XXX if load-immediate-32 else
     X

Y <= X if pushing else
     Z if popping else
     Y

Z <= Y if pushing else
     T if popping else
     Z

T <= Z if pushing else
     X if popping else
     T

PC <= CP if yielding else
      PC+6 if immediate-32 else
      PC+2

CP <= PC+2 if yielding else
      X if cp-swapping else
      CP

S <= M if s-leaping else
     X if s-swapping else
     s-effective-address if s-creeping else
     S << 2 if s-shifting else
     S

D <= M if d-leaping else
     X if d-swapping else
     d-effective-address if d-creeping else
     D << 2 if d-shifting else
     D

A <= s-effective-address if loading else
     d-effective-address if storing else
     PC if fetching else
     0

ALU-output = sum if adding else
             diff if subtracting else
             abj if abjuncting else
             conj if anding else
             xor if xoring else
             negated if negating else
             Y if discarding else
             0 if zeroing else
             incremented if incrementing else
```

```
              decremented if decrementing else
              double if doubling else
              octuple if octupling else
              half if halving else
              eighth if eighthing else
              tristate

sum         = X + Y     # N - ½ full adders
diff        = X - Y     # same
abj         = X & ~Y    # N AND gates
conj        = X & Y     # same
xor         = X ^ Y     # N XOR gates
negated     = ~X        # N NOT gates
incremented = X + 1     # N half adders
decremented = X - 1     # same
double      = X << 1    # just wires
octuple     = X << 3
half        = X >> 1    # also just wires. unsigned
eighth      = X >> 3

yielding = unconditional-yield ∨
           zero-conditional ∧ X == 0 ∨
           sign-conditional ∧ X[31]
```

The "XXX if load-immediate-32" case and the A register point out that sometimes extra cycles will be needed during which almost all of the above will be paused, because it's fetching an immediate 32-bit value (possibly unaligned). If I want to build up an RTL design incrementally I probably want to start with those troublesome cases so the control state machine starts out as complicated as it's going to get.

But I think we can sort of reasonably estimate the above as about 27 N-wide 2-muxes or tristate buffers for control and another 14 for ALU result selection, and another 9 or so for things I haven't thought of yet, 50 in all; here "N-wide" means whatever width the internal data paths for 32-bit data are, which might be 32 bits for a fast implementation or 4 or 8 bits for a small one. The ALU needs about 16N gates, maybe a bit more for lookahead carry. We can sort of reasonably ballpark this at 400 gates of muxing for an 8-bit implementation, plus 128 gates of ALU, which seems like an unreasonably small ALU by comparison. With 32-bit data paths these would be 1600 gates of muxing and 512 gates of ALU.

There's a separate N-wide AND for the `X == 0` condition and some more muxes and adders for effective address computation, something like this:

```
s-effective-address = S + operand-byte if s-creeping else
                      S + PC if pc-relative else
                      S
d-effective-address = D + operand-byte if d-creeping else
                      D + PC if pc-relative else
                      D
```

The instruction decode logic depends on the instruction encoding,

but the above strawman has it and the microcycle logic producing the following control bits: `ALU-instruction`, `fetching-into-X`, `load-immediate-32`, `load-immediate-8`, `s-swapping`, `d-swapping`, `cp-swapping`, `pushing`, `popping`, `yielding`, `imm32`, `imm8`, `s-leaping`, `s-creeping`, `s-shifting`, `d-creeping`, `d-shifting`, `loading`, `storing`, `fetching`, `adding`, `subtracting`, `abjuncting`, `anding`, `xoring`, `negating`, `discarding`, `zeroing`, `incrementing`, `decrementing`, `doubling`, `octupling`, `halving`, `eighthing`, `unconditional-yield`, `zero-conditional`, and `sign-conditional`. That's 38 control signals, and probably something like 9×38 = 342 two-input AND gates to compute them, if that's how it's done, or possibly a much smaller number of wider AND gates.

So we've only accounted for about 1600+512+342 ≈ 2500 gates of an internally-32-bit implementation, ten thousand transistors. The 8 architectural registers and 3 non-architectural registers add 352 flip-flops, probably another 3000 transistors, for a total of 13000. If that were the whole story, this design's transistor count would be between the 9000-transistor 8-bit 6809 and the 29000-transistor 16-bit 8086, both from 01978, nowhere near the 68000-transistor 68000, which was 32-bit architecturally but 16-bit internally, much less the 190k-transistor 68020 (01984) or the 275k-transistor i386 (01985). It's even substantially smaller than the ARM 1 (25000 transistors, 01985), but it's close to Chuck Moore's 16-bit Novix NC4016 (16000 transistors, also 01985). Most likely I just haven't noticed the majority of the transistors that are needed to make the .xosm actually run. Where are they?

(However, Moore's later 21-bit MuP21 design (01994), one of the design inspirations for for the .xosm, was only 7000 transistors, including an NTSC-generation coprocessor.)

It probably isn't extremely useful to keep a general-purpose CPU much smaller than 16384 transistors, like 4096 2-input NAND gates, unless your RAM is a drum or an acoustic delay line or something. The COSMAC VIP, the early personal computer where we get the CHIP-8 videogame virtual machine design, shipped with 2 KiB of RAM, which was probably 6T SRAM: 16384 bits and 98304 transistors of RAM. Now we'd use 16384 capacitors and 16384 transistors of DRAM, plus 128 6-transistor sense amplifiers along the edge. But Wozniak thought 4 KiB was the minimum to run a usable BASIC on the Apple, and he was likely right, although the x18 GreenArrays cores make do with 64 words of RAM (and 64 of ROM) per core, forcing you to split all but the smallest programs across multiple of the 144 cores on the chip. If you already have 32768 components in your RAM, then whatever benefit you get from reducing your CPU from 16384 components to 8192 is probably not worth the sacrifices required.

Some preliminary notes on the amazing RISC-V architecture (p. 82) mentions that Claire Wolf's PicoRV32 RISC-V design can be configured to run in 761 slice LUTs on a Xilinx 7-series FPGA, uses 48 LUTs as memory, and also 442 slice registers; I think those are 4-LUTs, which can compute any arbitrary 4-input Boolean function, so that's roughly equivalent to 2300 2-input NAND gates and 500 flip-flops, which seems pretty comparable to the .xosm, actually, but supporting interrupts and a wider range of operations and stuff. I should check out Wolf's design.

picorv32.v is 1913 unique lines of Verilog so I'm not sure where to start! It's enormous. I think the interrupt controller is compiled out in the small configuration I mentioned above, though.

Topics

- Virtual machines (p. 1182) (9 notes)
- Instruction sets (p. 1214) (6 notes)
- Stack machines (p. 1320) (2 notes)
- MuP21

Open coded primitives

Kragen Javier Sitaker, 02021-03-22 (26 minutes)

As I was reading Chris Seaton's dissertation, I was thinking about compilation strategies for late-bound languages. I think there's a straightforward ahead-of-time compilation strategy that would maybe give a significant fraction of optimal (CPU) performance, maybe 30%, while retaining full dynamic-dispatch semantics.

Experience from Ur-Scheme

My toy compiler Ur-Scheme got surprisingly good performance. About 20% of GCC on the dumb fibonacci microbenchmark at the time (though see below about how GCC has improved!), and about 250% of Chicken's performance on itself compiling itself, i.e., when compiled with Chicken, it took 2½ times as long to compile itself as when compiled with itself. One technique it used for this was to open-code primitive operations following a type check. For example, each call to the `string-length` function was converted into the following code:

```
    # string-length inlined primitive
    call ensure_string
    lea 8(%eax), %ebx
    push %ebx
    movl 4(%eax), %eax
    pop %ebx
    sal %eax
    sal %eax
    inc %eax
```

This was generated by this Scheme code:

```
(define (inline-string-length nargs)
  (assert-equal 1 nargs)
  (comment "string-length inlined primitive")
  (extract-string)
  (asm-pop ebx)
  (native-to-scheme-integer tos))
```

The `ensure_string` millicode routine could also have been open-coded and perhaps should have:

```
ensure_string:
        # test whether %eax has magic: 0xbabb1e
        # first, ensure that it's a pointer, not something unboxed
        test $3, %eax
        jnz k_2
        # now, test its magic number
        cmpl $0xbabb1e, (%eax)
        jnz k_2
        ret
```

But on modern CPUs, this is a fetch, four ops after macro-op fusion, and no mispredicted branches or cache misses, so it's not extremely expensive. In Ur-Scheme, the handling of failed type checks like these is nasty, brutal, and short; the program just spews out an error to stderr and peremptorily exits.

```
        .section .rodata
        # align pointers so they end in binary 00
        .align 4
_stringP_4:
        .long 0xbabb1e
        .long 20
        .ascii "error: not a string\n"
        .text
k_2:
        movl $_stringP_4, %eax
        jmp report_error
report_error:
        call ensure_string
        lea 8(%eax), %ebx
        push %ebx
        movl 4(%eax), %eax
        # fd 2: stderr
        movl $2, %ebx
        movl %eax, %edx
        pop %ecx
        movl $4, %eax
        int $0x80
        movl $1, %ebx
        movl $1, %eax
        int $0x80
```

But, at the point that this happens, *the address of the failing code is on the stack*. You could look at it and see what operation it was trying to apply, then fall back to a generic method dispatch, then jump back to where the result from `string-length` was expected.

If you open-code the check, such fallbacks would be a little more complicated, because i386 and amd64 don't have the conditional call instructions their predecessor the 8080 had, so you can't just blithely jump to `k_2` — you'll never get back! You'd have to jump to a trampoline generated for just that callsite, which, as we'll see later, is what SBCL and OCaml do.

In some cases I *did* open-code the dynamic type checks. The code `(char->integer #\0)`, for example, gets compiled as follows:

```
    push %eax
    movl $2 + 48<<2, %eax
    test $1, %eax
    jnz not_a_character
    test $2, %eax
    je not_a_character
    cmpl $2 + 256<<2, %eax
    jnb not_a_character
    dec %eax
```

This code is obviously kind of stupid; among other things, we know `#\0` is a character because it's a compile-time character constant, and furthermore the whole expression should just constant-fold to something like `movl $193, %eax`.

So it surprised me that this kind of nonsense was still faster than what most Scheme compilers were doing: the extra 5 or 6 (!) instructions and the millicode calls and returns are still faster! However, Chez Scheme has been open-sourced since then, and both it and probably Ikarus likely generate higher-quality code.

SBCL: almost twice as fast as Ur-Scheme, before adding declarations

SBCL *does* do the whole thing I was saying above where you could open-code your type check followed by open-coding the primitive operation. Here we see how it handles `cdr` on what it hopes is a list:

```
This is SBCL 1.0.57.0.debian, an implementation of ANSI Common Lisp.
...
* (defun mylen (lst) (if (null lst) 0 (mylen (cdr lst))))

MYLEN
* (mylen '(3 5 1))

0
* (mylen '(3 5 . 1))

debugger invoked on a TYPE-ERROR in thread
#<THREAD "main thread" RUNNING {1002978D43}>:
  The value 1 is not of type LIST.
...
(MYLEN 1)
0] 0
```

So when we invoked it on an improper list, `cdr` crashed. What does the underlying code look like?

```
* (disassemble 'mylen)

; disassembly for MYLEN

; 029B5D48:       4881FE17001020   CMP RSI, 537919511          ; no-arg-parsing en
try point
;       4F:       7508             JNE L0
```

That ridiculous number, 0x20100017, is NIL. This is what the `(if (null lst) ...)` compiled to.

```
;       51:       31D2             XOR EDX, EDX
;       53:       488BE5           MOV RSP, RBP
;       56:       F8               CLC
;       57:       5D               POP RBP
;       58:       C3               RET
```

That's "return 0". In SBCL the integer type-tag is 0 in the low bit, so 0 as an integer is really 0 (in `EDX`), and in SBCL's bizarre calling convention, the carry flag needs to be clear to indicate that this isn't a multiple-value return.

Now we are going to see what `cdr` looks like. First we check to see if the argument is a list:

```
;       59: L0:   8BC6             MOV EAX, ESI
;       5B:       240F             AND AL, 15
;       5D:       3C07             CMP AL, 7
;       5F:       751E             JNE L1
```

So that's the open-coded type test, four instructions, then a predicted-not-taken branch to an error-handling trampoline welded onto the end of the function. When there are multiple such checks in a function, each check gets its own trampoline. So now we have the open-coded implementation of `cdr` itself:

```
;       61:       488BC6           MOV RAX, RSI
;       64:       488B5001         MOV RDX, [RAX+1]
```

That's it, that's all of `cdr`. Actually half of that was just a totally unnecessary register move. So then all that's left is the function's tail call:

```
;       68:       488B0581FFFFFF   MOV RAX, [RIP-127]          ; #<FDEFINITION obj
ect for MYLEN>
;       6F:       B902000000       MOV ECX, 2
;       74:       FF7508           PUSH QWORD PTR [RBP+8]
;       77:       FF6009           JMP QWORD PTR [RAX+9]
```

Then we have two error trampolines on the end of the function, the first of which has no visible callers:

```
;       7A:       CC0A             BREAK 10                    ; error trap
;       7C:       02               BYTE #X02

;       7D:       18               BYTE #X18                   ; INVALID-ARG-COUNT
-ERROR
;       7E:       54               BYTE #X54                   ; RCX
```

But the other is the one we invoked above:

```
;       7F: L1:   CC0A             BREAK 10                    ; error trap
;       81:       04               BYTE #X04

;       82:       02               BYTE #X02                   ; OBJECT-NOT-LIST-E
RROR
;       83:       FE9501           BYTE #XFE, #X95, #X01       ; RSI
```

The trap instruction here is followed by five bytes that the trap

handler presumably can load, by indexing off the program counter the BREAK saves on the stack, and interpret as it wishes.

Perhaps a more illuminating function for these purposes would be something that conses, because when it conses it has to have an "exception handler" for the nursery being full, in which case it beseeches the garbage collector for memory, similar to falling back to generic method dispatch. We'll just wrap the built-in `cons` function.

```
* (defun mycons (a d) (cons a d))

MYCONS
* (cons 'x '(30 2))

(X 30 2)
* (disassemble 'mycons)

; disassembly for MYCONS

; 02A2D7E8:       49896C2440       MOV [R12+64], RBP           ; no-arg-parsing en
try point
```

That instruction has to do with SBCL's "pseudo-atomic" handling of interrupts during memory allocation. I think R12 is where SBCL keeps a pointer to thread-local data (from a note on Steve Losh's blog and a note on Paul Khuong's blog) and that [R12+64] in particular is `thread.pseudo-atomic-bits`.

Next we load the (thread-local!) allocation pointer `thread.alloc-region` into L11 and bump it by 16 bytes:

```
;       7ED:       4D8B5C2418       MOV R11, [R12+24]
;       7F2:       498D4B10         LEA RCX, [R11+16]
```

But now we need to see if the new candidate allocation pointer is still within the nursery:

```
;       7F6:       49394C2420       CMP [R12+32], RCX
;       7FB:       7628             JBE L2
```

If not, we jump off the fast path to the garbage-collection trampoline at L2; but normally we continue on the fast path:

```
;       7FD:       49894C2418       MOV [R12+24], RCX
;       802:       498D4B07         LEA RCX, [R11+7]
```

So now we have our newly allocated dotted pair; it cost us six instructions, plus the following three instructions for deferred interrupt handling:

```
;       806: L0:   49316C2440       XOR [R12+64], RBP
;       80B:       7402             JEQ L1

;       80D:       CC09             BREAK 9                     ; pending interrupt
 trap
```

The L0 there is where the slow path rejoins the main flow of the program. Normally we expect the value at [R12+64] to be unchanged from when we stored RBP there earlier, but if not, we handle the pending interrupt here.

Now that we're done with the open-coded memory allocation, all that's left is the open-coded initialization of `cons` itself and then the function epilogue:

```
;       80F: L1:   488959F9         MOV [RCX-7], RBX
;       813:       48897901         MOV [RCX+1], RDI
;       817:       488BD1           MOV RDX, RCX
;       81A:       488BE5           MOV RSP, RBP
;       81D:       F8               CLC
;       81E:       5D               POP RBP
;       81F:       C3               RET
```

And then we have the exception handlers that are welded onto the end of the function:

```
;       820:       CC0A             BREAK 10                         ; error trap
;       822:       02               BYTE #X02

;       823:       18               BYTE #X18                        ; INVALID-ARG-COUNT⚬
⚬-ERROR
;       824:       54               BYTE #X54                        ; RCX
```

And here's the one we came for, beseeching the garbage collector for 16 precious bytes for our `cons`:

```
;       825: L2:   6A10             PUSH 16
;       827:       4C8D1C2570724200 LEA R11, [#x427270]              ; alloc_tramp
;       82F:       41FFD3           CALL R11
;       832:       59               POP RCX
;       833:       488D4907         LEA RCX, [RCX+7]
;       837:       EBCD             JMP L0
```

Although SBCL doesn't, you could use precisely the same approach, with the inlined primitive fast path and falling back to a slow path, for generic method dispatch.

(Above I've said that I think the allocation pointer is thread-local, but I haven't observed wonderful scalability running allocation-heavy code in multiple SBCL threads.)

SBCL does about 57 million dumbfib leaf calls per second on this laptop without declarations, dispatching all its arithmetic through generic functions:

```
* (defun dumbfib (x) (if (< x 2) 1 (+ (dumbfib (1- x)) (dumbfib (1- (1- x))))))

DUMBFIB
* (mapcar #'dumbfib '(0 1 2 3 4 5 6 7))

(1 1 2 3 5 8 13 21)
* (time (dumbfib 35))
```

```
Evaluation took:
  0.263 seconds of real time
  0.264016 seconds of total run time (0.264016 user, 0.000000 system)
  100.38% CPU
  734,384,636 processor cycles
  0 bytes consed

14930352
```

The surprising comparison with GCC (11× as fast) and Ur-Scheme (half as fast as SBCL)

GCC by comparison does about 650 million leaf calls per second, 11 times as fast.

```
$ cat fib.c
/* ... */
__attribute__((fastcall)) int fib(int n)
{
    return n < 2 ? 1 : fib(n-1) + fib(n-2);
}
main(int c, char **v) { printf("%d\n", fib(atoi(v[1]))); }
$ gcc -O3 fib.c -o fib
fib.c:7:1: warning: ‘fastcall’ attribute ignored [-Wattributes]
fib.c: In function ‘main’:

fib.c:10:25: warning: incompatible implicit declaration of built-in function ‘pri
ntf’ [enabled by default]
```

(The compiler is warning us here that `fastcall` here is not relevant on amd64 — it gives the compiler permission not to use the shitty inefficient calling convention specified by the standard i386 ABI.)

```
: user@debian:~/devel/dev3; time ./fib 40
165580141

real    0m0.254s
user    0m0.252s
sys     0m0.000s
```

However, comparing Ur-Scheme, I got a nasty surprise! Ur-Scheme's code presumably is the same as when I wrote it 13 years ago in 02008, but now it SUCKS compared to GCC — it's even slower than SBCL!

```
$ cat fib.scm
;; Dumb Fibonacci picobenchmark
(define (fib n) (if (< n 2) 1 (+ (fib (1- n)) (fib (1- (1- n))))))
(display (number->string (fib 35))) (newline)
$ ./urscheme-compiler < fib.scm > fib.s
: user@debian:~/devel/urscheme; gcc fib.s -o dumbfib
fib.s: Assembler messages:
fib.s:5: Error: operand type mismatch for `push'
...
```

```
$ gcc -m32 fib.s -o dumbfib
$ time ./dumbfib
14930352

real    0m0.661s
user    0m0.552s
sys     0m0.000s
```

That's only 27 million leaf calls per (user) second. Now instead of 20 *percent* of GCC's performance, it's getting one *twentieth*, or *five* percent. Actually *four* percent. I'm guessing that the shitty code it emits is a lot worse at exploiting modern superscalar OoO processors because it's unceasingly full of dependencies. The 20% number was on my 700MHz Pentium III! Valgrind says it runs "2,851,829,048" instructions, so that's about two instructions per cycle.

It's clear that I don't know much and should spend some time taking measurements!

Hmm, it looks like GCC went pretty hard in on the optimization here... the above one-line C `fib` function compiled to 169 instructions. I think GCC inlined it into itself 13 times! But `-fno-inline` only makes it slightly slower:

```
$ gcc -fno-inline -O3 fib.c -o fib
fib.c:7:1: warning: 'fastcall' attribute ignored [-Wattributes]
fib.c: In function 'main':

fib.c:10:25: warning: incompatible implicit declaration of built-in function 'printf' [enabled by default]
$ time ./fib 40
165580141

real    0m0.390s
user    0m0.388s
sys     0m0.000s
```

I mean that's still 430 million leaf calls per second, and some of those are real calls, although GCC still seems to have removed half of the recursion by realizing that integer addition is associative:

```
  400570:       55                      push   %rbp
  400571:       53                      push   %rbx

  400572:       89 fb                   mov    %edi,%ebx            # argument n is in rdi (...rsi, rdx, rcx...)
  400574:       48 83 ec 08             sub    $0x8,%rsp
  400578:       83 ff 01                cmp    $0x1,%edi            # n < 2?
  40057b:       7e 1f                   jle    40059c <fib+0x2c>    # otherwise:

  40057d:       31 ed                   xor    %ebp,%ebp            # initialize loop accumulator to 0
  40057f:       90                      nop

  400580:       8d 7b ff                lea    -0x1(%rbx),%edi      # edi ← n-1 (set up argument)
```

```
  400583:       83 eb 02                sub    $0x2,%ebx              # ebx ← n-2 (
callee-saved!)

  400586:       e8 e5 ff ff ff          callq  400570 <fib>           # (recursive 
call)

  40058b:       01 c5                   add    %eax,%ebp              # add return 
value to accumulator

  40058d:       83 fb 01                cmp    $0x1,%ebx              # loop termin
ation test

  400590:       7f ee                   jg     400580 <fib+0x10>      # loop back t
o the lea

  400592:       8d 45 01                lea    0x1(%rbp),%eax         # add one fin
al 1 to the accumulator for return

  400595:       48 83 c4 08             add    $0x8,%rsp              # function ep
ilogue
  400599:       5b                      pop    %rbx
  40059a:       5d                      pop    %rbp
  40059b:       c3                      retq

  40059c:       b8 01 00 00 00          mov    $0x1,%eax              # base case a
lmost never aken
  4005a1:       eb f2                   jmp    400595 <fib+0x25>
```

OCaml allocation

I wrote this code in OCaml:

```
let rec nlist n cdr = if n = 0 then cdr else nlist (n-1) (n::cdr)
let rec mnlist m n = if m = 0 then [] else (ignore (nlist n []); mnlist (m-1) n)
let m = 2000*1000 and n = 500 ;;

print_endline ("m=" ^ (string_of_int m) ^ " n=" ^ (string_of_int n)) ;
mnlist m n
```

This ran in about 2.3 seconds, thus consing a billion list nodes in 2.3 nanoseconds each. Here's the disassembled machine code of `nlist`:

```
Dump of assembler code for function camlTimealloc2__nlist_1030:
   0x0000000000403530 <+0>:     sub    $0x8,%rsp
   0x0000000000403534 <+4>:     mov    %rax,%rsi
   0x0000000000403537 <+7>:     cmp    $0x1,%rsi
   0x000000000040353b <+11>:    jne    0x403548 <camlTimealloc2__nlist_1030+24>
   0x000000000040353d <+13>:    mov    %rbx,%rax
   0x0000000000403540 <+16>:    add    $0x8,%rsp
   0x0000000000403544 <+20>:    retq
   0x0000000000403545 <+21>:    nopl   (%rax)
=> 0x0000000000403548 <+24>:    sub    $0x18,%r15
   0x000000000040354c <+28>:    mov    0x21780d(%rip),%rax        # 0x61ad60
```

```
   0x0000000000403553 <+35>:    cmp    (%rax),%r15
   0x0000000000403556 <+38>:    jb     0x403577 <camlTimealloc2__nlist_1030+71>
   0x0000000000403558 <+40>:    lea    0x8(%r15),%rdi
   0x000000000040355c <+44>:    movq   $0x800,-0x8(%rdi)
   0x0000000000403564 <+52>:    mov    %rsi,(%rdi)
   0x0000000000403567 <+55>:    mov    %rbx,0x8(%rdi)
   0x000000000040356b <+59>:    mov    %rsi,%rax
   0x000000000040356e <+62>:    add    $0xfffffffffffffffe,%rax
   0x0000000000403572 <+66>:    mov    %rdi,%rbx
   0x0000000000403575 <+69>:    jmp    0x403534 <camlTimealloc2__nlist_1030+4>
   0x0000000000403577 <+71>:    callq  0x411898 <caml_call_gc>
   0x000000000040357c <+76>:    jmp    0x403548 <camlTimealloc2__nlist_1030+24>
End of assembler dump.
```

The allocation fast path is just these five instructions, rather than the nine used by SBCL:

```
=> 0x0000000000403548 <+24>:    sub    $0x18,%r15
   0x000000000040354c <+28>:    mov    0x21780d(%rip),%rax        # 0x61ad60
   0x0000000000403553 <+35>:    cmp    (%rax),%r15
   0x0000000000403556 <+38>:    jb     0x403577 <camlTimealloc2__nlist_1030+71>
   0x0000000000403558 <+40>:    lea    0x8(%r15),%rdi
```

Here our allocation pointer moves *down* and is kept in %r15.

Ooh, I just learned that with `ocamlopt -S` I can coax the raw assembly out of ocamlopt, so here's the relevant part of the function:

```
.L102:  subq    $24, %r15
        movq    caml_young_limit@GOTPCREL(%rip), %rax
        cmpq    (%rax), %r15
        jb      .L103
        leaq    8(%r15), %rdi
```

That's the chunk I quoted twice above. Then the new cons node gets initialized:

```
        movq    $2048, -8(%rdi)
        movq    %rsi, (%rdi)
        movq    %rbx, 8(%rdi)
```

Then it builds up the arguments for the tail call:

```
        movq    %rsi, %rax

        addq    $-2, %rax      # n - 1, in OCaml's weird tagged integer representa
tion
        movq    %rdi, %rbx
        jmp     .L101
```

Finally, this is the "allocation trampoline", six instructions in the SBCL code:

```
.L103:  call    caml_call_gc@PLT
.L104:  jmp     .L102
```

In this case we just restart the allocation instead of asking the GC to do it for us.

C pointer-bump allocation

I tried the OCaml approach in C, getting even higher speeds:

```
kmregion *p = km_start(km_libc_disc, &err);
if (!p) abort();
struct n { int i; struct n *next; } *list = NULL;

for (size_t j = 0; j < 5000; j++) {
  struct n *q = km_new(p, sizeof(*q));
  q->i = j;
  q->next = list;
  list = q;
}

...
km_end(p);
```

Here `km_new` is defined in the header file as follows:

```
static inline void *
km_new(kmregion *r, size_t n)
{
  n = (n + alignof(void*) - 1) & ~(alignof(void*) - 1);
  size_t p = r->n - n;
  if (p <= r->n) {
    r->n = p;
    return r->bp + p;
  }

  return km_slow_path_allocate(r, n);
}
```

This manages to allocate 520 million list nodes per second, 1.94 nanoseconds per allocation (and initialization), 40% faster than OCaml despite not allocating a register for the allocation pointer. `km_slow_path_allocate`, which is separately compiled and thus not inlineable, invokes malloc through a function pointer from `km_libc_disc`, 32 kibibytes at a time, plus occasionally also invoking it to store the array of block pointers, and also invoking `longjmp` in case of allocation failure. Since all this happens one out of every 2048 allocations, the performance cost is normally insignificant:

```
void *
km_slow_path_allocate(kmregion *p, size_t n)
{
  if (n > block_size / 4) return km_add_block(p, n);
  void *block = km_add_block(p, block_size);
  if (!block) return NULL;
  p->bp = block;
  p->n = block_size;
```

```
  return km_new(p, n);
}
```

GCC does a better job than I would have done at optimizing the inner loop, scrambling it into a strange order that saves an unconditional jump per iteration:

```
        call km_start
        testq %rax, %rax
        movq %rax, %rbp             # %rbp = kmregion pointer
        je .L21
        xorl %ebx, %ebx             # j = 0
        xorl %r13d, %r13d           # list = NULL
        jmp .L6
.L23:
        movq 8(%rbp), %rax          # load base pointer
        movq %rdx, 0(%rbp)          # store allocation counter
        addq %rdx, %rax             # offset base pointer with allocation
.L8:                                #     pointer to get address
        movl %ebx, (%rax)           # store j in list node
        addq $1, %rbx               # increment j
        movq %r13, 8(%rax)          # store list pointer into list node
        cmpq $5000, %rbx            # loop counter termination test
        je .L22
        movq %rax, %r13             # point list pointer at new list node
.L6:
        movq 0(%rbp), %rcx          # load allocation counter
        leaq -16(%rcx), %rdx        # decrement it
        cmpq %rdx, %rcx             # check against its old value
        jae .L23                    # if it decreased it didn't underflow

        movl $16, %esi
        movq %rbp, %rdi
        call km_slow_path_allocate
        jmp .L8
.L22:
```

The fast-path allocation here is 7 instructions, split between movq leaq cmpq jae at the end of the loop, then movq movq addq at the beginning. The slow-path call is glued onto the end of the loop, thus saving the unconditional jump, but maybe welding it onto the bottom of the function as OCaml and SBCL do would be a better idea.

Failover in overflow cases

A number of late-bound languages (Smalltalk and Lisps, mostly, but also recent versions of Python, as well as Ruby, naturally) handle fixnum overflow transparently by failing over to bignum arithmetic, which surely causes difficulty with both type inference and performance predictability, but is sometimes worth the pain and suffering.

Above we saw OCaml compile `(n-1)` to

```
        addq    $-2, %rax
```

and we saw GCC compile the addition of two Fibonacci return values to

```
  40058b:       01 c5                   add    %eax,%ebp            # add return ⚬
⚬value to accumulator
```

You could imagine following such a single-instruction operation being followed by a conditional jump on overflow to a fixup trampoline welded onto the end of the function:

```
        addq    $-2, %rax
        jo      .tramp304
        ...
.tramp304:
        movq    $-2, %rdi
        movq    %rax, %rsi
        call    addition_overflow_handler
        jmp     .resume305
```

The COMFY-6502 "win/lose" mechanism is tempting here: we might be tempted to just treat such overflow traps as "lose continuations", but the fact is that we need to weld on a separate trampoline for each of them — they're more like conditionals. But they may never rejoin the original fast-path control flow: it may be convenient to preserve the property that all the integer values on the fast path are fixnums, so that we don't have to test their tags repeatedly. So the simple COMFY one-entry two-exits approach may not work as well as one might hope.

Alternatively, it may be perfectly adequate to just do the type test every time, though not as inefficiently as Ur-Scheme does; this might result in compiling `n - 1` to fast-path code like this:

```
        test    $1, %rax     # use the low bit as the type tag
        jz      .tramp303    # using 1 as the fixnum type tag, like OCaml
        addq    $-2, %rax
        jo      .tramp304
.resume305:
        ...
```

Note that the trampoline at `.tramp303` isn't limited to doing bignum arithmetic; it can do a fully general object-oriented dispatch, use an inline cache (polymorphic or not), and so on. This is probably not going to make your dynamic object-oriented system run as fast as C or OCaml — maybe 30% as fast — but it'll probably be substantially faster than SBCL.

Generic arithmetic with an open-coded fixnum path, and other fast paths

If you want to abjure overflow-to-bignum but still support generic arithmetic operations, just open-coding the common fixnum fast path, it's a little more lightweight:

```
        test    $1, %rax     # use the low bit as the type tag
        jz      .tramp303    # using 1 as the fixnum type tag, like OCaml
        addq    $-2, %rax
.resume305:
        ...
```

We should expect this tag test and conditional bailout to be similar in cost to the allocation guard instructions in my allocation loop above:

```
        cmpq %rdx, %rcx                 # check against its old value
        jae .L23                        # if it decreased it didn't underflow
```

We have an upper bound on that cost: the whole loop runs in 1.94 ns per iteration. And valgrind says this code runs about 6.7 billion instructions when running 100'000 `km_regions` of 5000 nodes each: about 13.4 instructions per loop iteration, so it's probably on the order of 300 ps per such test.

There are probably a couple dozen methods or operators to which you'd want to give an open-coded primitive path like this for normal code; depending on the language, perhaps integer arithmetic (+ - × ÷ // % == < > <= >= == 1+ 1-), bit operations (^ & | << >> >>> &^), floating-point arithmetic, if-then-else, range iteration, container iteration, array indexing, car/cdr, some kind of polymorphic list append, identity tests, field lookup, field access, string concatenation, substring testing, string element access, pointer arithmetic, pointer dereference, string length, array length, set arithmetic (union, intersection, subtraction, elementwise construction, membership testing), dictionary access (membership testing, insertion, lookup, deletion), and pattern matching. As I count, that's a bit over 50 operations (see A survey of imperative programming operations' prevalence (p. 201)), but probably any given language would get most of the benefit from only the few of them that are most used in that language. (I think Ur-Scheme, for example, implements about 13 of them at all.)

The logic of this is that, even if, say, you have printf-style string formatting built into your language as a built-in operator that's also used in other contexts for a fundamental arithmetic operation, as Python does, actually executing that operation involves copying enough characters around that doing a full method dispatch to get there is only a little extra work. So it might take, say, 100 ns to format the string† and an extra 3 ns to do the full method dispatch, so avoiding the dispatch might speed up your program by 3%. But if you can cut that 3 ns down to 0.3 ns when the operation is, say, an 0.3-ns multiplication instruction, you've sped up your program by 5×, a 400% increase. So if you have to choose, it's usually better to speed up some programs by 400% than others by 3%. It means that at least your language can do *some* things fast, while the other choice is for it to do everything slowly.

This approach will give better performance if you design the language to avoid the case where multiple different types implement *the same operation* in ways that are all important to optimize in this way; the usual suspects are integer and floating-point arithmetic,

both of which can be very fast on modern machines, but which commonly use the same operators, such as + and *. OCaml instead uses +. and *. for floating-point arithmetic, but other alternatives include doing your floating-point math with numpy-like vector types which can amortize the dispatch overhead over a number of array members.

†I just did a test with snprintf in C and it took more like 400 ns per iteration to fill a 100-byte buffer! A small format string change cut it to 150 ns, producing different output.

Topics

- Performance (p. 1155) (22 notes)
- Experiment report (p. 1162) (14 notes)
- Lisp (p. 1174) (11 notes)
- Assembly-language programming (p. 1175) (11 notes)
- Compilers (p. 1178) (10 notes)
- OCaml (p. 1249) (4 notes)
- Garbage collection (p. 1255) (4 notes)
- COMFY-* (p. 1300) (3 notes)
- Allocation performance (p. 1308) (3 notes)
- Steel Bank Common Lisp (p. 1330) (2 notes)
- Ur-Scheme

Failing to stabilize the amplitude of an opamp phase-delay oscillator

Kragen Javier Sitaker, 02021-03-23 (updated 02021-03-24)
(10 minutes)

I was playing with Falstad's simulator and managed to rig up an analog oscillator with a single op-amp and a phase-delay network.

```
$ 1 0.000004999999999999996 9.384708165144016 72 5 43 5e-11
r 144 368 208 368 0 1000
c 208 368 208 432 0 0.00001 1.2650021009105155 0.001
g 208 432 208 464 0 0
368 208 368 208 304 0 0
r 208 368 288 368 0 1000
c 288 368 288 432 0 0.00001 1.5479202079950323 0.001
g 288 432 288 464 0 0
368 288 368 288 304 0 0
368 512 384 512 320 0 0
w 512 384 592 384 0
w 592 384 592 224 0
a 400 384 512 384 9 15 -15 1000000 1.3636380375651858 1.5479202079950323 100000
w 400 400 400 432 0
w 288 368 400 368 0
w 592 384 592 432 0
r 592 432 400 432 0 10000
r 400 432 400 496 0 1000
g 400 496 400 512 0 0
c 144 224 592 224 0 0.0000056 -14.316394194122497 0.001
r 144 368 144 448 0 1000
g 144 448 144 464 0 0
w 144 224 144 368 0
r 592 384 688 384 0 22000
d 688 384 752 384 2 default
c 752 384 752 480 0 4.7000000000000004e-8 14.452075111159305 0.001
g 752 480 752 512 0 0
w 752 384 816 384 0
r 816 384 816 480 0 1000000
g 816 480 816 512 0 0
368 752 384 752 320 0 0
o 3 64 0 4099 2.5 3.2 0 2 3 3
o 7 64 0 4099 2.5 3.2 1 2 7 3
o 8 64 0 4099 20 25.6 1 2 8 3
o 29 64 0 4099 20 25.6 2 2 29 3
```

The big issue here is that the waveform is not very controllable; it spontaneously starts oscillating in what is at first a nice exponentially growing sinusoid, but rapidly hits the opamp rails, at which point it continues to oscillate nicely but no longer sinusoidally. So I was thinking I could maybe rig up a peak detector and use that to control a MOSFET to control the loop gain. I was scared to use a low-impedance peak detector, since the impedances in the rest of the

circuit are kind of high, but now I realize that was silly since the peak detector is directly connected to the op-amp output... so even 100Ω and 1μF don't load the simulated circuit.

So the idea is to decrease the amplifier gain when the peak detector voltage is too high, and increase it when it's too low. An N-MOSFET in the ohmic region decreases its resistance when its gate voltage goes high, and increases it when it's low. So if I add an N-MOSFET into the negative feedback path for the op-amp, I would want to add it where the gain goes up when its resistance goes up, which is to say, between the inverting input and the output. This has the complicating factor that the source of the MOSFET won't be at ground but rather somewhere near the input, which could result in modulating the gain with the waveform.

This almost works:

```
$ 1 0.000004999999999999996 58.69854309417089 72 5 43 5e-11
r 144 368 208 368 0 1000
c 208 368 208 432 0 0.00001 0.2588422704884921 0.001
g 208 432 208 464 0 0
368 208 368 208 304 0 0
r 208 368 288 368 0 1000
c 288 368 288 432 0 0.00001 0.19461855112354048 0.001
g 288 432 288 464 0 0
368 288 368 288 304 0 0
368 512 384 512 320 0 0
w 512 384 592 384 0
w 592 384 592 224 0
a 400 384 512 384 9 15 -15 1000000 0.19460053421583465 0.19461855112354048 100000
w 400 400 400 432 0
w 288 368 400 368 0
w 592 384 592 432 0
r 496 432 400 432 0 8200
r 400 432 400 496 0 1000
g 400 496 400 512 0 0
c 144 224 592 224 0 0.0000056 -1.4716772955110715 0.001
r 144 368 144 448 0 1000
g 144 448 144 464 0 0
w 144 224 144 368 0
r 592 384 688 384 0 100
```

```
d 688 384 752 384 2 default
c 752 384 752 480 0 0.000001 4.174094771849374 0.001
g 752 480 752 512 0 0
w 752 384 816 384 0
r 816 384 816 480 0 1000000
g 816 480 816 512 0 0
368 752 384 752 320 0 0
f 544 464 544 432 32 3 0.02
w 560 432 592 432 0
w 496 432 528 432 0
c 496 528 496 432 0 0.00001 3.928413464834166 0.001
w 496 528 544 528 0
w 544 528 544 464 0
r 544 528 656 528 0 470000
w 656 528 752 384 0
r 496 528 496 608 0 270000
g 496 608 496 624 0 0
R 624 464 656 432 0 0 40 15 0 0 0.5
r 624 464 544 528 0 1000000
o 3 256 0 4099 5 6.4 0 2 3 3
o 7 256 0 4099 2.5 3.2 1 2 7 3
o 8 256 0 4355 20 25.6 1 2 8 3
o 29 256 0 4355 20 25.6 2 2 29 3
o 33 256 0 4355 5 0.0001953125 3 2 33 3
```

At startup, this version of the circuit oscillates between the rails until the peak detector charges up enough to push the AGC MOSFET into its linear region. But this leaves the gain slightly too low, and the gain *stays* too low until the oscillations die away (the Q is about 2 I think). The gain then rises so that over two or three oscillations the wave grows from 10 mV to 15 V peak amplitude and saturates again, at which point the cycle starts over.

There are different things I could try to improve this situation. When it's oscillating successfully, the channel resistance is about 50–900Ω, and 900Ω is about the point where it explodes and saturates again; I could try to make the gain respond more rapidly to changes in the channel resistance by ensmallening the fixed resistors in the feedback path. I could try to reduce the AGC control loop delay, although it seems like it would be difficult to solve the problem entirely that way. I think I'll try the ensmallening approach first; instead of 1kΩ and 8.2kΩ I'll try 100Ω and 820Ω. That does reduce the problem a bit but doesn't eliminate the problem.

The particular way that the problem manifests is that, as the waveform is growing, at some point the MOSFET is carrying so much current (about 6.5 mA) that it goes past its linear region into saturation. At this point, just as surely as in cutoff, the opamp can raise its output voltage with no effect on the voltage it sees on its inverting input — and so its voltage immediately shoots to the positive rail.

Further tweaking doesn't solve the problems:

```
$ 1 0.000004999999999999996 28.90693621245522 72 5 43 5e-11
r 144 368 208 368 0 1000
```

```
c 208 368 208 432 0 0.000001 0.011752762198423625 0.001
g 208 432 208 464 0 0
368 208 368 208 304 0 0
r 208 368 288 368 0 1000
c 288 368 288 432 0 0.000001 0.00350939848442611 0.001
g 288 432 288 464 0 0
368 288 368 288 304 0 0
368 512 384 512 320 0 0
w 512 384 592 384 0
w 592 384 592 224 0

a 400 384 512 384 9 15 -15 1000000 0.003509064780051761 0.00350939848442611 100000
w 400 400 400 432 0
w 288 368 400 368 0
w 592 384 592 432 0
r 496 432 400 432 0 819.9999999999999
r 400 432 400 496 0 100
g 400 496 400 512 0 0
c 144 224 592 224 0 5.599999999999999e-7 -0.006312701073670032 0.001
r 144 368 144 448 0 1000
g 144 448 144 464 0 0
w 144 224 144 368 0
r 592 384 688 384 0 100
d 688 384 752 384 2 default
c 752 384 752 480 0 6.8e-7 5.6158121109211905 0.001
g 752 480 752 512 0 0
368 752 384 752 320 0 0
f 544 464 544 432 32 3 0.02
w 560 432 592 432 0
w 512 432 528 432 0
c 496 528 496 432 0 0.0000067999999999999999 4.810690328360631 0.001
w 496 528 544 528 0
w 544 528 544 464 0
r 544 528 656 528 0 220000.00000000003
w 656 528 752 384 0
r 496 528 496 608 0 270000
g 496 608 496 624 0 0
R 624 464 656 432 0 0 40 15 0 0 0.5
r 624 464 544 528 0 1000000
w 512 432 496 432 0
o 3 128 0 4099 2.5 3.2 0 2 3 3
o 7 128 0 4099 1.25 1.6 1 2 7 3
o 8 128 0 4355 20 12.8 1 2 8 3
o 26 256 0 4355 20 25.6 2 2 26 3
o 30 256 0 4355 10 0.0001953125 3 2 30 3
```


Topics

- Electronics (p. 1145) (39 notes)
- Falstad's circuit simulator (p. 1198) (7 notes)
- Facepalm (p. 1199) (7 notes)

- Oscillators (p. 1283) (3 notes)

Running scripts once per frame for guaranteed GUI responsivity

Kragen Javier Sitaker, 02021-03-23 (updated 02021-10-12)
(7 minutes)

User interfaces are real-time programs. David Liddle says:

> The programming world was not designed for the kind of programming we were doing. These graphical user interfaces — that's real-time computing, okay? And so all the programming languages that were in existence had been spawned, intended to do batch-like computing. All real-time stuff that was out there — it was all written in machine language! I mean it was just hard coded, you know! Honeywell's probably still selling Sigma 7s [the final SDS computer, bought by Xerox and then sold to Honeywell during Liddle's tenure at Xerox], because there were a bunch of countries in Latin America that all got together and wrote flight simulators for the Sigma 7, okay? And nobody wanted to pay to move that stuff, and it was all machine language, as *all* real-time [code] was back then!
>
> Well, we were doing real-time computing! We were tracking the mouse and moving windows and all that kind of stuff. You know how frustrating it is and there's even a *short* delay from a *manual* input. So we had no choice. … After 80 milliseconds, it's cold pizza, man. You know? I mean, you can tolerate a *veerrry* tiny lag, okay? But, you know, the human being wants to operate at, you know, 12½ hertz or go home.

Suppose you want to run some arbitrary scripts in an interactive display system, like a game, using a flexible programming language like Lisp, but you want to ensure that those scripts don't cause it to use more memory or become unresponsive. One possible way to handle this is to run the scripts once per frame, allocating only from a per-frame arena heap which gets nuked before the next frame, similar to the nursery of a generational garbage collector. The difference is that any permanent effects need to go through some kind of "interprocess communication" eye of the needle which will not pass references into the per-frame heap — so you can pass, say, byte strings, or maybe JSON-serializable objects, but not, say, mutable data structures. There is no automatic copying out of retained objects when the nursery is full.

An advantage of doing things this way is that, to the extent that you can contain any side effects from script execution inside a "transaction" of some kind, you have the option of fearlessly aborting a script at any point, either because it's run out of time, because it ran out of *memory*, or because it detected an error. Changes within the per-frame heap are entirely exempted from this because they will get vaporized when the script ends, whether succeeding or failing.

If you run the pending event handlers in a frame starting from the highest-priority ones, you can ensure that if anything fails to run because you ran out of time, it's the lowest-priority scripts. Often, in interactive systems, the handling of an event can be separated into several parts which can to some extent fail independently:

- **Bottom-half handlers**. Some kind of minimal state update that provides feedback that the event has happened; for example, a bullet hitting a player might make the screen go red, or a keystroke might put a letter on the screen.

- **Background tasks.** Some kind of cascading state changes that happen as a result; for example, a keystroke might update an editor buffer, which might cause a lot of text to get re-syntax-highlighted, or a player entering a new region might h spawn a bunch of mobs, which then begin pathfinding to attack the player.
- **Isochronous tasks.** Some kind of logic to generate a screen image from the current internal state.

Normally you would like #3 to run *after* #1 and #2, so that it takes into account the latest events, but generally #3 has a hard deadline to generate some pixels, and if it doesn't complete in time, the program will miss a frame, which is not ok. So if something has to get aborted it should be a background task, #2. But there isn't a clearly obvious time at which that should happen; if the frame is due at 8.3 milliseconds, say, and the current time is 5.2 milliseconds, should we abort scripts from #2 or not? And the answer depends on whether #3 needs more or less than 3.1 milliseconds to reliably run to completion.

One plausible way to handle this is to analyze the isochronous code in #3 for its worst-case execution time (WCET) and use that to compute the deadline, so the deadline will always be hit. A different alternative is to measure how long it takes, and how variable that timing usually is, and use something like the mean of the last 30 execution times, plus, say, three standard deviations. This will sometimes miss frames, but perhaps rarely enough to be acceptable for interactive applications.

Computing WCET is harder when there are interrupts, because interrupts can happen during the isochronous code. You need some kind of bound on how frequently they can happen and how long the (top-half) handler can take to run.

Any time left over at the end of the frame can be used to run (bottom-half) handlers and background tasks.

In general it is totally okay for the script's ephemeral heap to contain pointers to things outside of it, just not the other way around. But, if writing within the ephemeral heap is supported, the system needs to be able to distinguish these potentially-external pointers from heap-internal pointers.

When a new event comes in, it would be undesirable for any currently-running background script to yield the CPU to the bottom-half handler. This can be achieved by aborting it and resetting the allocation pointer or by running the handler on a different heap. If you have multiple cores you presumably want to try to run background tasks on all of them, so you'll probably have to suspend or abort one of them.

Pointer-bumping allocators can be very quick.

Topics

- Human-computer interaction (p. 1156) (22 notes)
- Safe programming languages (p. 1172) (11 notes)
- Real time (p. 1195) (7 notes)
- GUIs (p. 1216) (6 notes)

- Transactions (p. 1239) (4 notes)
- Allocation performance (p. 1308) (3 notes)
- Latency (p. 1358) (2 notes)
- Interrupts (p. 1361) (2 notes)

Minor improvements to pattern matching

Kragen Javier Sitaker, 02021-03-24 (updated 02021-04-08)
(10 minutes)

Reading EOPL I enountered their `variant-case` structure. The idea is that if you have, say, a tree made out of `interior` records and `leaf` records defined as (p. 80, §3.4.1)

```
(define-record interior (symbol left-tree right-tree))
(define-record leaf (number))
```

then you can define, say (p. 81, §3.4.2, slightly tweaked):

```
(define (leaf-sum tree)
  (variant-case tree
     (leaf (number) number)
     (interior (left-tree right-tree)
       (+ (leaf-sum left-tree) (leaf-sum right-tree)))
     (else (error "leaf-sum: Invalid tree" tree))))
```

Polymorphic variants in OCaml

This is pretty closely analogous to polymorphic variants in OCaml, except that the fields are named; in the last case, the `symbol` field is unused and so not mentioned. In OCaml we can define this without defining the record types first, but the fields are named only positionally:

```
# let rec leaf_sum = function `Leaf n -> n | `Interior (_, left, right) -> leaf_so
oum left + leaf_sum right ;;
val leaf_sum : ([< `Interior of 'b * 'a * 'a | `Leaf of int ] as 'a) -> int =
  <fun>
# leaf_sum (`Interior (`Leaf 4, `Interior (`Leaf 5, `Leaf 6)));;
Characters 9-60:
  leaf_sum (`Interior (`Leaf 4, `Interior (`Leaf 5, `Leaf 6)));;
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Error: This expression has type
         [> `Interior of
              [> `Leaf of int ] *
              [> `Interior of [> `Leaf of int ] * [> `Leaf of int ] ] ]
       but an expression was expected of type
         [< `Interior of 'b * 'a * 'a | `Leaf of int ] as 'a
       Types for tag `Interior are incompatible
# leaf_sum (`Interior ("foo", `Leaf 4, `Interior ("bar", `Leaf 5, `Leaf 6)));;
- : int = 15
```

The type inferred is not actually fully general, because it requires the type for a given tag to be consistent:

```
# leaf_sum (`Interior ("foo", `Leaf 4, `Interior (3.14, `Leaf 5, `Leaf 6)));;
Characters 9-73:
  leaf_sum (`Interior ("foo", `Leaf 4, `Interior (3.14, `Leaf 5, `Leaf 6)));;
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Error: This expression has type
         [> `Interior of
              string *
              ([< `Interior of string * 'a * 'a | `Leaf of int > `Leaf ]
               as 'a) *
              [> `Interior of float * [> `Leaf of int ] * [> `Leaf of int ] ] ]
       but an expression was expected of type 'a
       Types for tag `Interior are incompatible
```

Named fields for terser code

Usually you have more than one function operating on a given type, so it occurred to me that the Scheme code is somewhat redundant; as long as it's only dispatching on record types, it could be written as follows:

```
(define (leaf-sum tree)
  (variant-case tree
     (leaf number)
     (interior (+ (leaf-sum left-tree) (leaf-sum right-tree)))
     (else (error "leaf-sum: Invalid tree" tree))))
```

Moreover the `error` could be implicit as it is in OCaml.

This way of doing things requires you to name your fields in a type declaration, and only accommodates the simplest pattern-matches, but those are nevertheless the most commonly used ones. (It also has the disadvantage that adding fields to a record type could silently change the meaning of existing code, instead of just breaking it as it normally does.) So you could imagine saying, for example:

```
a tree:
    a leaf:
        n: int
    a interior:
        sym: symbol
        left-tree: tree
        right-tree: tree

to leaf-sum:
    on leaf:
        n
    on interior:
        leaf-sum left-tree + leaf-sum right-tree
```

Pointer-bit variant discrimination

Another vaguely related pattern-matching note is that if your record types are all non-polymorphic sum types like the above, and you do full type erasure, as is normal in ML, then in most cases you

can get away with discriminating them entirely with pointer tag bits, avoiding embedding a tag field in the record itself. `tree` above, for example, needs only one tag bit, to distinguish `leaf` from `interior`; very many such sum types need only 2–4. You could provide an "overflow tag", say, when all the pointer-tag bits are 1, which indicates that the record does indeed contain a tag field further discriminates the record type, but only types with 8 or more variants will need it if your pointers are 64-bit aligned.

Here are the last few sum types I defined. These are from porting μKanren to OCaml:

```
type var = Var of Index.t (* the index is a counter typically from call_fresh *)
type term = Vart of var | Const of int | Pair of term * term
type 'a stream = Cons of 'a * 'a stream | Thunk of (unit -> 'a stream) | Mzero
type state = State of env * Index.t (* index of the next variable to create *)
```

This is from an incomplete port of COMFY-65 to OCaml; the real type would have about five more variants:

```
type ast = If of ast * ast * ast | Not of ast | Seq of ast list | Const of int
```

This is also sort of an example:

```
type num = Int of int | Float of float
type expr = Sum of expr * expr | Product of expr * expr | Const of num
```

This was also sort of an example:

```
type test_item = Hematocrit of int | Creatinine of float | Glucose of int
type test_items = EmptyTest | TestCons of test_item * test_items
type test = Test of (int * float * test_items)
type int_tag = HematocritT | GlucoseT
type float_tag = CreatinineT
type by_type_tag = EmptyBTT
                 | BTTConsInt of (int_tag * int * by_type_tag)
                 | BTTConsFloat of (float_tag * float * by_type_tag)
type int_item = HematocritI | GlucoseI
type float_item = CreatinineI
type item = IntItem of int_item * int | FloatItem of float_item * float
type maps_test = MTest of (int IntMap.t * float FloatMap.t)
  type item = Int of K.int_key * int
            | Float of K.float_key * float
  type int_key = Hematocrit | Glucose
  type float_key = Creatinine
```

This was from Neel Krishnaswami:

```
type 'a exp =
  | Var of string
  | App of 'a exp * 'a exp
  | Lam of string * 'a
```

This is a regular expression engine, based on a remark by Dave Long, which I cut down to use polymorphic variants in order to

minimize the amount of code:

```
let rec any = function `N -> false | `C (h, t) -> h || any (t ())
and map f = function `N -> `N | `C (a, b) -> `C (f a, fun () -> map f (b ()))
and iota m n = if m = n then `N else `C (m, fun () -> iota (m+1) n)
let rec splits s = let n = String.length s in
               map (fun i -> String.sub s 0 i, String.sub s i (n-i))
                 (iota 0 (n+1))
and matches s = function `Lit t -> s = t
                      | `Cat (h, t) -> any (map (fun (a, b) ->
                        matches a h && matches b t) (splits s))
                      | `Alt (a, b) -> matches s a || matches s b
                      | `Star r -> s = "" || matches s (`Cat (r, `Star r))
```

This uses two types, which could be defined in the conventional way as

```
type stream = Cons of bool * (unit -> stream) | Nil

type regex = Alt of regex * regex | Cat of regex * regex | Star of regex | Lit of string
```

And here are some types from Bicicleta:

```
type methods = NoDefs
                         (* name, body, is_positional ... *)
               | Definition of string * bicexpr * bool * methods
and bicexpr = Name of string
              | Call of bicexpr * string
              | Literal of string option * methods
              | Derivation of bicexpr * string option * methods
              | StringConstant of string
              | Integer of int
              | Float of float
              | NativeMethod of (lookup -> bicobj)
and userdata = UserString of string
              | UserInteger of int
              | UserFloat of float
                (* name, selfname, body, env *)
and bicmethod = string * string option * bicexpr * lookup
and bicobj = ProtoObject
             | BaseObject of lookup
            (* Derive of positional method names, methods, parent, cache *)
             | Derive of string list * bicmethod list * bicobj
                 * (string, bicobj) Hashtbl.t option ref
             | UserData of userdata
             | Error of string * string
and lookup = Phi | Add of string * bicobj * lookup ;;
```

So, in reverse order, these types have 2, 5, 1, 3, 8, 2, 4, 2, 3, 1, 2, 2, 1, 2, 1, 2, 3, 1, 2, 1, 2, 3, 3, 2, 4, 1, 3, 3, and 1 variant. So in most cases you could distinguish them entirely with pointer bits, even if you only had two pointer bits to play with.

A more aggressive way to handle this is to represent references of a

given type as integers, some of whose bits indicate which variant the object belongs to, while the other bits index an array of all objects of that variant. For example, the high 16 bits of a 32-bit oop might indicate whether an object is a ProtoObject, a BaseObject, a Derive, a UserData, or an Error, while the low 16 bits index an array of Derives or BaseObjects or whatever. For really simple generational garbage collection you could allocate a second set of typecodes for nursery ProtoObjects, nursery BaseObjects, and so on, where the "index" bits directly indicate an offset into the nursery (probably bit-shifted by whatever your nursery allocator alignment is.)

The regexp engine revisited

The regex engine above in OCaml with polymorphic variants consists of 547 non-indentation characters. We could rewrite it in the above notation:

```
a stream:
    a cons:
        car: bool
        cdr: unit -> stream
    a nil

a regex:
    a literal:
        content: string
    a catenation:
        head: regex
        tail: regex
    a alternation:
        a: regex
        b: regex
    a closure:
        content: regex

to any:
    on nil:
        false
    on cons:
        head or any (tail())

to map f:
    on nil:
        nil
    on cons:
        cons (f head) λ().map f (tail())

to iota m n:
    nil if m == n else cons m λ().iota (m+1) n

to splits s:
    n ← #s
    map (λi.s[0:i], s[i:n-i]) (iota 0 (n+1))

to matches s:
```

```
    on literal:
        s == content
    on catenation:
        any (map (λa b.matches a head and matches b tail) (splits s)
    on alternation:
        matches s a or matches s b
    on closure:
        s == "" or matches s (cat content (star content))
```

That's 646 non-indentation characters, 15% larger. You could imagine that if you had more than one function on regexps, you could start winning:

```
to can_be_empty:
    on literal:
        s == ""
    on catenation:
        can_be_empty head and can_be_empty tail
    on alternation:
        can_be_empty a or can_be_empty b
    on closure:
        true
```

Topics

- Programming (p. 1141) (49 notes)
- Performance (p. 1155) (22 notes)
- Lisp (p. 1174) (11 notes)
- Programming languages (p. 1192) (8 notes)
- Syntax (p. 1221) (5 notes)
- OCaml (p. 1249) (4 notes)
- Scheme (p. 1274) (3 notes)
- Pattern matching

Why Bitcoin is puzzling to people in rich countries

Kragen Javier Sitaker, 02021-03-31 (updated 02021-07-27)
(10 minutes)

Originally posted at https://news.ycombinator.com/item?id=26238410. Slightly edited at https://news.ycombinator.com/item?id=26654767, https://news.ycombinator.com/item?id=27337189, and https://news.ycombinator.com/item?id=27448744.

I'm not in El Salvador, but I do have some experience with how Bitcoin gets used in practice in low- and middle-income countries, despite the transaction fees sometimes being high. I *don't* have experience with Strike or Lightning in general, so while in theory they should help a lot with the transaction-fee issue, I don't know how they work out in practice.

I've been using Bitcoin to get paid for a couple of years at this point where I live here in Argentina. It's currently 13 years after Bitcoin's invention, and some people think it's regressing instead of progressing. Well, 13 years after the internet's invention was 01982; not only couldn't you get so much as a weather report online, much less IRC, but many of the early interesting experiments like NLS at SRI had shut down, and more and more places were disabling guest access to their hosts—you couldn't run so much as a game of ADVENT without getting a username. And a password. Things were seriously regressing. The only people you could talk to on the internet were other people who really bought into the subculture.

If you live in a country with a highly functional banking system and no kleptocracy, Bitcoin is probably a bit puzzling unless you have family in Cuba. But it's not puzzling at all for those of us who live somewhere in the middle of the broad spectrum between Switzerland and Somalia, because most places have a *little* kleptocracy. Argentina is a stable democracy, far from being "a failed state,"† but if you want to send US$500 abroad via non-Bitcoin means it's basically impossible, and the only broadly available savings vehicle is real estate ("*ahorrar en ladrillos*"), which of course grossly inflates real-estate prices, with a substantial part of the capital city occupied by empty apartments someone bought "as an investment". Historically, Argentines have saved by buying dollars, but that's limited to US$200 a month now, and then only if you have a non-under-the-table job (about a third of total employment is under the table):

https://www.ambito.com/finanzas/dolares/cronologia-del-cepo-cambiario-se-cumple-un-ano-la-restriccion-impuesta-macri-n5129832

You can see that in September 02019 when this measure was imposed the price of a dollar was AR$63.50; now it's AR$155. So whatever savings you had in pesos in 02019 have lost 59% of their value to peso devaluation.

In 02001 a lot of Argentines had saved dollars in their

dollar-denominated bank accounts. This did not preserve their savings through the financial crisis that year; the cash-strapped government limited withdrawals to a trickle, then converted dollar deposits to pesos at a one-to-one rate, then released the exchange-rate peg, at which point peso went overnight from being worth US$1 to being worth US$0.25 before settling at about US$0.31 for the next few years. The US did something similar in 01933.

Some might suggest using "alternatives to banks like credit unions where customers—as owners—hold more power," but Credicoop depositors suffered the same two-thirds confiscation of savings as depositors in for-profit banks. And they pay the same 3% tax on bank transactions including checks. That's more than a fast Bitcoin transaction fee of US$15 for transactions over US$500.

But we're not a failed state. There are no gangs of bandits roving the streets in Argentine cities (though there are some pretty bad slums where you'll get robbed if you wander in without knowing anybody). Courts, free public hospitals, and roads continue to function, though there are more potholes than a year ago. Argentine infant mortality is 10 per 1000 live births, down from almost 20 in the late 01990s and the same as the late 01980s in the US; life expectancy at birth is 77 years, worse than Switzerland's 84, but the same as China and Hungary, and better than Saudi or Mexico. (Somalia is 54.)

Most of the world, and notably El Salvador, is worse off than Argentina, although not necessarily in such a statistically transparent fashion. About one fourth of the people in the world are unbanked, 51% here in Argentina, 70% in El Salvador; even advanced countries like Russia, Hungary, and Uruguay have roughly a quarter of the population unbanked:

https://www.gfmag.com/global-data/economic-data/worlds-most-◦
◦unbanked-countries

And if your family lives in a country like Iran or Venezuela subject to US sanctions, and you live in the US? Good luck sending them an ACH, instant or otherwise!‡ It's well known that Bitcoin is very popular in Venezuela, which kind of is a failed state, so one of the Venezuelan governments is trying to tax Bitcoin remittances at 15%.

https://archive.fo/ZRXzS

Bitcoin handles a few billion dollars per year in such remittances, which are the lifeblood of the Salvadoran economy. A few billion dollars a year might seem like a trivial amount of money to someone in a rich country, but in poor countries, it's enough to keep several million people alive.

Even in the US, it's common for the police to confiscate large amounts of paper currency just because they can ("civil forfeiture"); US bank accounts are probably fine for US$100K but probably somewhat risky for US$10M if the bank thinks you don't seem like the kind of person who ought to have it. US$10M in US$100 bills fits in a box you can wheel around on a dolly, but Bitcoin is a lot more practical. (And of course US$10M in dollar bills loses about US$200k per year to inflation.) The problems with official corruption in El Salvador are reputed to be dramatically worse than in the US, and Bitcoin should help a lot with that.

Transaction fees are usually high enough that you wouldn't want to use Bitcoin to pay for a can of Red Bull or even a restaurant dinner. But it's extremely practical as an alternative to Western Union or US$100 bills or gold, even with the current very high transaction fees. At the moment, the Bitcoin transaction fee is very low—the median Bitcoin transaction fee in the last block was 0.0495 millibitcoins, which is US$1.72:

https://btc.com/000000000000000000c28aea6e8c073e44e249460e80e16cfc4a46f3b47d536b?page=60&order_by=fee&asc=1

When I last checked a week ago, it was 0.00678 millibitcoins, which is US$0.25:

https://btc.com/0000000000000000000778ef382c1697706e34634696ece8d3243eb061e896d9?page=59&order_by=fee&asc=1

Three months ago it was at what I think of as a more normal rate of 0.31 millibitcoins, US$11, which is lower than the 3.4% spread you'd pay to a jeweler or black-market money changer for transactions over US$350:

https://btc.com/00000000000000000000476ab57eea9be8ada36e26803130287eb75c7e99797d?page=72&order_by=fee&asc=1

So, Bitcoin doesn't have to be a cypherpunk utopia to be a big improvement on the *status quo ante*. For those of you living in stable countries where your worries are things like "instant and extremely low-fee ACHs" and "decentralized utopia", this may be very confusing, but try to remember that most of the world lives in places with much more pressing concerns, concerns that Bitcoin helps a lot with. And you may live there too, soon—the loyal subjects of Kaiser Wilhelm in 01913 certainly didn't expect that in 15 years they'd be in the middle of a hyperinflation episode that remains legendary a century later.

I think that, by providing workarounds to the people who need them, cryptocurrencies probably not only ameliorate the most immediate and pressing concerns of poor parts of the population like Venezuelan immigrants and MS-13 victims, but probably also adjust the power balance in a more liberal and democratic direction. This will improve the chance of those concerns being ameliorated by public policy over the next decades as well. But it's hard to tell what will really happen. The potential disaster scenario is that, by making most taxation impossible, cryptocurrencies destroy the modern welfare state without providing anything to replace it. So the public hospitals close, the enormous police force starts to support itself by extracting tribute, and the infrastructure decays. Pretty similar to what's happened in the US over the last 50 years, in fact, only more so.

However, at this point I think the modern welfare state is already doing a good enough job of destroying itself without any significant help from cryptocurrencies—as evidence, I can point to Maduro, Macri, Bolsonaro, Trump, and Brexit, and metonymically to the social changes they betoken. So at this point I'm more worried about cushioning the collapse than preventing it.

† We've remained democratic since 01983, electing presidents from three different political parties (UCR, PJ, and PRO), and there's no serious insurgency. It's the *economy* and *government policy* that are ruinously unstable, to a point that seems satirical to anyone accustomed to the US, but is lamentably common worldwide. Rich people sometimes say they don't know of legitimate uses of Bitcoin outside of "failed states".

‡ Family remittances are specifically exempted from the US sanctions on Iran, but good luck finding a US bank that's willing and able to take that risk:
https://www.wiggin.com/wp-content/uploads/2019/09/26580_advisory-family-remittances-from-us-to-iran-not-prohibited-by-iranian-transactions-regulations-martini-glasser-november-2011.pdf

Topics

- History (p. 1153) (24 notes)
- Argentina (p. 1200) (7 notes)
- Incentives (p. 1230) (5 notes)
- Economics (p. 1258) (4 notes)
- Politics (p. 1279) (3 notes)
- Bitcoin

Statistics on the present and future of energy in the People's Republic of China

Kragen Javier Sitaker, 02021-04-01 (updated 02021-04-08)
(10 minutes)

I was surprised to learn that China built more wind-powered electrical generating capacity last year than coal, and also more solar than coal. I posted about this on the orange website.

Specifically, in 02020, the People's Republic of China installed 71.7 GW of new wind capacity, 48.2 GW of new solar capacity (which was already larger than the rest of the world combined), and 38.4 GW(e) of coal capacity. Assuming typical capacity factors of 40% for wind, 25% for solar, and 60% for coal, that would add up to 23 GW average new coal, 29 GW average new wind, and 12 GW average new solar. (But China's capacity factors are lower; see below.) New solar installations worldwide double on average every three years, which has slowed down from every two years in the 02010s.

Solar capacity factors vary widely by region. In California they're 28.1%, but in Germany and the Netherlands only 10%.

For scale, total German energy use was about 3800 TWh/year over 02007–02013, including things like transport fuels. This works out to about 430 GW. Of this, 576 TWh/year (65.7 GW) was produced as electrical energy, which had reduced to (+ 60.94 81.94 35.56 59.08 131.69 50.7 45.45 18.27) = 484 TWh/year (55 GW) by the year 02020.

But China is a larger country than Germany. Chinese marketed energy consumption was 28 PWh/year (3.2 TW) in 02010, of which 3.9 PWh/year (440 GW) was electric. In 02019 they produced 7330 TWh electric calculated as (+ 4554 233 148 349 1270 32 405 224 113) rounded to three places. That's 836 GW. (The 32 TWh of pumped-storage hydro may be double-counted.) In 02019 224 TWh/year (26 GW) was produced from solar and 405 TWh/year (46 GW) from wind, using 204 GW of solar capacity (capacity factor 13%) and 209 GW of wind capacity (capacity factor 22%). Also the 4554 TWh/year from coal (519.5 GW) is on a 1.041 TW basis, so their capacity factor is only 50.0%. Hopefully they'll start installing their energy plants in more propitious places, like the Gobi, and the capacity factor will go up.

So probably last year's new installations of 38.4 GW (coal), 71.7 GW (wind), and 48.2 GW (solar) will produce on average 19.2 GW (coal), 16 GW (wind), and 6.3 GW (solar). The resulting 22 GW (average) of renewable energy added last year amounts to 2.6% of the total current *electric* energy use of China. If we assume that China's *total* energy use has increased by 90% since 02010, just as their *electrical* energy use did by 02019, it would now be 6.1 TW, and 22 GW is 0.36% of it.

Even though wind turbines have a lower cost per kilowatt and higher capacity factors, I think solar is the more interesting thing here, because it lasts for many decades and taps a much larger resource, so I'm going to focus on solar.

The relation between new installations and existing installations gives us an estimate of the growth rate of solar capacity *in China*: it's increasing by 48/204 = 23.5% per year, giving a 3.3-year doubling time, similar to the way new solar capacity *in the world* has doubled every three years over the last couple of doublings; we can expect this to remain roughly exponential for a while. We can estimate the current installed capacity as 204 + 48 = 252 GW, or 0.252 TW. We can also perhaps estimate that China's total and electrical energy usage each continue to grow at the same exponential rate they have been; 7.4% per year gives us the 90% increase we seem to be observing from 02010 to 02019. We can write this model down as follows:

```
installed = 0.252
cf = 0.13                     # capacity factor
electric_usage = 0.836
total_usage = 6.1
fmt = '| %5s | %7s | %7s | %8s | %8s |'
print(fmt % ('', 'solar', 'solar', 'electric', 'total'))
print(fmt % ('year', 'TWp', 'TW', 'TW', 'TW'))
for i in range(40):
    print(fmt % ('%05d' % (i + 2021),
                 '%.3f' % (installed * 1.235 ** i),
                 '%.3f' % (installed * 1.235 ** i * cf),
                 '%.3f' % (electric_usage * 1.074 ** i),
                 '%.3f' % (total_usage * 1.074 ** i),
                 ))
```

With this model, China's solar energy production exceeds its 02021 current electrical energy consumption of 836 GW in 02037, but doesn't exceed its contemporary electrical energy consumption until 02045 (at which point we extrapolate that it will use 4.6 TWe) and finally catches up to its total energy consumption in 02059 at 92 TW.

	solar	solar	electric	total
year	TWp	TW	TW	TW
02021	0.252	0.033	0.836	6.100
02022	0.311	0.040	0.898	6.551
02023	0.384	0.050	0.964	7.036
02024	0.475	0.062	1.036	7.557
02025	0.586	0.076	1.112	8.116
02026	0.724	0.094	1.195	8.717
02027	0.894	0.116	1.283	9.362
02028	1.104	0.144	1.378	10.054
02029	1.364	0.177	1.480	10.799
02030	1.684	0.219	1.589	11.598
02031	2.080	0.270	1.707	12.456
02032	2.569	0.334	1.833	13.378
02033	3.173	0.412	1.969	14.368
02034	3.918	0.509	2.115	15.431
02035	4.839	0.629	2.271	16.573

```
| 02036 |    5.976 |    0.777 |    2.439 |   17.799 |
| 02037 |    7.380 |    0.959 |    2.620 |   19.116 |
| 02038 |    9.115 |    1.185 |    2.814 |   20.531 |
| 02039 |   11.257 |    1.463 |    3.022 |   22.050 |
| 02040 |   13.902 |    1.807 |    3.246 |   23.682 |
| 02041 |   17.169 |    2.232 |    3.486 |   25.434 |
| 02042 |   21.203 |    2.756 |    3.744 |   27.316 |
| 02043 |   26.186 |    3.404 |    4.021 |   29.338 |
| 02044 |   32.340 |    4.204 |    4.318 |   31.509 |
| 02045 |   39.940 |    5.192 |    4.638 |   33.840 |
| 02046 |   49.326 |    6.412 |    4.981 |   36.344 |
| 02047 |   60.917 |    7.919 |    5.350 |   39.034 |
| 02048 |   75.233 |    9.780 |    5.745 |   41.922 |
| 02049 |   92.913 |   12.079 |    6.171 |   45.025 |
| 02050 |  114.747 |   14.917 |    6.627 |   48.356 |
| 02051 |  141.713 |   18.423 |    7.118 |   51.935 |
| 02052 |  175.015 |   22.752 |    7.644 |   55.778 |
| 02053 |  216.144 |   28.099 |    8.210 |   59.906 |
| 02054 |  266.938 |   34.702 |    8.818 |   64.339 |
| 02055 |  329.668 |   42.857 |    9.470 |   69.100 |
| 02056 |  407.140 |   52.928 |   10.171 |   74.213 |
| 02057 |  502.818 |   65.366 |   10.923 |   79.705 |
| 02058 |  620.981 |   80.727 |   11.732 |   85.603 |
| 02059 |  766.911 |   99.698 |   12.600 |   91.937 |
| 02060 |  947.135 |  123.128 |   13.532 |   98.741 |
```

Extrapolating an exponential trend over 40 years is very likely to be wrong, particularly when the trend has only been in effect for a few years. If we look back to 01993, 28 years ago, we see a significantly faster exponential trend in *worldwide* photovoltaic installations: 508 GW (peak, DC, nameplate capacity) in 02018, grown from maybe 130 MW in 01993, which works out to 39% growth per year, arithmetic mean, which is a 2.1-year doubling rate. [The estimate for 02019] was 627 TW, though, which is only 23.4% higher than the 02018 estimate, in line with China's growth rate.

If China's photovoltaic installations were to suddenly start growing at this faster rate, the model looks like this instead. They're generating all their electrical energy from solar by 02034 instead of 02045 and all their energy from solar by 02042. Their total production in 02050 is 480 TW, 80 times more than their current energy consumption, 25 times more than the current *world* marketed energy consumption of some 18 TW, and an order of magnitude larger than their projected energy consumption at that time.

```
installed = 0.252
cf = 0.13                   # capacity factor
electric_usage = 0.836
total_usage = 6.1
ygps = 39.2                 # yearly growth percent, solar
ygpt = 7.4                  # yearly growth percent, total
fmt = '| %5s | %10s | %10s | %8s | %8s |'
print(fmt % ('', 'solar', 'solar', 'electric', 'total'))
print(fmt % ('year', 'TWp', 'TW', 'TW', 'TW'))
for i in range(40):
```

```
print(fmt % ('%05d' % (i + 2021),
             '%.3f' % (installed * (1 + ygps/100) ** i),
             '%.3f' % (installed * (1 + ygps/100) ** i * cf),
             '%.3f' % (electric_usage * (1 + ygpt/100) ** i),
             '%.3f' % (total_usage * (1 + ygpt/100) ** i),
             ))
```

	solar	solar	electric	total
year	TWp	TW	TW	TW
02021	0.252	0.033	0.836	6.100
02022	0.351	0.046	0.898	6.551
02023	0.488	0.063	0.964	7.036
02024	0.680	0.088	1.036	7.557
02025	0.946	0.123	1.112	8.116
02026	1.317	0.171	1.195	8.717
02027	1.833	0.238	1.283	9.362
02028	2.552	0.332	1.378	10.054
02029	3.552	0.462	1.480	10.799
02030	4.945	0.643	1.589	11.598
02031	6.883	0.895	1.707	12.456
02032	9.581	1.246	1.833	13.378
02033	13.337	1.734	1.969	14.368
02034	18.566	2.414	2.115	15.431
02035	25.843	3.360	2.271	16.573
02036	35.974	4.677	2.439	17.799
02037	50.076	6.510	2.620	19.116
02038	69.706	9.062	2.814	20.531
02039	97.030	12.614	3.022	22.050
02040	135.066	17.559	3.246	23.682
02041	188.012	24.442	3.486	25.434
02042	261.713	34.023	3.744	27.316
02043	364.304	47.359	4.021	29.338
02044	507.111	65.924	4.318	31.509
02045	705.898	91.767	4.638	33.840
02046	982.611	127.739	4.981	36.344
02047	1367.794	177.813	5.350	39.034
02048	1903.969	247.516	5.745	41.922
02049	2650.325	344.542	6.171	45.025
02050	3689.252	479.603	6.627	48.356
02051	5135.439	667.607	7.118	51.935
02052	7148.531	929.309	7.644	55.778
02053	9950.756	1293.598	8.210	59.906
02054	13851.452	1800.689	8.818	64.339
02055	19281.221	2506.559	9.470	69.100
02056	26839.460	3489.130	10.171	74.213
02057	37360.528	4856.869	10.923	79.705
02058	52005.856	6760.761	11.732	85.603
02059	72392.151	9410.980	12.600	91.937
02060	100769.874	13100.084	13.532	98.741

Of course no exponential curve representing a real-world phenomenon can go on forever. China is only 9.6 million square kilometers; conservatively assuming a 35% "capacity factor" for its sunlight, thus 350 W/m^2, China only receives 3.4 petawatts of sunlight, which this projection would have it crossing in 02056.

Covering all of China's territory with 21%-efficient solar panels would produce only 700 TW, which this projection would have it crossing in 02052. Continuing to increase energy production past this level would require putting the solar panels somewhere that isn't currently China, such as on the ocean, on the Moon, or in orbit around the Earth or the Sun.

Topics

- Python (p. 1166) (12 notes)
- Energy (p. 1170) (12 notes)
- Solar (p. 1203) (6 notes)
- The future (p. 1220) (5 notes)
- China (p. 1379) (2 notes)

Geneva wheel stopwork

Kragen Javier Sitaker, 02021-04-07 (updated 02021-04-08)
(6 minutes)

Watching the "Clickspring" series of videos on hobby clockwork, I came across the section about the "stopwork", which stops the mainspring from being wound by more than, say, five turns.

The way this is done is very simple. A pair of defective gearwheels mesh; one is held in place only by light friction applied by a spring, and has five teeth, with the rest of the gear solid out to the tip of the teeth (the addendum circle), so if it were to be meshed with another, non-defective wheel, it would only be able to rotate a fraction of a rotation before locking, as the teeth of the other wheel crash into the edge of the solid disc where no teeth have been cut.

However, it is instead meshed with a gearwheel that has been filed down to just the base circle, except for a single tooth that remains protruding. So, for most of each rotation, this single-toothed wheel doesn't contact the counter wheel at all, but when it does, it advances the counter wheel by a single tooth — unless the counter wheel has already rotated all five teeth, in which case the single tooth crashes into the solid disc much as the teeth of an ordinary gearwheel would have done.

Thus an up/down counter is provided, one which blocks further motion upon reaching the end of its count. Its memory is retained in an entirely analog fashion by friction, although the count being remembered is essentially digital.

It occurred to me that a different approach to solving this problem is to use a "Geneva drive" mechanism (aka "Maltese cross", "Geneva stop", or sometimes "Geneva wheel") which is similarly defective, with one of the slots for the drive pin being blocked, so the drive wheel can only spin three, or four, or six rotations, or whatever.

This is apparently the original use of the Geneva drive!

The Geneva drive does not depend on friction and thus is invulnerable to vibrations, and moreover is susceptible to being *chained* in a way that ordinary gear wheels are not, which requires further explanation.

The defective gear wheels in the stopwork mechanism demonstrated by Clickspring have the property that the "mechanical advantage" is fairly accurately 1 during the moment when they are engaged, since they happen to have the same pitch diameter, 0 when they are not engaged, and then ∞ when they are locked. This contrasts with the simplest straightforward stopwork mechanism in which a drive pinion spins a larger wheel which encounters a stop at some point in its rotation; if the drive pinion is to be allowed to turn 8 times, for example, we might drive a 73-tooth wheel with an 8-tooth pinion, occupying 9/73 of its rotation with a stop. But this stop needs to resist 8 times the torque applied to the drive pinion. The defective-wheel mechanism does not have this problem.

But the Geneva drive permits carrying this further: not only can we arrange for its mechanical advantage to average 1 during the

driving part of the cycle, but we can use one such wheel to drive another, which drives another. (Hmm, maybe that's not such a big difference after all; the one-tooth driver can do the same if it's driving an ordinary gearwheel, after all.)

A single Geneva wheel can be made with arbitrarily many slots, at the cost of pushing the duty cycle up toward 50% with a single drive pin. In the limit of 50% you have an intermittent-motion version of a rack and pinion, with the possibility of endstops.

In the Geneva wheel's original use as a stopwork, it was in fact a limit on *differential* rotation: it limited not the absolute rotation of the inner shaft of the mainspring or its outer barrel but their *relative* rotation.

By rotating one or more disc sectors in a plane parallel to the Maltese cross, it is possible to obstruct the entry of the drive pin into the slot. But perhaps a more interesting possibility for logic is axial displacement; in the usual construction, the drive wheel has a ward in the form of a partial circle that nestles into the Maltese cross to keep it from turning when the drive pin is not engaged, with a cutout in the circular ward around the drive pin to allow the Maltese cross to rotate when the drive pin *is* engaged. But the circular ward could be made from part of a solid round shaft, along which the Maltese cross can be slid; if the cutout does not extend along its entire length, then sliding the cross up it by the thickness of the cutout and enough to clear the pin, the cross can be exempted from being incremented or decremented by the next passage of the pin.

This potentially gives us a clocked-logic system similar to Drexler's rod logic or Merkle's buckling-spring logic, though probably less suitable for miniaturization than the latter, since it relies heavily on not only contact but sliding contact.

Topics

- Contrivances (p. 1143) (45 notes)
- Mechanical (p. 1159) (17 notes)
- Physical computation (p. 1208) (6 notes)
- Gears (p. 1365) (2 notes)

A bargain-basement Holter monitor with a BOM under US$2.50

Kragen Javier Sitaker, 02021-04-07 (updated 02021-07-27)
(33 minutes)

I think you ought to be able to take a weeks-long EKG (Elektrokardiogramm) from a person with an easy-to-fabricate electronic device that could retail under US$25. A bargain-basement Holter monitor.

Specifically, you should be able to hook up some low-noise analog amplifiers to a couple of electrodes on your skin to amplify the millivolt-level EKG signals by 60dB, digitize the resulting signals of a volt or two with any old ADC (as long as it doesn't produce too much noise) at about 1 ksps and 8–12 bits deep, and record the results in NAND Flash. You want to pot the whole thing so it doesn't get eaten by your sweat, then tape it to your chest for a week or a month. When your cellphone is nearby, you can copy over the resulting data wirelessly for offline analysis.

I think the overall BOM can probably be kept under US$2.50, auguring a total retail cost under US$25. The total energy budget is about 1 mW.

This outline is almost feasible:

Item	BOM cost (qty 100)	Average power dissipation
[CR2032 cell][25]	37¢	1 µW (internal resistance)
ATTiny1614 µC	65¢	
(sampling)		240 µW
(communicating)		320 µW
MCP6401 opamp	27¢	120 µW
S34MS01G2 Flash	104¢	35 µW
4×1µF bypass caps	6.1¢	0.03 µW
4×0.1µF bypass caps	2.2¢	(probably less)
Total	241¢	716 µW

(It's suffering from the fact that the microcontroller doesn't have enough pins to operate the Flash, it may be entirely too weak, the resistors for the opamp feedback aren't included, the opamp may need to be an inamp, and you probably need another 0.01-µF cap or something for the antialiasing filter. Maybe it would be best to build a prototype with more generously specified parts first before trying to cost-optimize it.)

BOM outline

You need a battery, some kind of communication, a

microcontroller, an ADC, an amplifier or two, maybe some Flash, maybe a voltage reference, maybe a linear regulator or two, maybe a crystal, and maybe a few passives. It might be possible to shrink this down to a battery, a mixed-signal microcontroller with an onboard oscillator, a NAND Flash chip, and a couple of bypass caps, plus a loop of wire for bit-banged NFC or BLE communications to your phone.

My thought with the Flash is that 1 ksps for a month is 2.6 billion samples, and 2.6 gigabytes of Flash uses a lot less power and costs a lot less than 2.6 gigabytes of SRAM, and costs less and uses enormously less power than 2.6 gigabytes of DRAM. And maybe you can get by with less Flash than that if you can send data to your phone more often.

I don't know much about mixed-signal microcontrollers, so I don't know which ones to use.

Skin electrodes

ENIG, a common PCB finish, deposits an 0.1-μm layer of gold (or a bit less) on top of a 6-μm layer of nickel on top of exposed PCB copper. Gold is safe for skin contact; nickel is safe for 80% of the population. You can fab a 2-layer PCB with exposed ENIG pads as skin-contact electrodes a few centimeters apart. Certain lead-free solders might be a more robust nontoxic alternative: tin, silver, zinc, indium, and (for most people) bismuth and copper are acceptable ingredients, but antimony, nickel, cobalt, cadmium, and lead are toxic, and a small number of people are allergic to copper, and a few more to bismuth. So, for example, SAC305 should be fine, being 96.5% tin, 3% silver, and 0.5% copper; the eutectic is 95.6:3.5:0.9. ASTM96TS eliminates the copper, and KappAloy9 is instead the eutectic of 91% tin, 9% zinc.

It should be possible to connect the skin-contact pads to printed traces that are covered with solder mask, and which run to plated-through vias a safe distance away from the exposed pads themselves to connect them to the other side of the board.

Hypo-allergenic metals commonly used for skin contact jewelry include gold, silver, titanium, aluminum, tantalum, niobium, rhodium, platinum, and palladium.

Power, storage, and communication

A CR2032 lithium coin cell contains about 2.2 kJ of energy (233 mAh at 2.7 V is 2.3 kJ). If we're aiming for a useful life of a month, that gives us a power budget of about 840 μW; 1 mW gives us 25 days. If we want to collect data continuously, we can't turn the amplifiers on and off, so they need to be low power enough to use only a fraction of that.

Probably the right NAND strategy is to buffer up on the order of 2048–65536 samples in the microcontroller's onboard SRAM before applying some kind of lossless data compression to them, powering up the Flash, and writing a sector or ten to the Flash.

As a ballpark on processing cost, the STM32L0 uses about 230 pJ per instruction, so if it were using the whole 1 mW power budget it

would be averaging 4.3 MIPS continuously. The popular STM32F chips use more like 1500 pJ/insn, which would be more like 0.67 MIPS. It's probably best to shoot for something like 0.07 MIPS, 105 μW. The devices also use about 1 μW in stop-with-RTC mode, which takes 5 μs to wake up, and can thus usefully do a 100-instruction quantum of work before going back to sleep; so waking up 500 times a second would work, and even 1000 times a second isn't out of the question.

Writing to NAND costs on the order of 10 nJ/byte, but can usually only be done a 2048-byte sector at a time. If we assume that our compression takes us from 12 bits per sample down to 3, and we're taking 1000 samples per second, the average NAND power would be about 4 μW.

More detailed design options

So, we have a 2.7-volt coin cell driving a low-power microcontroller with an onboard ADC, fed from a low-power differential amp connected to the person through some 100kΩ resistors or something, storing data in some NAND Flash at around 1 Hz, and occasionally talking to a phone with BLE or NFC. Maybe there's an antialiasing RC filter in between the opamp and the ADC, and each of the three chips has a couple of bypass caps. The microcontroller has additional bypassing on its Vdda to reduce the power-supply noise introduced by digital electronics.

So what chips should we use?

Hopefully no discrete voltage regulators at all are necessary with the appropriate choice of parts.

Microcontroller: STM32?

The STM32F103 seems to be the default microcontroller these days, though it's probably too expensive (US$3+). I'm looking at the short version of its 117-page datasheet, DocID 13587 revision 17.

The most common one, the STM32F103C8, runs at up to 72 MHz, has dual 1Msps 12-bit ADCs, 64 or 128 KiB of Flash, and 20 KiB of SRAM, with a single-cycle 32-bit multiplier, and runs on 2.0–3.6 V. It has an internal voltage reference rated to be between 1.16 V and 1.24 V (±3.4%) at -40°–85° with a 100-ppm/° tempco. I'm thinking that this is really quite excessive accuracy for our purposes since sticking the device in a slightly different place would probably attenuate its voltage more than that.

The power consumption part of the datasheet is extensive. At an externally-clocked 72 MHz, where it's most efficient, it draws 32.8 mA at 85° with all peripherals disabled; at 2.7 V that's 89 mW or 1200 pJ per cycle, which is pretty close to 1200 pJ per instruction. At 36 MHz, half the clock speed, it uses 19.8 mA, 53 mW, 1500 pJ per cycle, which would be a lot easier on a coin cell. You might get a 5% improvement by dropping the temperature to 25°. In Stop mode, with the regulator in Run mode, it claims to use more like 24 μA (65 uW), though that's without the oscillators. Using the internal RC oscillator instead, at 36 MHz it's 14.1 mA.

The ADCs, which are connected to the second APB, use 17.5 and 16.07 μA/MHz. The APB2 bridge uses another 3.75 μA/MHz (p. 51,

table 19). If these are clocked at 2 MHz then ADC2 would use 32 μA or 87 μW at 2.7 V and the APB would use 8 μA = 20 μW more for a total of 107 μW.

The reference manual RM0008 (Rev 20, 1134 pp.) goes into more detail. It explains (p. 215) that the ADC is successive-approximation and can't be clocked (ADCCLK) over 14 MHz, so I'm guessing it needs 14 clocks per sample. It's rated only down to 2.4 V even though the rest of the device works down to 2.0 V.

The cheapest in-stock STM32 on Digi-Key (other than the STM32G031J6M6) is the STM32L011F3P6 at US$1.40 in quantity 100. It runs at 32 MHz. Its reference manual (RM0377, DocID 025942 rev. 8) explains that it has a 12-bit successive-approximation ADC with up to 256× hardware oversampling that can run at up to 1.14 Msps at 12 bits, and it can run down to 1.65 V. Its clock can run at a lower speed than the APB clock (figure 33, p. 278) using a prescaler (1×–256×) from an independent clock source, or by dividing the APB clock by 1, 2, or 4. It supports DMA (§13.2, p. 272; §13.5.2, p. 290; §13.5.5, "Managing converted data using the DMA", p. 291).

Oh, but maybe I made an error above: in STM32 Stop mode the clocks are stopped, so I'm not sure the ADC can run. To leave the ADC running, maybe you need to use sleep or low-power sleep mode (p. 144). But maybe not; the HSI16 16MHz oscillator can still run in Stop mode (§6.3.9, p. 151) and you can run the ADC from it (§7.2, p. 166; also Table 57 in §13.3.5 on p. 279) and it looks like you can run the AHB PRESC off the HSI16 oscillator and the APB{1,2} PRESC off the AHB PRESC (Figure 17, clock tree, p. 168), so maybe you *can* use the ADC in Stop mode. Not sure about whether the DMA controller works in Stop mode.

HSI16 is factory calibrated to ±1% which may be too loosey-goosey for communications.

Looking at the 119-page datasheet (DocID 027973 rev. 5) the STM32L011x3/4 runs at up to 32MHz, uses 0.29 μA in Stop mode (0.8 μW at 2.7 V) and "down to" 76 μA/MHz in Run mode (200 μW/MHz, 200 pJ/cycle). It only has 2KiB SRAM, limiting the possibilities for pre-storage compression, and 16 KiB Flash. In sleep mode at 16 MHz, the device uses 1000 μA (2700 μW), which clearly isn't acceptable (§3.1, p. 14) but maybe if you can clock it at 4MHz or 2MHz with a clock prescaler it would be okay. The wakeup time from Stop mode is 3.5 μs, which is fine (56 clock cycles at 16MHz).

This StackOverflow question implies that running the ADC in Stop mode is impossible on an STM32L4.

Microcontroller: STM32G?

The STM32G031J6M6 mentioned above seems to be new as of 02019. It's very interesting: 64MHz, 8 KiB RAM, 32 KiB Flash, 2.5Msps 12-bit ADC, which can run up to 4.38Msps at 6-bit precision. The US$1.30 package is an 8-pin SOIC, but it comes in other packages with up to 32 pins (STM32G031K, US$2.78) or even 48 (STM32G031C, US$2.98). It seems like an evolution of the STM32F chips: slightly tighter tolerances on Vrefint (±2.5%), lower Vrefint tempco (30 ppm/°) and Vddcoeff of 250 ppm/V. Energy consumption is similar to the STM32L line: 5.2 mA at 64 MHz and

25° running from Flash, which at 2.7 V is 220 pJ per cycle and probably per instruction. Low-power run mode at 2 MHz running from SRAM is 146 μA, 197 pJ per cycle.

The STM32G031J seems like it would probably be a superior choice to the STM32F or STM32L011 chips for this device, at least if software support is adequate.

Microcontroller: CKS32F?

CKS makes a line of STM32 replacement chips such as the CKS32F051C8T6 (US$1.08 in quantity 100 from LCSC). This is a drop-in compatible replacement for the STM32F051C8T6, which would cost more than twice as much at Digi-Key except that it's out of stock. (Digi-Key does have a 36-WLCSP version, the STM32F051T8Y6TR, for US$2.43 in quantity 100.)

Section 5.3.5 of the CKS datasheet, which is entirely in Chinese except for some plots they probably copied from the ST datasheet, is power consumption; table 24 on p. 42 says that running from Flash on the HSI oscillator at 32 MHz and 25° it uses 15.5 mA, which at 2.7 V would be 1300 pJ per cycle, quite comparable from the STM32F figures.

I feel like this kind of thing would probably be about the same as using an STM32F, but neither CKS nor GD seem to have an STM32L or STM32G equivalent.

Microcontrollers: AVRs? The ATTiny1614 can maybe do 240 μW for 65¢

There are some very cheap AVRs out there that don't use a lot of energy, but for example the 30¢ ATTiny25V-15MT only has 128 bytes of RAM, and in general AVRs are too power-hungry for this application.

The ATTiny1614, which seems to be new since 02018, is 65¢ in quantity 100 and has 2KiB of SRAM and an ADC. I think you can get it to work if you don't try to run the ADC continuously, but it will suck up a quarter of the whole power budget.

According to its 598-page datasheet (DS40002204A), it runs at 20MHz (though only 5 MHz at 2.7 V or less), runs anywhere from 1.8V to 5.5V, and has dual 10-bit 115-ksps ADCs and 12 GPIOs. It belongs to the "tinyAVR 1-series". It has a couple of FPGA-style 3-LUTs on the chip with it ("configurable custom logic"), and an analog comparator, but no opamps or differential ADCs.

Active power consumption at 5MHz and 3V is given as 3.2mA (table 36-4, §36.4, p. 508), which works out to a lamentable 1900 pJ per cycle — particularly lamentable considering these are 8-bit instructions. "Standby" power consumption at 3V is given as about 0.7 μA with a 32kHz oscillator running (internal or external); "power-down" consumption, with all peripherals stopped, is given as 0.1 μA (table 36-5). These numbers are quite adequate. But "idle" power consumption would be 0.6 mA at 5 MHz and 3V, which is far too much.

However, no figures are given for "standby" with a high-speed oscillator running!

So, if we can stay in "standby" or "power-down" most of the time, this chip might be a cheaper but less efficient alternative to the STM32. The question is, can it wake up 1000 times a second to take a sample, or leave the ADC running while in "standby"?

§10.3.4.1.1 says the internal OSC20M 16/20MHz oscillator starts up in "the analog start-up time plus four oscillator cycles". There's also a 32.768 kHz internal oscillator (OSCULP32K, §10.3.4.1.2) which can be used as a clock source (CLKSEL[1:0] = 0x1, §10.5.1; see also the block diagram in Figure 10-1, §0.2.1, p. 75); I suspect that might be fast enough for the ADC.

§11 describes the sleep controller. §11.3.2.1 says, "SleepWalking is available for the ADC module," but doesn't explain further, but Table 11-1 says the ADCs *can* be enabled in idle or standby mode, but not power-down mode; the RTC, PTC ("peripheral touch controller"), TCBn ("timer/counter type B #n"), BOD, and WDT can too. Moreover the ADC/PTC interrupts can wake the CPU. Wakeup time is 6 main clock cycles, which is totally insignificant, plus possibly time to restart the clock, which could probably be avoided here, although table 36-6 in §36.5 on p. 509 says 10 μs. That's still fine though.

§30.1 says the ADC has "accumulation of up to 64 samples per conversion," which I think means oversampling (giving 13 bits of precision), and "interrupt available on conversion complete", which would wake up the CPU. I don't think there's any DMA, but with 6-clock-cycle wakeup I don't think we need it.

§30.3.2.2 (p. 431) shows the ADC clock prescaler dividing `CLK_PER` by anything from 2 to 256 to get `CLK_ADC`. So if `CLK_PER` were 5 MHz (maybe from dividing OSC20M by 4) we could run `CLK_ADC` at anywhere from 19.5 kHz to 2.5 MHz, both of which are outside the 50 kHz–1.5 MHz range demanded on that page "for maximum resolution". A normal conversion takes 13 `CLK_ADC` cycles (§30.3.2.3, p. 432) so if we want 1ksps with 64× oversampling then we want `CLK_ADC` to be about 832 kHz; probably the best compromise is setting `CLK_ADC` to `CLK_PER/8` = 625 kHz (by setting the PRESC field in CTRLC to 0x2, §30.5.3, p. 440) and using 32× oversampling for 1.502 ksps (by setting the SAMPNUM field in CTRLB to 0x5, §30.5.2, p. 439).

So then I guess every 666 μs we get an ADC interrupt, wake up another microsecond later, store the conversion result in RAM, and then go back to standby? Probably setting the ADC to "free-running mode" would be best so the clock doesn't ever get turned off — but see below for why this isn't viable.

If we were stuck with a 32.768-kHz oscillator for `CLK_PER` then `CLK_ADC` would be, at best, 16.384 kHz, which the datasheet says is too slow for full accuracy; but then we'd get 1.260 ksps without any oversampling.

Table 36-7 in §36.6 (p. 510) says the ADC itself uses 325 μA at 50ksps or 340 μA at 100ksps. That is, by itself, 0.92 mW at 2.7 V, which is probably enough to blow our 1-mW power budget. Also, OSC20M uses 125 μA. So, free-running mode is far too power-hungry.

A more exotic "compressed sensing" approach might be viable. Instead of sampling the signal every 1000 μs, sample it at a random

time within each 1000-µs interval, awoken by TCB0. You need 10 µs to come out of standby mode, 8 µs to start up OSC20M (table 36-12, §36.9, p. 512), running with `CLK_PER` at 5 MHz because we're at 2.7V, and with `CLK_ADC` configured as 5 MHz/4 = 1.25MHz, another 10.4 µs to take the sample. This works out to 28.4 µs or a 2.84% duty cycle, so the 8.4 mW when active averages out to 240 µW: suboptimal but possibly acceptable.

Actually if you do the antialiasing filter on the analog side, like with a simple passive RC filter, you can just sample every 1000 µs using that same approach. So, 240 µW it is.

There are also things like the AVR32DA28T with 4 KiB of RAM and running on 4.7 mA at 24 MHz, anywhere from 1.8V to 5.5V, with different power-down modes ranging from 650 nA to 2.3 mA. Despite the name, it's an 8-bit AVR, but it costs US$1.20 in quantity 100 anyway!

Discrete amplifiers

We don't care much about signals above about 500 Hz or below about 0.1 Hz, so the amp or amps don't need to be fast, low-offset, chopper-stabilized, or any of that good stuff. But we do need a lot of gain and pretty low power. We can bias the signal to be halfway between our rails, so we don't need rail-to-rail inputs or outputs. And we only need one differential amp if we only have two electrodes. The input impedance is the skin contact resistance; without gel this is standardly modeled as 1.5 kΩ but can easily be up to 1 MΩ or so, so our input bias current needs to be about 1 nA or less.

500 Hz bandwidth and a closed-loop gain of 1000 means we need at least half a MHz of gain-bandwidth product.

My first thought was that this needs to be high precision, so an instrumentation amp is in order.

The MCP6N11 is Digi-Key's cheapest in-stock INA. But it costs US$1.42 in quantity 1, just about blowing out our hopes for a US$2 BOM in a single shot. It's 35 MHz GBW (about two orders of magnitude more than we need), 10 pA input bias, rail-to-rail output, and uses 800µA, which would also just about blow out our power budget.

Also, though, the excellent properties of INAs are useless here. Input impedance matching? Don't need it, no RF here. Low drift? Irrelevant. Low noise? Well, our input signal isn't actually all that tiny. High CMRR? The whole circuit is floating so this doesn't matter at all.

Digi-Key's cheapest in-stock opamp in quantity 1 is the MCP6001RT-I/OT, which costs US$0.24 in quantity 1 or US$0.18 in quantity 100. It's a teensy little SOT23-5 with 1 MHz GBW, 1 pA input bias, 4.5 mV input offset voltage, drawing 100 µA of power at down to 1.8 V, so it seems like it might be adequate. The 50-page MCP6001 datasheet has lots of data, including an 86dB PSRR, 88dB open-loop gain, 6 mA output short-circuit current even at 1.8 V, 50–170 µA quiescent current, and 6.1-µV input noise voltage. Though it claims 28 nV/√Hz, which would be only 0.6 µV in the 500-Hz bandwidth of interest, the 1/f knee in figure 2-12 on p.8

("Input noise voltage density vs. frequency") is around 1 kHz, so almost all this noise is actually in range.

However, in quantity 100, Digi-Key also has in stock (at the moment) the Diodes Inc. AZ4558CMTR-G1 *dual* op-amp for only 16.7¢, or 11¢ in quantity 500. But in quantity 1, they'll charge you 40¢ for the damn thing. *It* has 5.5 MHz GBW and lower input noise of 10 nV/√Hz, but because it's bipolar, it uses *2500* μA of supply current (typical! The max is 4500!), making it far too power-hungry for this application—it works up to ±20 V supply and 60 mA output.

A TI LM358LVIDDFR dual op-amp is just as cheap (19¢ in quantity 100, 10.1¢ in quantity 1000, 41¢ in quantity 1) and uses even less power (90 μA per channel), and also has 1 MHz of GBW. It has a lower input offset voltage, which we don't care about (1 mV) and a higher 15 pA input bias current, which still isn't high enough to cause problems. It has about the same noise (5.1 μV peak-to-peak, 40 nV/√Hz) which is probably plenty low. One drawback is that it's only specced to run down to 2.7 V instead of the Microchip's 1.8 V. The LM321 single-device package of the same opamp might be a better fit—half the quiescent current—but costs slightly *more*, 20¢ in quantity 100.

There are *much* lower-power opamps than that, though. TI's OPA379 claims 2.9 μA, but isn't suitable—it only has 90 kHz GBW and costs US$1.24 in quantity 1, 86¢ in quantity 100, or 59¢ in quantity 1000. This astonishingly low power usage is the very first thing in the datasheet title: "1.8V, 2.9μA, 90kHz Rail-to-Rail I/O OPERATIONAL AMPLIFIERS".

Another possible option, though, is Microchip's MCP6401 at 27¢ (37¢ in quantity 1), which runs on 1.8–6 V and 45 μA (70 μA max), delivering 1 MHz GBW and 3.6 μV p-p input noise. Its open-loop gain at 500 Hz is supposedly still about 65 dB.

If we're looking for one of the "basic components" at JLCPCB, maybe the MCP6002T-I/SN would work — it's the only "low-power opamp" in that category. It's the same MCP6001 circuit, with its 1-MHz GBW and 100μA at 1.8–6V, but has two op-amps on the chip, so it actually sucks 200 μA. JLCPCB wants to charge you 34.9¢ for it I think, though LCSC would charge 45.1¢ in quantity 10.

NAND Flash

We need to be able to buffer up some EKG data during times when the user's phone is out of communication range. If we're recording 1000 samples per second and can manage to use 8 bits per sample, that's 1000 bytes per second; 2048 bytes of RAM only buffers up 2 seconds' worth. 24 hours' worth would be 86.4 megabytes of data. The only practical way to buffer up so much data at under a milliwatt is nonvolatile memory, of which NAND Flash is the cheapest and also takes the least power to write.

Something like the "obsolete" GigaDevice GD5F1GQ4RF9IGR might be a good start. It's 128 mebibytes of NAND Flash in an 8-LGA with an SPI/dual-SPI/quad-SPI interface that can run at 100MHz. In quantity 100, it costs a dismaying US$2.21, instantly torpedoing any hopes of the US$2 BOM cost mentioned above. Its

top voltage is 2V, so it would need an external linear regulator. Parallel-interface NAND Flash chips like the S34MS01G2 (also 128 MiB) are cheaper but require lots of I/O lines. Like 15. It uses 10 μA at idle.

2-gibibit Flash seems to start at US$2.60 in quantity 100 with things like the Micron MT29F2G08, with no datasheet available, or US$2.79 with the Kioxia TC58BVG1S3HTA00, which runs on 2.7–3.6 V, uses 30 mA in operation and 50 μA in standby, and has a serial interface. Programming a 2048+64-byte page takes 330 μs, erasing a 64-page block takes 2.5 ms, and despite its 48 pins it looks like you can run it on 15 GPIOs or less. Doing the math, erasing takes 19 ns per byte, writing takes 161 ns per byte, and the total 180 ns at 30 mA and 2.7 V takes 15 nJ per byte. This is a bit higher than the 4.4 nJ number I computed for the S34MS01G2, but not ridiculous. 1000 bytes per second at 15 nJ is 15 μW, which is adequately small. (NOR Flash takes about 100× as much power to write, which would blow our power budget.)

The Kioxia part seems to be specced for 40MHz, but probably 8MHz is a more realistic speed for driving it from a cheap microcontroller. This adds another 256 μs or so of time to write each page, which might or might not double its power usage, but either way it's fine.

Although Kioxia's pricing suggests that maybe 1-gibibit NAND is already at or above the linear pricing region, Digi-Key only stocks one 512-mebibit NAND chip, the Winbond W25N512GVEIG, which is actually *more* expensive than the gibibit chips.

30 mA across the battery's internal resistance of 15 Ω or so would produce almost half a volt of power rail droop, which could be a big problem not only for analog measurements but maybe even for digital circuit stability.

The NAND is probably also the heaviest current draw, so it might give us an idea of how much capacitance we need. If we wanted to handle 30 mA for 2.5 ms without dropping the voltage by more than, say, 0.3 V, we'd need 250 μF. Ouch! Not gonna find a cheap ceramic cap with that.

Bypass caps

Something like the 30¢ Vishay TMCMA0G227MTRF tantalum might work to keep the NAND from harrowing the battery; it's a 1206 4-V 220-μF tantalum. But if we leave it connected all the time it will leak about 30 μA, 81 μW — not out of the question, but several times as much as the NAND itself.

We probably can't get by without bypassing, and to get down below 1 mW, we need at least 10kΩ leakage resistance in all the bypass caps put together. This is a sufficiently undemanding spec that I think random ceramics will be fine; we don't need to go for 30V or 100V caps, which would be a way to cut down bypass losses further.

Sufficiently beefy bypass caps might make a coin-cell battery last longer by preventing current draw at the battery from spiking to 30 mA or more during normal use.

The 0.55¢ Samsung X5R CL05A104KA5NNNC seems like a reasonable 0.1μF bypass cap, though the datasheet doesn't specify

leakage, and, e.g., the Yageo X5R 0201 1.52¢ Yageo CC0201MRX5R5BB105 is probably adequate for a 1-μF bypass cap. Its datasheet suggests insulation resistance of 10 GΩ "or Rins × Cr ≥ 50Ω.F, whichever is less," which works out to just 1 GΩ. So it exceeds requirements by four orders of magnitude in this case.

XXX maybe 0.001 μF too? And maybe dump the 1μF one?

Bluetooth Low Energy

Wikipedia say BLE give 0.27–1.37 Mbps of "application throughput" for 0.01–0.50 W and under 15 mA. Suppose this means we can get 1 Mbps for 40 mW (15 mA at 2.7 V), which seems plausible. Well, that's 40 nJ per bit, or 320 nJ/byte. It's about an 0.8% duty cycle if we're producing 1000 bytes per second of sample data, as suggested above, but that still works out to 320 μW, which is not impossible but is a major part of our energy budget.

Transmitting a full 128-MiB Flash chip load of EKG data would take almost 20 minutes at 1 Mbps, which is annoyingly slow but not infeasible.

It may be feasible to bitbang BLE on an ATTiny24 and an nRF24L01+, at least for transmitting but this is probably not really viable without using BLE hardware.

Near-field communication

Lots of cellphones and other hand computers now support ISO/IEC 18000-3 13.56 MHz NFC, at 106–424 kbps, and also under 15 mA; because the data rate is five times slower, the cost per bit is probably about five times higher, or 1.5 mW.

Wikipedia claims NFC communication can be added for 10¢, but I don't know how to do that. It seems to be popular. One popular NFC chip is the NXP PN532, which I guess is an 8051 with NFC hardware, but it seems to cost more like 700¢; even a cheaper alternative like the NXP 512 is still almost US$4. Nobody seems to be bitbanging it successfully either. ST sells a US$1.33 8-SOIC called the M24LR64E-RMN6T/2, which is a pretty complete energy-harvesting RFID tag that also supports I²C, but it seems like that's only to read and write its memory, not to use it as a transceiver.

Topics

- Contrivances (p. 1143) (45 notes)
- Electronics (p. 1145) (39 notes)
- Pricing (p. 1147) (35 notes)
- Manufacturing (p. 1151) (29 notes)
- Energy (p. 1170) (12 notes)
- Microcontrollers (p. 1211) (6 notes)
- Memory hardware (p. 1250) (4 notes)
- Communication (p. 1264) (4 notes)
- Radio (p. 1278) (3 notes)
- STM32 microcontrollers
- Skin electrodes
- Healthcare

Locking telescope

Kragen Javier Sitaker, 02021-04-07 (updated 02021-12-30)
(2 minutes)

Looking at a telescoping antenna today it occurred to me that there was a way to stabilize telescoping mechanisms for the purpose of, for example, folding bicycles, with a bayonet-connector-like approach.

Specifically, given two coaxial tubes, one inside the other, you have two or more pegs in one of them (say, the inner) which engage slots in the other (say, the outer) when they're close to the fully extended position; by rotating the tubes relative to one another, the pegs move into channels that move the tubes into a *really* fully extended position, working against a spring, and when the rotation is continued, the channel allows the pegs to slip into a detent, allowing the spring to relax slightly. The only difference from a BNC connector is that the action is reversed: the pegs engage when the tubes are nearly apart instead of nearly together, resisting compression on the column formed by the two joined tubes instead of tension. (If you need to resist tension you can do that better with a cable, which can be tensioned to hold the strut in compression.)

Optionally you can include a second analogous set of channels to lock the tubes in the fully collapsed position as well, or any number of other positions.

In this way a telescoping tube can provide a strong structural member that can be easily collapsed for storage or transport.

Topics

- Contrivances (p. 1143) (45 notes)
- Mechanical (p. 1159) (17 notes)
- Connectors

Logarithmic low-power SERDES

Kragen Javier Sitaker, 02021-04-08 (4 minutes)

I think the usual way to make a SERDES is with a shift register. Let's consider the deserialization case, where we want to, say, convert a 2Gbps serial connection into a 31.25 MHz 64-bit parallel interface. We use a shift register with 64 bits in it clocked at 2GHz, toggling on average 32 bits per cycle and thus 64 gigatoggles per second. Every 64 cycles, we latch its current state into a 64-bit output register, also toggling 32 bits on average, but this only pushes us from 64 to 65 gigatoggles per second.

A different way to do this is with a logarithmically slowing series of stages, like a DSP polyphase filter. Initially we have a 2-bit shift register which is clocked at 2GHz as usual. Every 2 cycles, its output gets shifted into the low bits of two more 2-bit shift registers, which are thus clocked at 1GHz. Every two shifts, their 4 bits of output are shifted into the low bits of four 2-bit shift registers, which are clocked at 500MHz. And similarly for 16 bits clocked at 250MHz, 32 bits clocked at 125MHz, 64 bits clocked at 62.5MHz, and finally a 64-bit output latch register clocked at 31.25MHz. This gives us 2 × 1 gigatoggles + 4 × 500 megatoggles + 8 × 250 megatoggles + 16 × 125 megatoggles + 32 × 62.5 megatoggles + 16 × 31.25 megatoggles × 2 = 10 gigatoggles per second, 6.5 times lower power consumption than the straightforward shift register approach. Moreover, it's possible that the stages after the first 2 or 4 bits can be driven at lower voltage or built in a simpler fashion, because they don't need to run nearly as fast.

Instead of 128 latches, now we need 190, but that's less than 50% overhead. It's probably useful to stagger the phases of the 7 different clocks to smooth out the load current and thus keep the voltage rails more constant.

(You can of course just time-reverse this to get a serializer instead of a deserializer.)

A slightly different way to design the device is as a binary tree of 127 flip-flops. The flip-flop at the root of the tree is clocked at 2GHz, fed directly from the incoming data stream. Each of its children is clocked at 1GHz, but on alternate 2GHz clock cycles, latching in the root's output. Each 2GHz clock cycle, one of the four latches at the next level of the tree is clocked, and so on. So on each 2GHz clock cycle, the bits shift along some 7-bit path from the root of the tree down to one leaf, but which leaf changes every cycle. This way the number of flip-flops clocked per cycle is constant at 7, and there are on average 3½ toggles per cycle.

However, this is sort of cheating, because an output bit still changes every 500 ps. If you want to read it at 31.25MHz, that probably isn't okay, so you probably need another 64 bits to latch the output, bringing this design back up to 191 latches, and clocking all the output latches once every 64 cycles.

A different approach is to distribute the input signal using pass transistors or write-enable bits or something instead of flip-flops.

This approach is exactly like the binary-tree approach, except without all the intermediate tree levels: the input signal is buffered and possibly latched at 2GHz and driven onto the inputs of 64 leaf flip-flops, but only one of those flip-flops is clocked each cycle. In a sense this requires 64-way fanout, which requires a fairly beefy buffer for it to be fast.

Topics

- Contrivances (p. 1143) (45 notes)
- Electronics (p. 1145) (39 notes)
- Energy (p. 1170) (12 notes)

:fqozl, a normal-order text macro language

Kragen Javier Sitaker, 02021-04-09 (updated 02021-07-27)
(14 minutes)

Textual macro replacement languages like m4 are famous for being easy to implement and eliminating artificial barriers to factoring out duplication. They also have a very low barrier to entry for new programmers. However, they also have a lot of problems.

m4 is notoriously hard to use, although it's often effective. Part of the problem is that data is re-executed both on its way into and on its way out of macro invocations, so you often need quoting — confusingly, multiple levels thereof. Worse, if you are missing a level of quoting, your code will often appear to work until some previously inert data happens to have a macro invocation in it.

m4's design has a number of other avoidable defects. It's whitespace-sensitive in a way that impairs readability. Often m4 macros collide with ordinary words, resulting in accidental macro invocations that corrupt the text. Since it was defined, the ASCII apostrophe has been redefined as a terrible symmetrical typewriter glyph, depriving m4's default quote characters of their symmetry. And m4's parameter-passing mechanism is so purely positional that, as in Forth, you can't even name your arguments.

Can we define a text macro language that retains the basic processing paradigm of m4, m6, and cpp, but has a better balance of power and usability?

Existing macro languages

TeX is a pure macro-expansion system, perhaps the most successful attempt, but I suspect we can do better. Tcl is semantically mostly a macro-expansion language, and although it's somewhat limited it's at least usable. Mooers's TRAC is hard to find information about. MediaWiki templates, sh, make, PHP, ES6, Perl, cpp, macro assemblers, and many other languages implement string interpolation in more or less central ways.

Normal-order macro expansion

If macro *output* is not subject to macro expansion, we have “applicative-order macro expansion”, and m4 loses its Turing-completeness, making it much more comprehensible (all the macro invocations that will be expanded are explicitly present in the input file) but also much less useful. (m4's predecessor m6†, which was included with early versions of Unix, offered the call-time option to expand a macro in this fashion by terminating the call with a semicolon.)

On the other hand, if macro *input* is not subject to macro expansion, much of the need for quoting disappears, but Turing-completeness remains. This is “normal-order macro expansion”. Done naïvely, there are cases where it will take

exponentially longer than m4's strategy, but I think these can be mostly avoided in practice, and a more sophisticated implementation can optimize them away. Make's standard `$(variable)` mechanism works this way, though without the ability to define parameterized macros.

Tcl, instead, doesn't rely on macro expansion for computational power; its `proc` mechanism is not defined in terms of macro expansion producing an enormous string, just the construction of the arguments to a Tcl command.

† "the program contains about 25 subroutines, totaling about 600 executable statements," according to the Bell Labs Computing Science Technical Report #2 about m6 in 01971.

Lexical syntax for :fq0zl

m6† used "warning characters" to delimit macro invocations, a feature that is present in a certain sense in cpp and has been added again in GNU m4. In standard m4 we can write this and get 6:

```
define(`mylen',`ifelse($1,,0,`eval(1+mylen(substr($1,1)))')')dnl
mylen(kanawa)
```

But, in m6, where the quote characters were `<>` and the default warning characters were `#:`, I think you would have written:

```
#def,mysize,<#if,#seq,$1,:,0,1,<#add,1,#mysize,#substr,$1,1:::>:>:
#mysize,kanawa:
```

(As mentioned above, replacing `:` with `;` yielded applicative-order expansion, in which the macro's output would not be expanded further.)

Here `#seq,$1,:` tests whether the first argument to mylen was string-equal to the empty string, in which case the invocation of the builtin `#if` macro returns 0; otherwise it invokes `#eval` to recurse. I think the inner `<>` are necessary to prevent the recursion from being evaluated infinitely before invoking `#if`.

In GNU m4, if `changeword` is enabled, I think we can get a similar effect like this, but I don't have m4 compiled with that at the moment to test:

```
changequote(<,>)changeword(<#\([_a-zA-Z0-9]*\)>)
#define(<mylen>,<#ifelse($1,,0,<#eval(1+#mylen(#substr($1,1)))>)>)#dnl
#mylen(kanawa)
```

Overall this is all pretty nasty and unreadable to my eyes. I'd rather separate arguments with arbitrary amounts of whitespace and use an explicit, nestable, and visually symmetrical circumfix syntax for macro invocation. Single ASCII characters only offer a few choices:

```
<mylen kanawa>
[mylen kanawa]
(mylen kanawa)
{mylen kanawa}
```

```
‘mylen kanawa’  # formerly ASCII, now sabotaged by aping MS-DOS
```

Of these, I like <mylen kanawa> best — it most strongly implies that it's a placeholder — but that would run into a lot of trouble if you started trying to macro-expand C programs that say things like #include <string.h> and for (size_t i=0; i<result->len; ++i) all over the place, not to mention PDF files with hex strings like <668531e8e73e8ee1503359167219ef43>, and of course SGML, HTML, and XML — unless you pass undefined macro invocations through unchanged. But I want undefined macro invocations to crash.

So, to avoid these problems, with SGML, HTML, and XML, you must precede the macro name with :, as in <:mylen kanawa>.

Just as in m4 and m6, there’s the problem of how to embed the argument separator within an argument, which is more urgent when the separator is whitespace. You *could* use the same delimiters in such a case, or something like <:q this text is a single argument>, but I think it’s best to expropriate more delimiters in a way analogous to m6’s use of <> — but *only in the context of macro arguments*. And I think the best delimiters to use here are braces {}, backslashing } and backslash inside them if you need unbalanced braces.

And I think named arguments are pretty important. So instead of

```
define(`mylen',`ifelse($1,,0,`eval(1+mylen(substr($1,1)))')')
```

we might write

```
<:def <:mylen s>
    <:ifelse %s {} 0 <:eval 1+<:mylen <:substr %s 1>>> >
>
```

No quoting is needed here because we’re using normal-order macro-expansion; the <:mylen s> and <:ifelse ...> calls are parsed so we can see where they end, avoiding the need for extra {}, but they aren’t macro-expanded until and unless they end up in a strict context, such as a top-level output stream or the argument of <:eval ...>. And %s expands to the parameter s, as in MS-DOS batch files or in SGML parameter entities:

```
<!ENTITY crap "#PCDATA | %font | %phrase | %special | %formctrl">
```

However, in :fqozl, the substitution is only carried out on the text of a macro definition, and only using the parameters that are lexically within scope — it’s not done throughout the rest of the file, as in an SGML DTD (the above SGML defines &crap; to expand to all that crap, with %font and the like additionally expanded, anywhere in any document ruled by this DTD).

As in SGML, you can optionally terminate the parameter name with a ;, which is useful for contexts that would otherwise be a problem:

```
<:def <:mylen %s>
    <:ifelse %s; {} 0 <:eval 1+<:mylen <:substr %s; 1>>> >
>
```

Consider, for example, this definition from the Hammer parsing library:

```
#define HAMMER_FN_DECL_NOARG(rtype_t, name)            \
  rtype_t name(void);                                  \
  rtype_t name##__m(HAllocator* mm__)
```

We can define the equivalent in :fqozl as follows:

```
<:def <:hammer_fn_decl_noarg rtype_t name> {
    %rtype_t %name(void);
    %rtype_t %name;__m(HAllocator* mm_);
}>
```

Without the disambiguating ; we would have `%name__m`, which would abort with an error, since there is no parameter in scope named `name__m`.

Named parameters can have defaults, and you can pass parameters by name. So, for example, corresponding to this MediaWiki markup with its named parameters:

```
{{Infobox laboratory equipment
|name         = Holter monitor
|image        = HolterAFT1000.jpg
|caption      = Holter monitor

|inventor     = [[Norman Holter]] and Bill Glasscock at Holter Research Laborator◦
◦y
}}
```

we might have

```
<:infobox-laboratory-equipment
    name = {Holter monitor}
    image = {HolterAFT1000.jpg}
    caption = {Holter monitor}

    inventor = {[[Norman Holter]] and Bill Glasscock at Holter Research Laborator◦
◦y}
>
```

The spaces around the `=` are optional, so parsing this from left to right requires retconning `name` from being a parameter *value* "name" to being a parameter *name*. When we define the macro we can provide it with default parameter values:

```
<:def <:infobox-laboratory-equipment
        name caption inventor image={} model={}>
...>
```

It's reasonable to question having two separate replacement mechanisms, one for macros `<:s>` and the other for parameters `%s`. You

could of course provide parameters as locally-defined macros, but my intuition is that bloating references to what are really just local variables to a minimum of four characters is excessive and would make :fqozl clumsy to use.

Static function arguments

By defining a macro that expands to a `<:def ...>` we can do imperative programming, but what about higher-order functional programming?

In m4 we can pass function arguments by name:

```
define(parenthesize,`($1)')dnl
define(bracize,`{$1}')dnl
define(hello,`$1(h)$1(e)$1(l)$1(l)$1(o)')dnl
hello(`bracize')
hello(`parenthesize')
```

The `hello` macro invokes the macro whose name is passed in as an argument, so we get:

```
{h}{e}{l}{l}{o}
(h)(e)(l)(l)(o)
```

It took me about half an hour, and rereading much of the GNU m4 manual, to figure out that the reason this wasn't working was that I had forgotten to quote the argument to `hello`. In :fqozl we can do precisely the same thing, but quoting is unnecessary:

```
<:def <:parenthesize x> (%x)
><:def <:bracize x> {{%x}}
><:def <:hello f> <:%f h><:%f e><:%f l><:%f l><:%f o>
><:hello parenthesize>
<:hello bracize>
```

If we change the definition in m4, we can pass in a curried function instead; this produces the same output:

```
define(wrap, `$1$3$2')dnl
define(hello, `$1,h)$1,e)$1,l)$1,l)$1,o)')dnl
hello(`wrap(`(',`)'')
hello(`wrap({,}')
```

However, though this works in this case, it is clearly monstrous; it's incompatible with the previous `hello` definition, and not only is the definition of `hello` unreadable (you can't even see the nesting structure), it's buggy when you pass it alphanumerics:

```
define(wrap, `$1$3$2')dnl
define(hello, `$1,h)$1,e)$1,l)$1,l)$1,o)')dnl
hello(`wrap(_,_')
```

This produces the output

```
_h_wrap(_,_,e)_l_wrap(_,_,l)_o_
```

Two of the five calls to the callback argument got cabbaged because when the scanner got to them they'd undergone token pasting. The m6/:fqozl design is somewhat safer here, but this is far from the only such problem with macro expansion.

It's not impossible to fix this bug within m4:

```
define(wrap, `$1$3$2')dnl
define(hello, `$1,h)`'$1,e)`'$1,l)`'$1,l)`'$1,o)')dnl
hello(`wrap(_,_')
```

This produces the desired output:

```
_h__e__l__l__o_
```

But the definition is, if possible, even uglier than before.

The straightforward equivalent definition in :fqozl would use the same definition of `:hello` as above:

```
<:def <:wrap l r m> %l;%m;%r;
><:def <:hello f> <:%f h><:%f e><:%f l><:%f l><:%f o>
><:hello {wrap ( )}>
<:hello {wrap \{ \}}>
```

But first we should explore the question of expansion semantics in more detail.

Expansion semantics

A big problem with make, sh, and m4 is that they tend to have subtle bugs when handling "special characters": make totally fails on filenames with spaces. sh by default re-splits the results of $variable and `` `command` `` expansion to "help" you use shell variables and command output as arrays of filenames instead of individual filenames. The m4 behavior I exploited above to pass in a curried function as an argument is not only pretty unreadable in this context, it's also an abundant source of bugs.

Another problem with the m4 semantics is that they're slow: the macro output must be rescanned character by character in case the token boundaries have moved.

We have to scan the macro arguments when we first encounter them in order to figure out where they end. The right solution is to store the resulting list structure and treat *that* as primary, rather than the character-by-character syntax. This need not diminish the abstraction power of the language.

Consider a macro invocation like this:

```
<:foo %a b %c>
```

It is desirable that we can statically know that this invocation invokes `foo` with three arguments, the second of which is the string `b`, which is a positional parameter. It is not desirable for `b` to be the

second or third argument depending on the value of a.

I'll handwave over actually establishing semantics here for now, though.

Lambda-lifting to construct data structures out of curried functions

With this ability to construct and invoke curried functions, we can define naturals, booleans, and Lisp-style lists in the λ-calculus fashion, even though we don't have anonymous functions. To examine a natural, we invoke it with two continuations, one to return if the natural is zero, and another to invoke with the number's predecessor if it is nonzero:

```
<:def <:zero ifzero ifsucc> %ifzero>
<:def <:succ n> {succ2 %n}>
<:def <:succ2 n ifzero ifsucc> <:%ifsucc %n>>
```

Given this definition, we can define addition recursively: 0+y is just y, and succ(n)+y is succ(n+y):

```
<:def <:add x y> <:%x ifzero=%y ifsucc={add2 %y}>>
<:def <:add2 y n> <:succ <:add %n %y>>>
```

Comparing a number to zero is fairly trivial:

```
<:def <:eq0 x> <:%x ifzero=true ifsucc=eq01>>
<:def <:eq01 _> false>
```

However, what are true and false? We can define them in the same way, as functions that invoke different continuations:

```
<:def <:true iftrue iffalse> %iftrue>
<:def <:false iftrue iffalse> %iffalse>
```

Now we can compare two naturals with the same recursive approach we used for addition. If the left argument is zero, then we get the result by checking to see if the right argument is zero:

```
<:def <:eq x y> <:%x ifzero=<:eq0 %y> ifsucc={eq2 %y}>>
```

Otherwise,

```
<:def <:eq2 y x1> <:%y ifzero=false ifsucc={eq %x1}>>
```

```
<:def <:nil ifnil ifpair> %ifnil>
<:def <:pair car cdr ifnil ifpair> <:%ifpair %car %cdr>>
<:def <:mapcar f list> <:%list nil {mapcar2 %f}>
<:def <:mapcar2 f car cdr> <:pair <:%f %car> <:mapcar %f %cdr>>>
<:def <:append xs ys> <:%xs %ys {append2 %ys}>>
<:def <:append2 ys car cdr> <:pair %car <:append %cdr %ys>>>
```

Topics

- Programming languages (p. 1192) (8 notes)
- End user programming (p. 1217) (6 notes)
- m4 (p. 1352) (2 notes)

- Macros

Forming steel with copper instead of vice versa

Kragen Javier Sitaker, 02021-04-16 (updated 02021-06-12)
(2 minutes)

Resistance welding with copper electrodes is the standard way to spot-weld steel; for high duty cycles they are water-cooled, but low duty cycles are often just solid copper. You'd think this would totally fail because copper melts at 1084.62° while steel typically melts around 1500°, so the copper would melt long before the steel. In fact, though, at room temperature, copper has an electrical resistivity of about 16.8 nΩ m, while 1010 carbon steel is more like 143 nΩ m, and copper's thermal conductivity of 400 W/m/K is also greater than carbon steels' of around 30–100 W/m/K.

So the same current density running through copper and steel generates 8.5 times as much heat per unit volume in the copper, and the copper can conduct it away from the point of generation 4–12 times as fast, so the temperature rise in the copper tends to be about 30–100 times less. These numbers change at higher temperatures but I think the overall tendency remains the same. In EDM, even with the ultra-high temperatures of the arcs, copper electrodes are considered to be "free of wear".

Of course the heat equation tells us that the copper and steel in contact immediately form a continuous temperature distribution, so what you're really doing is melting steel *under* the surface, while the surface steel remains solid, chilled by the copper to under 1000°.

It occurs to me that you could use this same approach for *forming* the steel instead of welding pieces of it together: by locally melting it with a pulse of current, it can be formed by pressing even a soft copper ball into it. By doing this repeatedly while moving the copper ball around to precise positions in three dimensions, you can achieve arbitrarily complex surface geometry without the high side loads and noise of a mill or lathe, and regardless of how hard the steel is. If it's a high-carbon steel, the forming process will inherently case-harden the product, as each melt is quenched by the mass of the steel. With proper planning of the toolpaths, especially the finishing toolpath, it should be possible to keep heat-induced distortion of the workpiece small by keeping the heating very localized.

Topics

- Contrivances (p. 1143) (45 notes)
- Manufacturing (p. 1151) (29 notes)
- Physics (p. 1157) (18 notes)
- Strength of materials (p. 1164) (13 notes)
- Machining (p. 1165) (13 notes)
- Pulsed machinery (p. 1167) (12 notes)
- Self replication (p. 1204) (6 notes)

- Steel (p. 1222) (5 notes)
- Copper (p. 1234) (5 notes)
- Forming (p. 1295) (3 notes)

Notes on pricing of locally available oscilloscopes

Kragen Javier Sitaker, 02021-04-16 (updated 02021-07-27) (2 minutes)

Scoping out oscilloscopes again.

02021-04-16

The dollar is currently at AR$140.

A Hantek 6022be USB scope goes for AR$13900 (US$99): 20 MHz, 8 bits, 48Msps, two channels, new, two probes, up to 5V, or 50V with the probes set to 10×. It's fully supported in sigrok and loads its firmware over USB. That's a seller in Chacarita, and several other sellers have similar prices.

Several sellers sell the toy single-channel DSO138 200kHz for less, like AR$8000 (US$50).

Used analog oscilloscopes are the other option; a 20MHz 2-channel Hitachi V-212 in Rosario for AR$20k (but no shipping), a 20MHz 2-channel Protomax ??? missing its probes in Buenos Aires for AR$13800, a 20MHz 2-channel Pintek PS-200 in Haedo for AR$28000, a 20MHz 2-channel Beckman 9102 in Martínez for AR$50k, a 20MHz single-channel GW GOS-3310 missing its probes and maybe not working in Tres Lomas for AR$9000, an 8MHz single-channel Monfrini missing its probes and maybe not working in Lomas de Zamora for AR$8900, a 20MHz 2-channel Pintek PS-200 for AR$14500 in Santa Rosa, San Luis, and so on.

None of the analog oscilloscopes are all three of ① complete and working, ② possessed of two channels, and ③ cheaper than the Hantek scope by more than a trivial amount. Most of them are simultaneously more expensive and less capable. And there don't seem to be any USB scopes that are significantly cheaper.

But I don't know if it's really worth spending US$100 on. Maybe I should wait to see if something cheaper comes along.

Topics

- Electronics (p. 1145) (39 notes)
- Pricing (p. 1147) (35 notes)
- Argentina (p. 1200) (7 notes)

Can you get JLCPCB to fabricate a CPU for you affordably from "basic" parts?

Kragen Javier Sitaker, 02021-04-17 (updated 02021-12-30)
(9 minutes)

JiaLiChuang PCB will fab ten tiny prototype boards for you for US$2. They also offer a service where they will stuff the board for you with certain components, ones from LCSC's "LCSC assembled" list, which has about 30k components. I have a couple of questions about this:

- How much does it cost?
- What are the parts most suitable for building a CPU?
- How big would such a CPU be?

Electronoobs reports on his experience. He was paying €0.0035 and €0.0097 for precision 0805 resistors, €0.0559 for large bright blue LEDs, €0.0530 for his WS2811 LED drivers, etc., *all* of which were "extended" parts; and €0.0031 for some 0805 180Ω 1% resistors, €0.0670 for some SOT-23 P-MOSFETs (AO3401A), and €0.0155 for some 0.1μF X7R 50V 0805 caps, which were selected from among their 689 "basic" components that don't require a US$3 per-component-type fee. However, I guess JLCPCB was paying him to pay them. He was building 30 7-segment displays with 57 LEDs on them, 80 components total, for about US$2.90 each (€79.30 or US$85 for 30 PCBs is US$2.83⅓ each), controlled by some WS2811 8-SOICs. They were 2-layer boards with plated-through vias.

He reports that the stuffing service only supports SMD, and only on one side, and only for 2-layer and 4-layer green boards. Apparently since then JLCPCB has added 6-layer and some other colors of solder mask.

Roughly estimating, 10,000 SOT23 transistors would be enough for a CPU, which would be 100×100 transistors. You could maybe bit-slice a CPU across multiple boards, since I think JLC has a minimum of 5 or 10 boards per prototype order.

There's a third-party parametric search engine for the parts library, which has a cached zip file of JSON listing the parts, which is 260 mebibytes.

So, what do the 689 "basic" parts include?

Most of the parts are not in stock at any given time, a common source of frustration among forum posters.

Roughly half the "basic" parts are resistors. Some of these are very small; the 0.34¢ Uniroyal 4D02WGJ0102TCE is an array of four 1kΩ 5% resistors in a 1 mm × 2 mm package, which I guess is twice the size of an 0402. That brings the cost per resistor down to 0.085¢, plus 0.3¢ for soldering. 10kΩ, 4.7kΩ, 470Ω, and 33Ω are also available in this form, or 4 in an 0603 (0.43¢ or maybe 0.38¢). Other

denominations included in small form factors include 4.7MΩ, 2.2MΩ, 1.5MΩ, 1.2MΩ, 1MΩ, 620kΩ, 300kΩ, 270kΩ, 100kΩ, 75kΩ, 49.9kΩ, 40.2kΩ, 24kΩ, 22kΩ, 8.2kΩ, 6.8kΩ, 6.2kΩ, 5.6kΩ, 5.1kΩ, 4.3kΩ, 3.9kΩ, 2.2kΩ, 1.2kΩ, 1kΩ, 750Ω, 680Ω, 360Ω, 330Ω, 300Ω, 240Ω, 120Ω, 100Ω, 75Ω, 56Ω, 33Ω, 22Ω, 10Ω, 2.2Ω, 1Ω, etc.; these are about 0.3¢ for (rare) 1206s (¼W!), 0.2¢ for 0805s, 0.1¢ for (rare) 0603s, or 0.05¢ for 0402s, and ±1% is typical tolerance. This means the discrete 0402s are actually *cheaper* than the resistors in the arrays (usually) but take up twice as much space.

Much of the remainder is MLCCs: roughly 1.3¢ for a 1206, whether C0G, Y5V, or X7R, 0.4¢ for an 0603, or 0.1¢ for an 0402. With MLCCs there's a tradeoff between size, voltage, capacitance, and precision.

In logic and quasi-logic, we have the 555 timer (7.58¢); the 74HC244 octal tristate buffer (12.87¢); the 74HC14 hex inverting Schmitt trigger (7.64¢); the 74HC04 hex inverter (9.11¢); the 8-bit 74HC164 serial-to-parallel (9.04¢) and 74HC165 parallel-to-serial (10.1¢) shift registers, as well as the 8-bit (?) 74HC595 serial-to-parallel latched-output shift register (10.84¢); the 74HC08 quad AND gate (9.85¢); the 74HC138 3-to-8 decoder (9.15¢); the 8-channel analog mux/demuxers CD4051 (15.01¢), CD4052 (14.56¢), and CD4053 (14.35¢); the 74HC573 tristate octal transparent latch (22.14¢); the 74LVC4245 tristating bidi 3–5 volt octal level shifter (32.46¢); and the 74HC245 tristating bidi octal buffer (18.13¢).

I think the combination of a 74164 and a CD4051 gives you an async 3-LUT for 24.05¢ plus 4.5¢ for assembly (14 74164 pins and 16 CD4051 pins), 28.55¢ total: you shift your LUT bits into the 74164 and feed them into the CD4051's 8 inputs, then drive its channel-select inputs from the logic signals you actually want to compute on. If one of your logic signals is available in inverted form as well, you can gang together two such combos (wiring the CD4051 outputs together and connecting the fourth logic input to INH on one CD4051 and inverted to INH on the other) to get a 4-input LUT.

Even with a decoder, the 74244 would not work for this because it only has two output enable pins, controlling four bits each; the 74573 only has one.

There's also the 7-darlington 16-pin ULN2003 (13.64¢ + 2.4¢ assembly = 16.04¢) and the 18-pin 8-darlington ULN2803 (50.32¢), so the ULN2003 costs 2.29¢ per transistor as long as you don't mind all the emitters being tied together. Discrete transistors cost less, though: a SOT-23-3 P-MOSFET like the Leshan Radio Company LBSS84LT1G is 1.34¢ (+0.45¢ assembly), and the Changjiang Electronics Tech SOT-23-3 2N7002 (a N-MOSFET, of course) is 1.23¢ (+0.45¢). Bipolar transistors are even cheaper — the SOT-89-3 PNP B772 from Changjiang is 5.38¢ (+0.45¢), but the SOT-23-3 Changjiang NPN S9013 is only 1.13¢ (+0.45¢) or possibly 0.929¢?, and its complementary PNP S9012 is 1.16¢ or maybe 0.891¢, and their PNP S9015 is only 0.98¢ (+0.45¢) or 0.909¢, and the CJ MMBT3904 NPN is 0.87¢ (+0.45¢) or 0.682¢.

SOT-23 seems to be the smallest discrete transistor they have in "basic". SOT-89 is a bit larger, 4.5 mm × 4.1 mm.

The ULN2003 is packaged in an SOIC-16, which is 10 mm × 6 mm, which works out to 8.57 mm² per transistor, very nearly the same as an SOT-23 per transistor.

No connectors at all are included among "basic" parts.

They do have LEDs among "basic" parts: 1.63¢ for yellow 0603s, 1.54¢ (or maybe 0.909¢?) for red 0805s, 1.82¢ for green, 1.1¢ for blue 0603s, 0.75¢ or 0.57¢ for blue 0805s, and 0.78¢ or 0.56¢ for white 0805s. This suggests a price of around US$20 per kilopixel if you're into that kind of thing.

10ppm SMD 8MHz crystals are 22.78¢ or maybe 18.27¢ in quantity 1 in an "SMD-5032_2P" package.

Power diodes cost 0.5¢, Schottkys 2.15¢ (or maybe 1.32¢), zeners 1.35¢ (or maybe 0.96¢) (5.6V and 3.3V only), fast recovery rectifiers 0.74¢ or maybe 0.6¢, 1N4148Ws 0.78¢ (or maybe 0.77¢), and 5.8V TVSs 2.91¢ (or maybe 2.94¢).

CPU size estimation

If we estimate a basic CPU as being 1000 gates, evenly split between AND and NOT, that would be 84 74HC14s in and 125 74HC08s, all in SOIC-14s. An SOIC-14 is 6 mm × 8.6 mm, so these chip packages alone would be 10784 mm², about 100 mm × 100 mm, or 1078 mm² on each of 10 boards, 33 mm square. I think the parts would cost US$8.82 for the 74HC14s, US$9.00 for the 74HC08s, plus US$4.39 to solder the 2926 joints, for a total of US$22.21. In practice you'd need about twice that much space, and of course you'd have registers and things in there too.

If we estimate a basic CPU as being 500 3-LUTs, that's 500 74164s (in SOIC-14) and 500 CD4051s (in SOIC-16), which would be US$142.75. An SOIC-16 is 60 mm² so this is (51.6 + 60) × 500 = 55800 mm². If divided into 10 square boards, each would be 75 mm square, or more realistically a bit bigger than that.

If we estimate a basic CPU as being 10,000 transistors, each with two resistors, each SOT-23 transistor is about 3 mm × 3 mm, and each 0402 resistor is 1.0 mm × 0.5 mm, for a total of 10 mm² per transistor-plus-resistors. This would give us a total of 100000 mm² or 10 100 mm × 100 mm boards. If the transistors are half 2N7002s (1.23¢) and half LBSS84LT1Gs (1.34¢) and the resistors all cost 0.05¢, then we have US$10 for 20,000 resistors, US$61.50 for 5000 2N7002s, US$67 for 5000 LBSS84LT1Gs, and US$105 for all the SMD soldering, for a total of US$243.50.

All of the above omits the costs of things like voltage regulation and bypass capacitors, but I think those would probably be a minority; also it omits the cost of the PCB fabrication itself.

Conclusions

SSI random logic chips are probably the most suitable way to build such a CPU, which would probably cost under US$50 through JLCPCB's service and be less than 150 mm square; if it's 16 layers thick, it should fit into a 40 mm cube. Some hand-soldering will be needed to connect multiple boards together.

Topics

- Electronics (p. 1145) (39 notes)
- Pricing (p. 1147) (35 notes)
- Digital fabrication (p. 1149) (31 notes)
- Manufacturing (p. 1151) (29 notes)
- Bootstrapping (p. 1171) (12 notes)
- Independence (p. 1215) (6 notes)
- JLCPCB (JiaLiChuang) (p. 1360) (2 notes)

Safe FORTH with the FORTRAN memory model?

Kragen Javier Sitaker, 02021-04-21 (updated 02021-06-12)
(2 minutes)

What if we use the Fortran memory model in Forth? With bounds checking? No more free access to memory. Like, a segmented kind of thing.

Create a scalar variable x with value 3:

```
3 value x
```

Set it to 4:

```
4 to x
```

Create an array a containing the values 0, 1, 4, 9, 16:

```
create a 0 , 1 , 4 , 9 , 16 ,
```

Load 9 from its position 3, with bounds checking:

```
a 3 @
```

Store -4 in its position 2, overwriting 4:

```
-4 a 2 !
```

Define a word "cons" that allocates two cells and stores values on the stack in them:

```
: cons create , , ;
```

Use it:

```
3 4 cons mycon
```

Define a word "cdr" that loads the second field, and use it:

```
: cdr 1 @ ;
mycon cdr
```

Define a different word "cons" that defines words that push the two values on the stack when run:

```
: cons create , , does> dup 0 @ swap 1 @ ;
```

Define a word that creates a one-dimensional array of cells of a given size:

```
: array create cells allot ;
```

Use it:

```
10 array xs  500 xs 3 !  xs 3 @   \ retrieves the 500
```

For multi-dimensional arrays and for things like dynamically allocating conses, we probably need pointer arithmetic. But we can bounds-check the pointer arithmetic with fat pointers. That allows us to say, for example:

```
0 value this
: 2darray create dup , * cells allot  \ save Y-dimension
   does> to this  this 0 @ * cells this + 1+ ;
```

This lets us get a pointer to the Nth row of the 2darray, and we can then use that row as a normal 1-D array, but with weaker bounds-checking:

```
32 64 2darray tile  5 tile  \ get pointer to row 5 of 32
100 @    \ will work even though the row is smaller than that
```

Similarly, this allows us to allocate cons cells from an arena that we can later garbage-collect. Address arithmetic within the arena is fine; trying to index out of your arena will fail.

You can also have `allocate` if you want to be able to create new "segments" at run-time.

One problem I've run into with the similar semantics in C is that it's impossible to write memmove.

Topics

- Programming (p. 1141) (49 notes)
- Safe programming languages (p. 1172) (11 notes)
- Programming languages (p. 1192) (8 notes)
- FORTH (p. 1231) (5 notes)
- Memory models (p. 1285) (3 notes)

Manually writing code in static single assignment (SSA) form, inspired by Kemeny's DOPE, isn't worth it

Kragen Javier Sitaker, 02021-04-21 (updated 02021-06-12)
(3 minutes)

Darius mentioned DOPE, Kemeny's predecessor to BASIC, and it occurred to me that you could simplify it further by unifying destination variables with line labels.

In DOPE each line of the program was similar to a line of assembly, with implicit variable declarations and explicit line numbers and inputs and outputs. So to calculate $-b+\sqrt{(b^2-4ac)}$ you might say:

```
10 * 4 a d    # d := 4 * a
20 * d c d    # d := c * d
30 * b b q    # q := b * b
40 - q d d    # d := q - d
50 sqr d d    # d := √d
60 - d b d    # d := b - d
```

The line numbers permit gotos. If we require destination variables to be unique, we could have just as much goto messiness with less verbosity, and you could also shift the operand one position to the right to improve readability slightly:

```
d:  4 * a     # d  := 4 * a
d2: d * c     # d₂ := d * c
q:  b * b
d3: q - d2
d4: d3 sqr
s:  d4 - b
```

Labels can be omitted for statements executed for side effects. Conditionals and loops would require some equivalent of SSA φ functions. DOPE used FORTRAN-style arithmetic IF (an opcode named "C" with 4 operands), but we can avoid such abominations with conditional jump or conditional skip; for absolute value, for example:

```
pos:  x > 0
      pos → r        # jump to r if x is positive
xn:   0 - x          # negate x into xn
r:    x \/ xn        # set r to either x or xn, whichever is most recent
```

A different approach, though it doesn't subsume the need for jumps, is a conditional operator:

```
pos:  x > 0
```

```
xn:   0 - x
r:    pos ? x xn
```

Here's a dot-product routine for nonzero-length vectors without a high-level FOR construct:

```
init: 0
i0:   0
i:    i0 \/ i'
s:    init \/ t
ai:   a @ i
bi:   b @ i
p:    ai * bi
t:    s + p
i':   i + 1
cont: i < n
      cont → i
```

Changing it to handle zero-length vectors makes it two lines longer:

```
init: 0
i0:   0
i:    i0 \/ i'
s:    init \/ t
stop: i = n
      stop → end
ai:   a @ i
bi:   b @ i
p:    ai * bi
t:    s + p
i':   i + 1
      → i
end:  nop
```

For *writing* to arrays, you'd probably need some kind of side-effecting indexed-store operator, like `a ! i x` or something.

However, all of this is pretty shitty compared to Forth — harder to write and harder to read — and Forth is probably just as easy to implement, if not to optimize. These versions are 2–10× smaller, depending on how you count, and I think more readable, though still worse than infix:

```
0 b - b b * 4 a * c * - sqrt +
x 0 < if 0 x - else x then
0 s !  n 0 do  i a @ i b @ *  s +! loop  s @
```

Topics

- History (p. 1153) (24 notes)
- Assembly-language programming (p. 1175) (11 notes)

- Facepalm (p. 1199) (7 notes)
- FORTH (p. 1231) (5 notes)
- BASIC (p. 1303) (3 notes)

Diskstrings: Bernstein's netstrings for single-pass streaming output

Kragen Javier Sitaker, 02021-04-21 (updated 02021-07-27)
(4 minutes)

Bernstein's netstrings are a sort of TLV encoding with arbitrary-precision L and without the T: `foo` is encoded as `3:foo`, and `3:foo,3:bar,` is encoded as `12:3:foo,3:bar,,`. The intent is to make protocols easy to parse reliably without making them non-textual. (One assumes that he considered and rejected Fortran's `12H3Hfoo.3Hbar..` .)

Though they're self-delimiting, they're not fully self-describing; you need some external schema information to distinguish a netstring containing more netstrings from a netstring containing just a blob. So you could, for example, precede each netstring with a type byte: `d12:s3:foo,s3:bar,,` might represent JSON's `{"foo":"bar"}`. Unlike JSON, netstrings permit skipping over the contents in constant time, without parsing them.

However, because the length field is variable-length, and minimal encoding is mandatory (leading zeroes are prohibited), the length of the length field depends on the length of the content. Because the content is *after* the length field, the offset at which the content begins also depends on the content's length. So, when outputting a netstring sequentially, it is not possible to output any of the content before knowing the length of the whole content. So a large netstring in general cannot be emitted in a one-pass fashion.

A higher-level protocol can of course provide a facility analogous to HTTP's "Content-Encoding: Chunked", for example permitting `foo` to be encoded as `s3:foo,`, `s1:f,+2:oo,`, etc. This has drawbacks, though: it eliminates the ability to skip an item in constant time, it adds back in the opportunities for bugs that netstrings were designed to avoid, and without further restrictions it eliminates the bijective nature of the netstrings encoding.

In many contexts, some kind of out-of-band framing indicates the end of data — for example, the file size in a Unix or MS-DOS filesystem, framing bytes in SLIP or PPP packets, the Frame Check Sequence in HDLC, or a fixed-length count field in a program's memory. In these contexts, we can place the length field at the *end* of the representation instead of the beginning, encoding `foo` as, for example, `(foo)3` and `(foo)3(bar)3` as `((foo)3(bar)3)12`, or `\t\tfoo\tbar\n3\n12`. This permits single-pass *output* of an arbitrarily large tree of these strings, requiring memory proportional only to the tree depth, and then constant-time navigation operations on the resulting serialization as long as it is stored in random-access memory. However, unlike bytestuffing or quoting approaches, it does not permit reliable partial parsing of any truncated serialization. For example, `(())2)3(()8)3)12` is also valid, representing the 12-byte string `())2)3(()8)3`, which is the concatenation of the representation of the 3-byte string `))2` and the representation of the 3-byte string `()8`. But its prefixes `(())2)3(()8` and `(())2` are also valid.

Rather than “netstrings” we might call this representation “diskstrings”.

If we concatenate diskstrings with type bytes (prefixed or suffixed) we can represent JSON-like data in a random-access-friendly ASCII-clean way; for example, `(37)2s` might represent the 2-byte binary string `37`, while `(37)2n` represents the number 37_{10}, `((37)2s(37)2n)12a` represents a sequence or array of them both (Python `[b"37": 37]`), `((37)2s(37)2n)12d` represents a dictionary mapping the first to the second (Python `{b"37": 37}`), `?` represents a nil value, `t` represents Boolean true, and `f` represents Boolean false. If canonicalization or rapid searching is important, as in bencode, we can require that dictionary keys be lexicographically ordered (by their representations, to permit comparisons between values of different types).

Prefix type bytes might have more desirable lexicographical ordering properties or be more human-readable: `s(37)2`, `n(37)2`, `a(s(37)2n(37)2)12`. In that case punctuation type bytes would probably improve readability further by breaking up the visual unity of “2n”: `,("(37)2+(37)2)12` or something.

If compactness were more important than the extra error-detection, the type bytes could be further merged with the trailing delimiter, and the leading delimiter eliminated, thus `37H237#2@8`. But it’s hard to see when this would be a good tradeoff.

Topics

- Programming (p. 1141) (49 notes)
- File formats (p. 1233) (5 notes)

Phased-array imaging sonar from a mesh network of self-localizing sensor nodes

Kragen Javier Sitaker, 02021-04-27 (updated 02021-12-30)
(8 minutes)

Suppose you have some devices that send out periodic ultrasound signals to each other. If one sends a ping and another responds, the two can measure the distance between them using the speed of sound. If you have three, they can thus measure their pairwise distances and thus angles, and if there are four or more, they can measure more or less precise three-dimensional positions up to rotation and reflection.

Multipath in this case should mostly just produce extra signals that are delayed by more; it should usually be possible to distinguish the shortest-path signal.

The array of devices can then coordinate to function as a phased array for sonar imaging.

The speed of sound

In dry air at sea level, sound travels 343 m/s. Wikipedia says this is 20.05 \sqrt T m/s, so the temperature coefficient is about 0.6 m/s per degree, ranging from 331.4 m/s at 0 degrees up to 354.8 m/s at 40 degrees:

```
>>> C
array([ 0,  1,  5, 10, 15, 20, 21, 30, 40])
>>> 20.05 * (273.15 + C)**.5
array([331.37136701, 331.97738684, 334.39048338, 337.38258383,
   340.34838089, 343.28855628, 343.8735747 , 349.09462596,
   354.80569735])
```

This means that every 1-degree error in temperature estimation of the air will give rise to a scale error of about 0.17%.

There's also an inverse-quadratic variation with the molar mass of the air: dropping the molar mass of the air by 1% will increase the speed of sound by 0.5%. Humidity affects this: higher humidity means lighter molecules and thus faster sounds, although this is partly canceled by the higher adiabatic index of the non-collinear triatomic water molecules, so the variation is only about 1.5 m/s for 0% to 100% relative humidity at STP. The temperature-induced error is probably smaller.

Measurement precision

A 40kHz wavepacket is hard to localize to better than about 25 microseconds, though probably with repetition and averaging we can get down to around 2 microseconds. This is an error of about 0.7 mm. It would be feasible for one or more of the devices to include a meter-long constricted pipe which generates an echo a meter away;

this measures the round-trip time of the sound over a known distance of 2 meters. 0.7 mm over this distance would be an error of 0.035%, plus whatever the error in the tube length is; this could then be used to calibrate the scale factor of the overall mesh model, rather than depending on temperature and humidity measurements. (It also *provides* a combination temperature/humidity measurement, where unknown humidity works out to about ±4.4° temperature imprecision.)

By using a syn-synack-ack triplet it's possible to measure a single round trip according to two clocks at nearly simultaneous, overlapping times, which allows you to measure the relative speeds of the two clocks as long as they drift over a period of time that is large compared to the round-trip time.

By using 2.4 GHz radio signals of known phase, it might be possible to improve this, even in a system made of microcontrollers that are too slow to respond in nanoseconds or even microseconds. One microcontroller starts by transmitting a "pilot wave" that the other locks a second-order PLL onto, driving the phase error to zero, so the second microcontroller's local oscillator is perfectly in phase with the signal it receives.

Then, the first microcontroller stops transmitting, and before the phase has time to drift much, the second microcontroller patches the still-oscillating LO through to an output amplifier hooked up to an antenna.

Now, the first microcontroller can receive this signal and measure the phase difference from its own local oscillator, down to, say, 0.2 radians. This gives you the distance of the round trip (up to multiples of the wavelength) precise to about 10 picoseconds, about 4 millimeters.

A single such measurement has an ambiguity of multiples of about 125 mm. By repeating the measurement at a second frequency, you can eliminate about 97% of the possible candidate distances; doing two more measurements at two more frequencies should bring the number of possibilities down to 1 if you know the distance is less than about 40 km.

This 4-mm precision for radio ranging is by itself worse than the 0.7-mm precision for the sonar approach, but it isn't subject to the 0.17%/° thermal error and 1.5% humidity error of the sonar. By combining the two, you should be able to get the best of both worlds. Suppose you have two nodes about 64 meters apart. Through radio ranging they know their distance to within 4 millimeters, an uncertainty of 0.006%. This allows them to compute the speed of sound between them to that 0.006% uncertainty, which allows them to use sonar to measure the distance to other nodes 10 m away and 20 m away to that same 0.006% precision --- in the 10-meter case, they're limited to 0.7-mm precision, but in the 20-meter case, they get 1.25-mm precision.

This 0.006% precision for the speed of sound translates to measuring the relative humidity to a precision of 0.4% if the temperature is known, or measuring the temperature to a precision of 40 millikelvins if the humidity is known.

(I'm handwaving a bit about things like the difference between

standard deviation, maximum error, and error interval size, because these precision calculations aren't that precise. Hopefully they're within a factor of 3 or so.)

Spirometry

By breathing through the tube, or perhaps two parallel tubes with check valves going in opposite directions, you can do spirometry. A typical tidal volume of 500 ml in a 1 cm^2 tube at a normal breathing rate of 15 breaths per minute amounts to an average exhalation breath velocity of 1.25 m/s, with peaks up to perhaps 3 m/s. This results in a quadratic change in the round-trip time. If outgoing sound travels at 346 m/s and returning sound travels at 340 m/s, then instead of taking 2.9155 ms on the way out and 2.9155 ms on the way back, the ping will take 2.8902 ms on the way out and 2.9412 ms on the way back, pushing the total round trip time up from 5.8309 ms to 5.8313 ms. This 77 ppm change is easily within the precision of a quartz crystal to measure, particularly since we're only interested in cyclic changes over less than a kilosecond.

However, the one-way trip time changes by almost 9000 ppm, so if you can put a sensor node at each end of your spirometry tube and run wires between them, you get 100× more precise spirometry, and an even bigger advantage at lower flow rates.

Phased-array sonar

Having a bunch of nodes like this dispersed in a 3-D space allows you to build a model of all of their locations relative to each other with accuracy and precision of a few millimeters. This ad-hoc phased array of microphones allows you to do passive sonar imaging of sound sources and sound reflectors in the surrounding environment down to a few millimeters as well; also, you can emit pings from the devices to sonically "illuminate" the scene for active sonar.

Orientation sensing

The simplest way to sense the 3-dimensional orientation of an object with such a system is to mount three nodes on it. Polarization of two or more antennas mounted on a node is another way.

Topics

- Contrivances (p. 1143) (45 notes)
- Physics (p. 1157) (18 notes)
- Sensors (p. 1191) (8 notes)
- Metrology (p. 1212) (6 notes)
- Cameras (p. 1301) (3 notes)
- Audio (p. 1304) (3 notes)
- Sonar
- Digital signal processing (DSP)

A boiler for submillisecond steam pulses

Kragen Javier Sitaker, 02021-04-28 (updated 02021-12-30) (10 minutes)

Normally we think of steam engines as being fairly slow, needing minutes to hours to get up a head of steam. But what if our heating elements and the spaces between them are really thin? They could have the fractal-heat-exchanger structure I described in Dercuano, where a very large and very rumpled surface pierced with many thin, short “capillaries” permits the transfer of a great deal of thermal energy into water or another working fluid very quickly.

Suppose you have 1-mm-diameter heating elements made of copper pierced or separated with 1-mm-diameter water channels. What’s the time constant of the relaxation of this thermal system when the copper is much hotter than the water?

Copper has a thermal conductivity of 401 W/m/K, liquid water of 0.5918 W/m/K, and steam around 0.01 W/m/K. My first thought is that a crude approximation is that the water in the middle of the passages is insulated from the heat from the copper by about 200 microns of water plus an insignificant amount of copper, which would give you about 3kW/m^2/K. Each passage, if circular, has a circumference of about 3 mm, so that’s 9 watts per millimeter of passage per kelvin. A millimeter of passage has on the order of 1 mg of water in it, and 9 watts would heat a milligram of 4.184 kJ/kg/K water at about 2000 kelvins per second, so under these assumptions the characteristic relaxation time is about half a millisecond.

That is, if you have a temperature difference of 1000 K, you’ll have 9000 W per mm of passage, which will be heating the mg of water in that mm by 2 megakelvins per second, so every microsecond the water closes 1/500 of the remaining temperature gap.

However, if you have Leidenfrost stuff going on, the water will start to be insulated from the walls by a layer of steam, which will slow the process down by a factor of 60 or so, up to timescales of 30 ms or so. On the other hand, if you have turbulent flow that recirculates the mass flow of water or steam from the center of the passage to the walls and back on submillisecond timescales, the process will accelerate further. On the gripping hand, hot steam condensing onto cold water also accelerates heat transfer, which is how nucleate boiling transfers heat.

So I think millimeter galleries being about a millisecond is probably about right. Maybe in reality it’s a tenth of a millimeter or something because it sure takes a lot more than 10 milliseconds for hot water flowing through a 10-millimeter pipe to cool down to the temperature around the pipe, but maybe that’s mostly because the concrete around the pipe is less conductive than copper.

Copper has some advantages for this kind of boiler thing, although it’s not as strong as some other metals. It has great conductivity and tends not to corrode until well above boiling.

PV = nRT, and the molar mass of water is like 18 grams. At one atmosphere and 0° a mole should be 1 mol RT/P = 22.4 liters and 1/18 mol should be 1.25 liters. At 100° it's 1.70 liters, so 1 cc of water boils into 1.7 liters at that temperature, but it's not doing any work at that point, since it's at 0 gauge pressure. At 250° at 1 atm it's 2.38 liters.

Suppose the steam is pushing a piston 10 cm in a cylinder of 20 mm diameter, thus 314 mm². That's 31.4 milliliters of steam volume. For it to do 1000 J of work over that distance, it needs 10 kN, which means averaging 31 MPa, 314 atmospheres. This would need to be supercritical steam; water's critical point is about 22 MPa at 650 K (377°). At 250° its vapor pressure is only about 3 MPa, so it would only do about 100 J of work, which is still pretty okay.

At 250° at 3 MPa the ideal gas law gives us a volume of 80 ml for 1 g of water. PV/RT is about 390 mg, so if we have more water than this, some of it will remain liquid. Say 500 mg.

Heating 500 mg of water to 250° should cost about 0.5 g × 4.184 J/g/K × 230° = 480 J, though I guess the specific heat goes down a little at higher temperature, and then boiling it (normally 44 kJ/mol) only takes about 32 kJ/mol or 1.8 kJ/g, so about another 900 J, for a total of about 1.4 kJ. This is not a very efficient steam engine, under 10%.

If we want the 1.4 kJ to be lost in the sensible heat of some copper as it drops from 300° to 250°, well, copper's specific heat is 0.385 J/g/K (at room temperature anyway), so that's about 73 g of copper. This is an unreasonably large amount of copper to put in contact with 500 mg of water, so a better approach may be to maintain the temperature of the copper at 300° (to keep it from oxidizing) by running electricity through it as the water boils. Alternatively, we could use a smaller amount of copper at a higher temperature and just sacrifice it; copper boils at 2562°, and so we'd only need 1.6 g of copper at that temperature to boil the water, though at that temperature the specific heat might be significantly lower. And of course the engine won't work for many iterations.

Dumping 1.4 kJ into the copper electrically during the millisecond of boiling would require 1.4 MW. If we use 2000 V, above which we start to encounter special problems, we need 700 A, requiring 2.86 ohms or less to avoid needing even higher voltages. If we dump this in from a capacitor, the capacitor needs to have an ESR that's not too large compared to those 3 ohms.

This is pretty challenging. AVX's "BestCap" low-ESR supercaps include, say, the BZ01CB153Z_B, which handles 12 volts, 15 millifarads, and 420 milliohms (at 1 kHz, which is a good speed for this). This is 28 mm × 17 mm × 6 mm, more or less, and an energy capacity of about 1.1 J. You'd need 1300 such caps to hold the 1400 J, totaling 3.7 liters; a bit awkward.

It gets worse, though. That's a time constant of 6 milliseconds, so in 1 ms you can only get about 15% of the energy out of it. Other supercaps are similar. Aluminum electrolytics are faster but even bulkier.

No, resistance heating is not the way to go. The right way to flash-boil the water is with a packed bed of little balls of something

inert, like quartz or aluminum oxide or porcelain. You can heat it to the requisite temperature by blowing hot air over a Kanthal or Nichrome filament and then over the packed bed, then pump in the water once the packed bed is hot.

Granite's specific heat is 0.79 J/g/K, fused silica 0.703, crystalline quartz sand 0.835. Alumina (thermal conductivity 30 W/m/K, still far better than the water) is 0.96 J/g/K, I think, and doesn't melt until 2277°. At 1100° the 1400 kJ might need 1.3 grams. 1-mm balls of fused silica might withstand the thermal shock better than the stronger but more expansive aluminum oxide, though, even though alumina conducts heat better: granite's thermal conductivity is about 1.8–3.8 W/m/K, fused silica around 1.4 W/m/K, sapphire closer to 27. Embedding copper wires below the surface might help.

Such oxides could perhaps also improve the efficiency and compactness of the engine by withstanding higher temperatures without corroding. If one gram of steam is at 600° instead of 250°, then nRT/P at one atmosphere would be 4 liters instead of 2.38 liters; at 3 MPa it's 134 ml. If allowed to expand to only 31.4 ml, nRT/V gives us 12.8 MPa; if this pressure were constant throughout this expansion, because the steam is generated exactly as fast as it expands, it does 400 J of work. If the pressure is higher at first, because steam generation finishes earlier, it could do more.

I haven't calculated here the energy needed to heat the water and then steam to this temperature, but the 1.4 kJ above was to heat half this amount of water to 250° and then boil it off at that temperature, so probably it'd be around 3 kJ.

How do you power the resistance heater? Suppose you have four 18650s (weighing 250 g or so) and you use the 1800-mAh 15C types sold for quadcopters. Each can provide nominally 27 A at 3.7 V, which is 100 W. So all four together can provide 400 W, thus providing these 3 kJ of heat over 7.5 seconds. So they could activate this piston every few seconds.

Topics

- Materials (p. 1138) (59 notes)
- Contrivances (p. 1143) (45 notes)
- Electronics (p. 1145) (39 notes)
- Physics (p. 1157) (18 notes)
- Mechanical (p. 1159) (17 notes)
- Pulsed machinery (p. 1167) (12 notes)
- Thermodynamics (p. 1219) (5 notes)
- Regenerators (p. 1334) (2 notes)

Three phase logic

Kragen Javier Sitaker, 02021-04-30 (updated 02021-07-27)
(9 minutes)

I was thinking about building logic out of discrete transistors and other such basic parts.

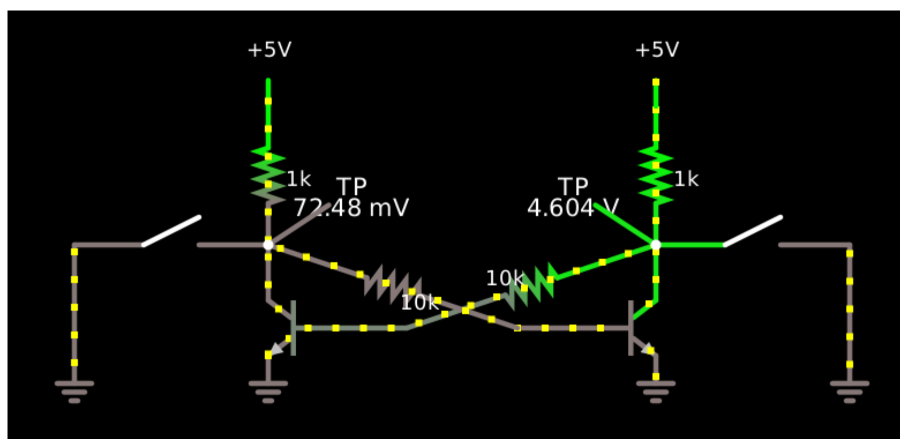
JLCPCB assembly pricing

This is in part driven by JiaLiChuang's PCB assembly service (see Can you get JLCPCB to fabricate a CPU for you affordably from "basic" parts? (p. 347)). They charge you I think 1.43¢ per PNP transistor, 1.32¢ per NPN transistor, 1.68¢ per N-MOSFET, and 1.79¢ per P-MOSFET, including assembly and soldering, but only 0.35¢ per 0402 resistor or 0.50¢ per 0805 resistor. So a standard CMOS two-input NAND built out of discrete power MOSFETs would be 6.94¢, while a standard RTL two-input NAND would be only 3.69¢, and also use a lot less space.

But can we do better? They only charge you 1.08¢ for a 1N4148W (again, including assembly). The LGP-30 CPU used 113 vacuum tubes and 1450 diodes; the tubes were set up as latches ("flip-flops") whose set and reset inputs were computed via diode logic. If the vacuum tubes could be replaced by MOSFETs, assembling these parts would cost US$17.56, of which US$1.90 would be the transistors.

The two-NPN RTL latch

Consider the traditional two-NPN RTL latch. Both NPN transistors have their emitters grounded, base resistors of 10kΩ to the collector of the other transistor, and collector pullups of 1kΩ to Vcc.




```
$ 1 0.000005 0.8031194996067259 50 5 43 5e-11
t 160 240 80 240 0 1 0.5676806399704408 0.6401601810026045 100 default
t 224 240 304 240 0 1 -4.531171328470282 0.07247958718181605 100 default
g 304 256 304 272 0 0
g 80 256 80 272 0 0
r 160 240 304 192 0 10000
r 224 240 80 192 0 10000
w 80 192 80 224 0
w 304 192 304 224 0
```

```
r 80 192 80 112 0 1000
r 304 192 304 112 0 1000
R 304 112 304 80 0 0 40 5 0 0 0.5
R 80 112 80 80 0 0 40 5 0 0 0.5
s 304 192 416 192 0 1 true
g 416 192 416 272 0 0
s -32 192 80 192 0 1 true
g -32 192 -32 272 0 0
368 80 192 128 160 0 0
368 304 192 256 160 0 0
o 16 64 0 4099 5 6.4 0 2 16 3
o 17 64 0 4099 5 6.4 0 2 17 3
```

Like RTL in general, this sinks current fairly strongly, but sources it fairly weakly. If $\beta = 100$ and Vcc = 5 V, the base current is (5 V - 600 mV) / 11kΩ = 400 μA, the transistor on the LOW side of the latch can sink 40 mA while remaining in saturation, but only sinks 5 mA in normal operation. You can flip the latch over by overwhelming it in either direction. By sourcing enough current into its LOW side you can pull it up to HIGH and knock the other side LOW, but this requires about 35 mA (a number which would push it up to 35 V if not restrained by the limited compliance of a current source). By contrast, sinking current from its HIGH side can flip it over with only about 3.9 mA, pushing the collector down by 3.9 V and leaving only 5 - 3.9 - .6 = 0.5 volts across the other transistor's base resistor, and thus only 50 μA of base current, at which point it desaturates and its Vce starts to soar, flipping the latch back.

This asymmetry is computationally promising: the latch can pull its inputs/outputs down to 0.2 volts sinking 40 mA, but pulling the other output down to 1.1 volts by sinking 3.9 mA is enough to flip the latch's state. There's more than a diode drop's worth of headroom in there and a fanout of about 10, and of course the latch itself provides amplification and inversion:

```
$ 1 0.000005 0.8031194996067259 50 5 43 5e-11
t 160 240 80 240 0 1 -4.531171328470283 0.07247958718181524 100 default
t 224 240 304 240 0 1 0.5676806399704415 0.6401601810026044 100 default
g 304 256 304 272 0 0
g 80 256 80 272 0 0
r 160 240 304 192 0 10000
r 224 240 80 192 0 10000
w 80 192 80 224 0
w 304 192 304 224 0
r 80 192 80 112 0 1000
r 304 192 304 112 0 1000
R 304 112 304 80 0 0 40 5 0 0 0.5
R 80 112 80 80 0 0 40 5 0 0 0.5
s 304 192 416 192 0 1 true
g 416 192 416 272 0 0
s -32 192 80 192 0 1 true
g -32 192 -32 272 0 0
368 80 192 128 160 0 0
368 304 192 256 160 0 0
368 800 192 752 160 0 0
```

```
368 576 192 624 160 0 0
g 464 192 464 272 0 0
s 464 192 576 192 0 1 true
g 912 192 912 272 0 0
s 800 192 912 192 0 1 true
R 576 112 576 80 0 0 40 5 0 0 0.5
R 800 112 800 80 0 0 40 5 0 0 0.5
r 800 192 800 112 0 1000
r 576 192 576 112 0 1000
w 800 192 800 224 0
w 576 192 576 224 0
r 720 240 576 192 0 10000
r 656 240 800 192 0 10000
g 576 256 576 272 0 0
g 800 256 800 272 0 0
t 720 240 800 240 0 1 -4.53117132847028 0.07247958718181717 100 default
t 656 240 576 240 0 1 0.56768063997044 0.6401601810026043 100 default
w 304 192 320 208 0
w 544 208 576 192 0
d 544 208 320 208 2 default
s 80 192 16 128 0 1 true
R 16 128 16 96 0 0 40 5 0 0 0.5
o 16 64 0 4099 5 6.4 0 2 16 3
o 17 64 0 4099 5 6.4 0 2 17 3
o 19 64 0 4099 5 6.4 0 2 19 3
o 18 64 0 4099 5 6.4 0 2 18 3
```

So by coupling some such latches together with diodes you can compute, among other things, arbitrary logic functions; by releasing a /S signal at some given time you can see whether any of the various sources connected through diodes to its /R input were low at the time, thus computing their AND (in conventional positive logic) and putting their NAND on the /S line at that time.

(The same asymmetry also means you can get 3-stable and 4-stable latches rather than just bistable ones.)

The three-phase thing

Something I think is more interesting is what happens when you power the latch from a clock signal instead of a constant positive voltage rail. Say the clock signal controls some kind of high-side switch like a PNP transistor; then when the switch is turned off both inputs are basically just 10kΩ to ground (+0.6V at currents above the most minimal) and 2kΩ to each other. The same switch can be shared between probably multiple gates.

If you divide your gates into three phases, you can activate each phase during a different three sixths of a single complete clock cyle:

```
CK0  CK1  CK2
 *
 *    *
      *
      *    *
           *
```

```
 *         *
```

In this way, gates in phase 1 can compute results from gates in phase 0, gates in phase 2 can compute results from gates in phase 1, and gates in phase 0 can compute results from gates in phase 2.

But how do you ensure a determinate outcome? As before you could release a /R input at a given point during the clock cycle, lengthening the full set of stages within the clock cycle from 6 to 9:

```
E0 R0 E1 R1 E2 R2
** **
**       **
**    ** **
**    **
      **       **
      **    ** **
      **    **
   **       **
** **       **
```

Topics

- Electronics (p. 1145) (39 notes)
- Pricing (p. 1147) (35 notes)
- Manufacturing (p. 1151) (29 notes)
- Falstad's circuit simulator (p. 1198) (7 notes)
- Physical computation (p. 1208) (6 notes)
- Oscillators (p. 1283) (3 notes)
- JLCPCB (JiaLiChuang) (p. 1360) (2 notes)

How fast do von Neumann probes need to reproduce to colonize space in our lifetimes?

Kragen Javier Sitaker, 02021-05-04 (updated 02021-06-12)
(5 minutes)

Suppose we want 16 earth-surfaces' worth of human living space by the year 02050, in orbit around the sun, using an exponentially growing colony of 3-D printers that starts growing in 02029.

How much material is this? The classic approach is rotating space stations in order to provide artificial gravity of one gee. Suppose we don't have a stronger material for this than music wire, whose free breaking length is 35 km (2700 MPa / (7.9 g/cc) / gravity). (We can hope that carborundum fiber (110 km), basalt fiber (183 km), monocrystalline iron whiskers (183 km), UHMWPE (378 km), carbon fiber (399 km), nanotube rope (4700 km), etc., are options, but maybe they won't work out.) We can't use *all* the structure's weight in tensile supports against its own centrifugal force, since we also want lakes and soil and stuff; suppose we use an "overhead factor" of 3.

(Ugh, I'm tangling up in mental knots trying to figure this out.)

So, uh, suppose our habitat soil and lakes etc. is about 10 meters deep, so we need about 20 tonnes of soil per square meter, plus another similar amount of our bargain-basement Victorian-era scrith, let's say 64 tonnes per square meter in total. 149 million km², the Earth's land surface, is then 9.5×10^{18} kg (9.5 zettagrams). 16 times that is 153 zettagrams, round it up to 200 zettagrams (2×10^{20} kg), about 0.3% of the mass of the Moon, all of the mass of asteroid Pallas, or 7% of the mass of the main asteroid belt.

If the initial 3-D printer weighs 100 g in 02029, that's a factor of 2×10^{21} growth in 21 years, which is fairly slow by the standards of bacteria and fungi, about 0.64% per day, doubling every 108 days. A human growing from 4 kg to 40 kg in 10 years is on average much slower, of course, but during their first 3 months after birth they grow perhaps from 3.5 kg to 6.5 kg, nearly the same speed.

At this speed, we have the following growth curve:

02029	02030	02031	02032	02033	02034	02035	02036	02037	02038	02039
100 g	1 kg	11 kg	108 kg	1.1 t	11 t	120 t	1.2 Gg	12 Gg	128 Gg	1.3 Tg
Gerbil	Rabbit	Pug	Dolphin	Narwhal	Three hippopotami	Blue whale	General Sherman	2×Pando	La Tour CN	Golden Gate Bridge

02040	02041	02042	02043	02044	02045	02046	02047	02048	02049	02050
13 Tg	138 Tg	1.4 Pg	15 Pg	150 Pg	1.6 Eg	16 Eg	160 Eg	1.7 Zg	17 Zg	**180 Zg**
2×Great Pyramid	the humans	the fish	???	Lake Tahoe	Lake Ontario	Lake Tanganyika	Gulf of California	Gulf of Mexico	Arctic Ocean, or asteroid Euphrosyne	Indian Ocean, or asteroid

As Machiavelli points out, any innovation is likely to provoke opposition from entrenched interests. Experience in 02020 has shown that existing human institutions are not equipped to stop or respond to exponential phenomena with doubling times of around a week, so this is probably a better benchmark to shoot for, although of course in this case the phenomenon is a liberatory phenomenon of human empowerment rather than an epidemic of a virus.

Having the seed on Earth is not equivalent to having it in orbit around the Sun; launching things into orbit is expensive and tightly surveilled. There are two likely ways to cross this bridge.

The first is to get a single spore off Earth as early as possible and grow it among the asteroids, which has the disadvantage that it requires access to space launch very early on, before a significant quantity of printers have been built. Worse, it's not just space launch but actually escape velocity, which means launching not just a printer but a thruster that can provide the other half of the Δv. this would benefit from having as small a printer as possible, especially absent buy-in from the entrenched interests mentioned above; the Space Surveillance Network publicly catalogs 17480 objects mostly 10cm (1ℓ) or larger, so we'd have to assume objects down to 3cm are usually detectable even today, especially if they're in LEO doing things like emitting plumes of plasma. So the target design size would probably have to be about 10g or less: the future of a galaxy contained in the mass of a coin.

The other possibility is to grow the 3-D printer ecosystem here on Earth until it can print abundant space launch resources, perhaps in 02040 or thereabouts. This avoids the risk of provoking opposition at an earlier, more fragile stage of the project, but it means that inevitably it will have to face opposition *before* making the leap off Earth, and consequently is at serious risk of being strangled in the cradle. It also limits the material and energy resources that will be available to the project during that time to what is available on Earth. However, everything done on Earth is enormously easier in some ways — to fix when it breaks, especially — and there are many resources on Earth that are hard to find elsewhere, such as microprocessors.

Topics

- Digital fabrication (p. 1149) (31 notes)
- Self replication (p. 1204) (6 notes)
- The future (p. 1220) (5 notes)
- Space (p. 1323) (2 notes)

List of random GUI ideas

Kragen Javier Sitaker, 02021-05-04 (updated 02021-07-27)
(6 minutes)

Some random ideas to try in UIs:

• How about crosshairs instead of a mouse pointer? Boundaries between gradients, maybe, rather than opaque black lines. Your mouse pointer shouldn't obscure anything, and being able to use it to see the alignment between things is occasionally useful.
• Or maybe crosshairs with a vertical line that only goes down?
• what if the crosshairs are done with some kind of filtering that is subtle enough that you barely notice it when it's not moving. like an edge enhancement kind of filter say.
• circles are a common alternative for indicating a point in some other contexts, like screencasting software and touch ui
• Maybe for 3-D rendering a bit of Lambertian surface bumpmapping would help with adding the illusion of 3-D-ness? If the bumpmapping is bandlimited well below the pixel frequency then visual discontinuities will coincide with depth discontinuities. Perlin noise maybe? Simplex noise? Blue noise? Ambient occlusion might help add depth cues too.
• You can render a sphere as a circle very quickly, but it looks like a disc. If you add an elliptical contour between light and dark, though, you can add a lot of spherishness very quickly. Spheres have some merits compared to triangles as 3-D primitives.
• Directly highlighting depth discontinuities with edge features (black lines, white lines, Gabors, sincs, whatever) might also improve rendering legibility.
• This NeRF stuff produces really impressive 3-D renderings. Can some kind of sparse representation of a radiance field be useful for user interfaces too?
• What does a UI for guiding heuristic A*-like search through an exponentially large search space look like?
• when exploring a search space covering orders of magnitude through manual dimensional search, maybe you can have buttons for ½×, 2×, 10×, .1×, etc., each accompanied by a preview of the variables of interest there. iphone software often displays the previews as the background of the buttons, also handy for things like choosing color palettes.
• what would a ui scripting language look like. like, something for easy exploration of different ui dynamics, like a simple game scripting language for ui components. how can we minimize the time between coming up with a ui idea and trying it out on yourself.
• low-ui-budget software i see screencasts of on youtube (optistruct, notepad plus plus, abaqus, matlab, origamizer, etc.) is mostly a fairly small number of ui components that look straight out of win95. pulldown menus, dockable buttonbars, radio buttons with black dots inside white circles, checkboxes, gray backgrounds everywhere, accelerators, buttons, tab panels, a status line along the bottom of the window, dropdowns, dialog boxes, text fields, scrollbars, occasionally a slider or spinbutton. one thing that's surprisingly common is a tree

control in the left pane. i suspect that a lot of this is stuff that people find very familiar by now and so departing from it should be done with care. even tinkercad has a fair bit of this feeling even though it's in the browser. catia definitely has its own look but it still has win95-style comboboxes with the little downward-pointing isosceles right triangle on a gray embossed button and shitty little toolbars around the viewport and shitty opaque gray dialog boxes full of forms.

- one way that a lot of this software is kind of nice is that they often display what could be tooltips in the status bar.
- meshmixer is a little nicer looking by virtue of, among other things, translucency, fewer larger buttons with labels, and more visual texture and gradients. ambient occlusion just about shows up in its clicky gui. prusaslicer also uses translucent overlays (with sparklines even) in the corners of the main viewport to good effect, and does the smae kind of thing with some dialog boxes. fusion 360 of course overlays its tree view on the left side of the viewport with cracks but not much translucency.
- iphone software often overlays text labels for buttons in a popup menu over the main viewport, so the (often translucent) buttons on the menu minimally obscure the main viewport. though this kind of text overlaying thing wouldn't be very useful if what's in the main viewport is actually text, in which case reflowing it to miss the menu would be more useful.
- also typically instead of radio buttons iphone software has a slidy switch thing with a circle that can slide to different positions in the radio button group.
- a common fui trope is to put statusbarry things in folder-tab-like things with 45° angles protruding onto the main viewport. also of course in fui everything is transparent, blue, and circular.
- slight delays between animating multiple items are common in modern chi prototypes and break up the appearance of solidity, reinforcing the multiplicity of objects
- graying out the background when a pie menu or similar modal pops up is really helpful for focusing attention
- integrating high-res touch surfaces into projected or glasses-displayed ar environments can provide higher-resolution and lower-latency interaction possibilities, though xiao's work has shown that common touch surfaces can't really do any better than depth cameras (5mm or so); you need better-quality touch
- just as sound is the lowest-latency interaction modality, vision is the highest-bandwidth one, and that's underused at present, in part because people aren't chameleons or octopodes. but they can draw. for some reason gesture tracking via camera analysis is still janky and jerky.

Topics

- Human-computer interaction (p. 1156) (22 notes)
- GUIs (p. 1216) (6 notes)

Thixotropic electrodeposition

Kragen Javier Sitaker, 02021-05-04 (updated 02021-12-31)
(2 minutes)

There are various ways to deposit thixotropic pastes in a more or less controlled way; you can squeeze them out of a nozzle, RepRap-style, or form them with a spatula like cake icing, or stamp them between dies, or imprint their surface with dies such as Sumerian rollable clay seals, or cut them with blades. These techniques have been central to human manufacturing since the Neolithic advent of pottery; clay bodies have the desirable trait of having an extremely small elastic deformation limit beyond which they deform plastically, permitting fairly precise dimensional control of the result, though this is often compromised by the grain size of tempers in the clay body and by contraction on drying.

Often dimensional precision is compromised by adhesion to the tools being used, but a spritz of liquid mold release or coating of solid powder is frequently sufficient to keep this source of error manageable. In other cases, a layer of rapidly flowing gas or liquid, as in an air bearing or air-hockey table, may eliminate this problem.

It occurred to me today as I was watching peaks of dulce de leche stubbornly fail to collapse that *emulsions* like mayonnaise and dulce de leche are generally thixotropic, as are liquid-gas foams. I guess the energy to deform droplets of the discontinuous phase out of sphericity manifests as enough elasticity to render the overall metamaterial thixotropic.

[Later I read some papers about dulce de leche rheology, and that’s not why it’s thixotropic. It's thixotropic because the sugars and polysaccharides form a gel.]

If the thixotropic paste is sufficiently dimensionally stable, this is the easiest way to realize arbitrary high-precision three-dimensional geometry. Often, though, the thixotropic substance itself doesn’t have the desired properties; mayonnaise, for example, noticeably lacks structural strength.

[Note that the above uses the wrong definition of thixotropy: I thought thixotropy was the phenomenon where viscosity increases instantaneously at low shear rates, but actually thixotropy is where viscosity increases progressively over time at low shear rates, which is a phenomenon that does happen with dulce de leche.]

Topics

- Materials (p. 1138) (59 notes)
- Digital fabrication (p. 1149) (31 notes)
- Clay (p. 1179) (10 notes)
- Ceramic (p. 1193) (8 notes)
- Thixotropy (p. 1317) (2 notes)

Cheap cutting jig

Kragen Javier Sitaker, 02021-05-06 (updated 02021-12-30) (1 minute)

Freehand plasma or oxy-acetylene cutting frequently wanders pretty far offcourse, but plasma cutting along a straightedge is a lot more precise. If you want a complicated shape cut out of metal sheet, rather than a straight line, you can cut out some kind of pattern instead of using a straightedge. For plasma cutting you may be able to use laser-cut MDF, but oxy probably needs something that can resist the heat.

Dried mud is quite possibly an adequate material for this, but if not, surely fired-clay ceramic is. You can roll the wet clay body into a plastic sheet, let it dry to leather-hardness to get most of the shrink out, then cut out the desired form with a knife or perhaps a wire. Alternatively, a sheet of gypsum can be either cut to shape or deposited as paste.

How do you know where to cut? You might be able to use a knife plotter like those used to cut vinyl, but a probably-lower-budget alternative is to project the shape to cut onto the surface with a laser, then cut by hand. For small projects, you can print out a plan on a laser or, better, inkjet printer, and glue it onto the pattern before hand-cutting.

Topics

- Digital fabrication (p. 1149) (31 notes)
- Ghettobotics (p. 1169) (12 notes)
- Clay (p. 1179) (10 notes)
- Ceramic (p. 1193) (8 notes)
- 2-D cutting (p. 1201) (7 notes)

Differential filming

Kragen Javier Sitaker, 02021-05-07 (updated 02021-12-30) (1 minute)

Illuminating an area with a somewhat bright flash produces more brightness on objects closer to the flash. By arithmetically subtracting a non-flash-illuminated frame from a flash-illuminated frame, you can extrapolate what the scene would look like if the flash were the only illumination. This could be useful to provide illumination for actors, etc., that isn't visible in a photograph or motion picture. This probably isn't useful for confidentiality, since brighter areas of the subtracted background will have larger amounts of noise in the subtraction result, but it could provide dramatic lighting effects, even permitting after-the-fact lighting adjustment.

Topics

- Algorithms (p. 1163) (14 notes)
- Sensors (p. 1191) (8 notes)
- Cameras (p. 1301) (3 notes)

Planetary roller screw worm drive

Kragen Javier Sitaker, 02021-05-07 (updated 02021-12-30)
(4 minutes)

One of the problems with worm drives is that they tend to be a little inefficient, which limits their maximum power because they heat up. This often prevents them from being backdriven. This is due to the sliding friction between the worm and the worm wheel, which becomes a bigger problem for larger reduction ratios because of the smaller leads.

Suppose you have an "inverse planetary roller screw", a planetary roller screw where a cage of screws revolves around a central screw but is not free to move relative to the central screw, but instead of driving longitudinal displacement of an outer "nut", it drives the rotation of a worm wheel. This eliminates sliding friction, and when the planetary screws are of the opposite handedness from the sun screw, also amounts to a differential gear drive, permitting very high reduction ratios. This worm wheel can also have much lower backlash than the traditional sliding-contact type.

The screws can to some extent be "throated", with the planets being hourglass-shaped in order to maintain contact with the wheel around some of its circumference instead of just a point, requiring the sun to bulge out into a cocklebur shape to compensate. This variation of radius reintroduces some sliding friction due to the varying radii, but I think it will still be much less than an ordinary worm drive, particularly if the worm is large compared to the worm wheel. The worm wheel can likewise be throated to maintain contact with the planets through more than a single point in their travel.

To maintain large areas of continuous contact it is desirable for the planets to be very small compared to their orbit size so that there can be very many of them, but this eliminates the possibility of extreme reduction ratios through the roller-screw mechanism itself. Extreme throating of the worm wheel, with each of its teeth encompassing a near-semicircle of the planets, might nevertheless allow such large ratios.

Another variant that would reduce the variability of contact from five or six rollers is to split the cage into two or more sections, each of which has separately rotating rollers, which are staggered. For example, one half of the cage might have roller screws around the center screw at 0°, 60°, 120°, 180°, 240°, and 300°, while the other half has them at 30°, 90°, 150°, 210°, 270°, and 330°. These roller screws would be half as long as they would be otherwise, so this doesn't increase the total amount of area of contact, but it does potentially distribute it better, producing smaller distortions.

The unbalanced forces on the shaft from pressing against the worm wheel can be balanced on the shaft in different ways. An additional identical worm wheel on the other side is one possibility; the shaft can held in heavy bearings on one or both sides of the worm drive; or the "top lands" or "threadform truncations" or "crests" on the tops of the planetary screw threads can roll against a simple cylinder on the

opposite side from the worm wheel, or an hourglass shape if the planets are throated.

Similarly, the radial side load on the worm wheel can be balanced by placing one or two other worms on the other side. If only balancing the radial force is desired, these additional worms can either be idlers or driven synchronously, but they can also serve as additional driven elements, either for sensing or to provide mechanical power.

Another way to use these differential roller screws is in place of a micrometer screw, whether to measure the sizes or angles of things or to actuate to precise distances or angles with very low backlash, though of course you need to minimize thermal distortion and possibly compensate for it before you get too far.

Topics

- Contrivances (p. 1143) (45 notes)
- Mechanical (p. 1159) (17 notes)
- Roller screws (p. 1275) (3 notes)
- Gears (p. 1365) (2 notes)

Fresnel mirror electropolishing

Kragen Javier Sitaker, 02021-05-08 (updated 02021-12-30)
(6 minutes)

Suppose you want to reflect light of a given wavelength (say, 555 nm) coming from a given direction into given directions from all points of a surface. It is sufficient to be able to determine the phase delay at all points of the surface; the gradient of that phase delay with respect to the u-v coordinates on the surface then determines the output beam direction.

If you can only control the phase delay up to some limit, then occasionally you will have a discontinuity, like in a Fresnel lens. This is minimally disruptive if the discontinuity jumps an integer number of wavelengths, such as 1. So it's sufficient to be able to control the phase delay over a single cycle, for example by etching the surface selectively to depths of up to half a wavelength, since a half-wavelength-deep pit will induce a whole wavelength of phase delay for normal light. Less depth is needed if the light is at an oblique angle.

277 nm is a pretty shallow etching depth, and the average depth is only half of that, 139 nm. If you want to achieve it purely via electropolishing, you're removing 139 picoliters per square millimeter. But if you can use a combination of electrodeposition and electropolishing, the average amount of material moved is only half of *that*, 69 nm or 69 pl/mm². If the material is copper, which weighs 8.89 g/cc, that's 610 nanograms per square millimeter. Copper is 63.546 g/mol, so that's 9.7 nanomol per square millimeter, which works out to 5.8×10^{15} atoms per square millimeter added and removed.

Electropolishing copper involves removing two electrons per atom, and electrodepositing it involves adding them, and the electron charge is about 1.6×10^{-19} coulombs, so that's about 1.9 millicoulombs per square millimeter.

So, if your feedback and control systems were up to the task, with 100 mA you could electroform/electropolish 54 mm² per second; that's probably about half a watt. For 22 A4 pages per minute, you'd need about 43 amps, which is a lot better than a regular laser printer.

If you were anodizing aluminum in water instead of electropolishing copper, you'd need to do the whole 139 nm depth since you can't electrodeposit aluminum in water, and you'd need to remove three electrons per atom instead of two. Aluminum is only 2.70 g/cc, while its atomic weight is only 26.9815384(3), so a mole of aluminum is 10 cc to copper's 7.14. So you actually need roughly the same amount of current per volume of aluminum as you do for copper.

Actually though the anodization layer you're depositing has a high refractive index; according to a couple of different papers, ranging from 2.1 at 2 volts or 10 mA/cm² down to 1.6 at 10 volts or 100 mA/cm². Either way the index gets even higher for blue light. This means your wavelength is 1.6 to 2.5 times shorter than the vacuum

wavelength, so you actually need about the same amount of current per area despite the inability to electrodeposit aluminum.

(This variation in ior with applied current suggests that fabrication of rugate filters in nanoporous aluminum oxide by applying a time-varying current may be feasible.)

Actually, it's even better than that; the above is calculated assuming the etched space is filled with this ior-1.8-or-whatever coating, but in fact typically the anodized coating on aluminum is twice as thick as the aluminum thus consumed. So each 10 nm of aluminum you anodize into oxide produces 20 nm of oxide, which the light will strike 10 nm earlier than for the untouched surface and leave 10 nm later. So the total phase delay is, say, 1.8 × (20 nm + 20 nm) - 20 nm, which ends up being 52 nm of phase delay.

At 30 mA/cm² our 2 mC/mm² or so would take a few seconds. Like, 7 seconds. So this seems like an eminently feasible process to carry out at a physical level; the only question is how to do the process control, using viscosity, positional control, current pulses, and perhaps optical feedback.

Even a 10-micron-thick layer of aluminum foil would be more than sufficient to electro-etch 80 nm deep into. You could imagine doing 10 or 100 wavelengths or more of phase delay instead of just one, thus allowing your mirror to function across a wide range of wavelengths and reducing the number of discontinuities and their associated stray-light losses. (Ordinary non-selective hard anodization coatings on aluminum do already indeed reach 50 microns routinely and 100 microns occasionally.) This will enable rapid computational production of white-light holograms, diffraction gratings, imaging mirrors, and solar concentrators.

Despite the inevitable micro-cracks, the shapes of anodizing coating thus thrust up from the aluminum surface may also be usable for non-optical, mechanical purposes such as stamping, or as mechanically mating surfaces that slide past one another.

Anodizing titanium rather than aluminum offers the possibility of stronger refraction and and thus thinner films and faster production, as well as of course the direct use of the stronger iridescence that comes from rutile's higher ior.

I think electropolishing of aluminum in chloride electrolytes such as sodium chloride produces water-soluble aluminum chloride rather than insoluble aluminum hydroxide, although some production of the gel is also reported; one experimenter reports 3.6 mm per minute drilling (60 microns/second) with 5–14% sodium chloride and a 0.1–0.4 mm interelectrode gap. By comparison, EMAG's "PECM" die-sinking ECM machines' oscillating process gap typically goes down below 50 microns; they imply it oscillates at 50 Hz and goes higher than 750 microns to improve sludge clearance, flushing at 10–50 m/s.

Topics

• Digital fabrication (p. 1149) (31 notes)

- Electrolysis (p. 1158) (18 notes)
- Machining (p. 1165) (13 notes)
- Aluminum (p. 1180) (10 notes)
- ECM (p. 1186) (9 notes)
- Optics (p. 1209) (6 notes)
- Small things (p. 1223) (5 notes)
- Electropolishing (p. 1371) (2 notes)

Leaf hypertext

Kragen Javier Sitaker, 02021-05-08 (updated 02021-12-30)
(3 minutes)

I've previously written a bit about a hypothetical new hypertext medium displaying several "cards" or "scraps" at once. I think "leaf" is maybe better terminology.

The idea is that your memex consists of some collection of "leaves" and dynamically lays out as many as will fit on the screen, something like TiddlyWiki or Ward's Simplest Federated Wiki; but the leaves are a bit smaller, like a line of text rather than a paragraph. Each leaf is identified by a globally unique leaf ID which can be used to link to it, and those links may be activated explicitly (for example by clicking) or implicitly in order to display some other leaf. Typically these implicit links include previous and next links, which permit reading through a linear document in a more or less straightforward fashion by shifting the focus to earlier or later leaves --- if necessary by clicking or tapping them, but usually just by centering them in the display. (This implies that the potentially infinite graph of implicit links cannot simply be fully traversed to decide what to display at any given time.)

In addition to this subatomic conception of hypertext, we have *parameters* in the links, like #fragment identifiers on the WWW. The rendering of a leaf for display is done by some arbitrary Turing-complete code, contained in the leaf or linked from it, which is supplied some arbitrary blob of parameters from the link that was followed to display it. By interacting with the display (clicking on it, typing characters into it, etc.) you can change these parameters under the control of that code.

In this way, it becomes straightforward to write a leaf that, for example, contains data and buttons to plot the data in different ways by invoking other leaves with the data as parameters, or that invokes an "infobox template" leaf with some parameters in order to present that data visually in a consistent way.

I'd like to do some kind of infinite regress thing where the leaf contents are themselves produced by invoking some piece of code with some parameters, and the leaf ID likewise consists of the ID of that piece of code and its parameters, so that ultimately we ground ourselves in a single "primitive leaf" that provides some kind of fundamental virtual machine. But I don't quite see how to do that yet.

In the stuff you have on the screen you can thus have a mix of stuff you wrote, stuff you're taking notes on, and code.

Topics

- Human-computer interaction (p. 1156) (22 notes)
- Virtual machines (p. 1182) (9 notes)
- Higher order programming (p. 1196) (7 notes)

- GUIs (p. 1216) (6 notes)
- End user programming (p. 1217) (6 notes)
- Caching (p. 1266) (4 notes)
- Reproducibility (p. 1277) (3 notes)
- Hypertext (p. 1291) (3 notes)
- Wiki (p. 1311) (2 notes)
- Archival (p. 1389) (2 notes)

Weighing an eyelash on an improvised Kibble balance

Kragen Javier Sitaker, 02021-05-08 (updated 02021-12-30)
(3 minutes)

Watched a Nominal Semidestructor advertising video recently about weighing an eyebrow hair with an analog meter movement. Paul Groke built his scale as follows: he oriented the needle horizontally, connected up an optointerruptor to an opamp that drove the current through the meter movement so as to maintain the optointerruptor half interupted, then dropped his eyelash on the end of the needle. Then the current required to restore the zero position gave a measurement of the weight. Result: 75 micrograms or something. I guess he calibrated it previously.

(The circuit is very simple: the opamp has a resistor from its output to its inverting input, which is connected to the photodiode, and its noninverting input is held at a fixed bias voltage. Then the output is just connected to the meter movement, which is then connected to a resistor to ground. So the current flowing through the resistor keeps the inverting input equal to the bias voltage, and that current is just the photocurrent, so the output is maintained at a voltage above the bias voltage proportional to the photocurrent by the factor of the resistance.)

The nice thing about this kind of Kibble balance is that, except for elastic deformation, the movement is in the same position when it's balanced as when it's empty. The magnetic field can be wildly nonuniform over the range of the meter's movement, but you don't care because once you're taking a reading you're back at the same position; all that's changed is the strength of the magnetic field (which is hopefully linear in the current) and the bending of the balance beam.

(This version of the ampere balance used to be called a "watt balance" because the power needed to restore the position is what's proportional to the weight. Kibble proposed it in 01975, and after he died in 02019 they renamed the instrument after him. The full-fledged Kibble balance includes some other refinements and can measure mass by reference to voltage, current, gravity, and speed.)

It occurred to me that a simpler and higher-resolution version of the instrument would use electrical contacts instead of an optointerruptor, ideally gold-plated spherical ball bearings to ensure a well-defined point of contact. Then you include a small ac or noise component in the coil current in order to get intermittent contact and thus a continuously varying feedback signal ("stochastic resonance"). Such electrical contacts can detect movements of nanometers rather than the microns you get from an optointerruptor. At this point thermal deformation and creep of the balance apparatus become more significant sources of error than the sensor.

Topics

- Contrivances (p. 1143) (45 notes)
- Ghettobotics (p. 1169) (12 notes)
- Bootstrapping (p. 1171) (12 notes)
- Precision (p. 1183) (9 notes)
- Metrology (p. 1212) (6 notes)
- Small things (p. 1223) (5 notes)
- Weighing (p. 1267) (3 notes)
- Stochastic resonance
- Kibble balances

Precisely measuring out particulates with a trickler

Kragen Javier Sitaker, 02021-05-09 (updated 02021-12-30) (17 minutes)

One of the problems I ran into with flux-deposition particle-bed 3-D printing was depositing small consistent amounts of flux particles. Fine particles clump and stick, which means dry particles don't deposit uniformly.

I learned today of a device called a "trickler" used for depositing small amounts of particulate (milligrams, not micrograms, at a rate of a few milligrams per second). It's a near-horizontal cylindrical tube that can rotate around its axis with some particles in it. As it rotates, the particles roll around in it, which helps break up clumps, and some fall out the end of the tube; some versions have a screw thread on the inside of the tube to push the particles along. There's a constant feed of new particles into the other end of the tube, achieved for example by immersing it in a bin of particles and having one or more holes in its side. If you stop rotating the tube, the particles stop falling out, because of friction with the floor of the tube.

It seems to me that this approach is likely to work well for solid particle flux deposition, although the resolution may be a bit coarse. If we're shooting for 100-micron "pixels", well, that's about two micrograms, and the idea is that only about 5% of the total particle bed is flux, so 10 nanograms per "pixel", and we'd like that 10 ng to be about 10 particles to reduce the random variation. (Weighing or filming the particulate bed during the operation may enable the counting of individual flux particles.)

1 ng is about 500 cubic microns; your particle diameter then needs to be on the order of 10 microns, which is about four or five linear orders of magnitude smaller than what the "tricklers" normally manage. Such fine particles tend to clump pretty aggressively.

Liquids

Of course, the standard solution to particulate-bed printing is to deposit "binder" via jets of liquid "ink" using a standard inkjet printer mechanism. I'd been thinking that this probably wasn't a viable option for flux deposition, because the fluxes usually aren't water-soluble, but now I think it's possible in many cases to use water-soluble forms of fluxes.

For fluxing quartz, for example, the highly water-soluble hydroxides, bicarbonates, acetates, or formates of sodium (soluble to 109, 9.6, 46.4, and 81.2 g/100ml of water at 20°) or potassium (112, 33.7, 256, and 337) would probably work; heating any of these will eventually leave only the oxide. Soluble salts of calcium such as the chloride (74.5), formate (16.6), acetate (34.7), or nitrate (121.2) may be helpful in addition, as they further lower the melting point of the quartz while reducing the water-solubility of the final product.

Things that decompose into lead oxide might be superior to the

above for fluxing quartz. Lead nitrate (54.3) is suitable; its acetate (44.3), formate (16), chlorate (144), and perchlorate (440) might be, too.

Boric acid is reasonably water-soluble (4.7 g/100ml) and liquefies at only 170.9°, at which point it can dissolve or react with a fair number of other things, most especially including quartz and other silicates. The whole sequence is somewhat complex: at 170.9° it becomes metaboric acid, which melts at 176° and converts to B_2O_3 at around 300°, which can crystallize into forms that melt at 450° or 510° but is more commonly amorphous.

Many other metals have highly soluble chlorates and perchlorates, although potassium's are only mildly soluble. These could be especially useful in contexts where mixing with potassium salts is not necessary, but oxidation is either harmless or desired. These decompose to produce oxygen and chloride when heated above 400°. This kind of phenomenon may be useful for providing heat for firing the printed part.

There are a number of highly water-soluble phosphate salts (TSP is 12.1 g/100ml, MSP is 59.9, DSP is 11.8, STPP 14.5, STMP 22, K_3PO_4 90, MKP (KH_2PO_4) 22.6, MAP 36, DAP a bit over 57.5, TAP 58) which can contribute phosphate ions; in particular the ammonium phosphates decompose to phosphoric acid and ammonia gas at moderate temperatures around 200°. Phosphate ions can react with polyvalent cations to form extremely stable materials like calcium phosphate (apatite, brushite, whitlockite, bone, hilgenstockite), aluminum phosphate (berlinite, augelite, variscite), and zinc phosphate (hopeite, parahopeite, tarbuttite, and dental cement). So you could imagine selectively stabilizing some calcium-bearing or zinc-bearing material by squirting phosphate salts on it, heating them to cause a reaction (which might bind particles together), then removing the untreated particles.

Alumina is attacked by hydrochloric acid to become aluminum chloride, which is not only highly soluble in water but also sublimes at 180°. Generalized chlorates (including perchlorates, chlorites, and hypochlorites such as that of calcium) may be suitable donors of chlorine for such a reaction, but even ordinary chlorides like those of sodium or calcium might work at a high enough temperature.

Soluble fluxes for 3-D printing metal alloys

For fluxing iron particles, carbohydrates like sugar would probably work well; they can dehydrate, leaving only carbon, long before the iron starts to absorb hydrogen, although oxidation of the iron with the resulting water molecules may be a concern. Paraffins contain no oxygen and aren't water-soluble, but other solvents may work to make paraffin inkjets, and they too will carbonize around 300°, long before the iron takes up their hydrogen; but anything but very-long-chain paraffins (polyethylene) will boil off pretty easily. So perhaps something like linseed oil, which is largely a triglyceride made of α-linolenic acid and linoleic acid, would work better, with nonzero oxygen content but much lower thermal stability. Turpentine is its traditional solvent.

Among low-melting metal alloys, my attention is drawn by 5% tin

or aluminum added to zinc depressing its melting point from 419.53° to 382°; 2.5% silver added to lead depressing its melting point from 327.46° to 304°; 9% zinc added to tin (KappAloy9) depressing its melting point from 231.93° to 199°; 3% silver added to indium depressing its melting point from 156.5985° to 143°; and 5.5% zinc, 4.5% indium, and 3.5% bismuth added to tin depressing its solidus from 231.93° to 174°. Also, in general the tin-lead alloys have a solidus of 183°, the melting point of the eutectic.

Popular higher-melting metal alloys include brass, bronze, aluminum bronze, and arsenical bronze. Zinc-copper brass doesn't really have a eutectic; its lowest-melting version is 100% zinc (419.53°) and at low zinc content there's a relatively slow drop of the melting point from copper's 1084.62°. The copper-tin system (ordinary bronze) is broadly similar but somewhat luckier, with the solidus depressing to about 900° with 10% tin (if I'm reading this phase diagram right). Aluminum bronze has a real eutectic at 548.2° and about 32% copper, with a fairly steep drop in aluminum's solidus from 660.32° down to that temperature with only about 6% copper, as well as a eutectoid on the other side around 92% copper and about 1050°. So you could imagine using something near 6% copper as a "sintering aid" for aluminum particles. The arsenic-copper system is similar, with an apparent eutectic at 685° and 20.8% As, but already depressing the solidus to that temperature at only 7.96% As. (Wait, that doesn't make sense, that's higher than aluminum's melting point.)

However, you probably can't get tin or aluminum to alloy with metal particles by adding water-soluble salts of tin or aluminum to it and then heating the mix. If you heat salts like aluminum nitrate it's going to be hard to get the metal out of them; instead you'll just get the oxide. (I'm not sure what stannous chloride decomposes to, but I imagine reducing it to tin is hard.) You can eventually reduce just about any metal oxide by heating it enough with hydrogen or ammonia, but that may complicate the situation further here. Such metals could maybe be handled by including some magnesium particles in the bed to steal their oxygens.

Copper and silver salts are more promising here. Silver doesn't really mind being reduced at all, and even copper is only mildly inerested in oxygen. In particular, if copper oxide has a chance to react with aluminum, it does so violently. So you could imagine that nitrates or acetates of copper could be inkjet-squirted into aluminum particles (making, say, about 20% $Cu(NO_3)_2$), heated to decompose them to oxide (say, about 8.4% CuO, which is 80% copper, so the mix was 1.7% oxygen), then reacted with the aluminum to produce an alloy (in the example case, 6.7% copper and about 3.6% sapphire, with the remainder being aluminum.) Some chlorides (of copper, say) might help to initiate the reaction with the aluminum by breaking down the aluminum oxide layer.

Lin, Han, and Li (2012) report that copper acetate dehydrates at 168° and decomposes to copper oxides at 302°. Naktiyok and Özer (2019) report similar results, though they report that the decomposition starts happening below that temperature.

Silver should be even easier than copper to reduce; the

silver–aluminum eutectic, though, is something like 28% aluminum and 566°, with 15% silver needed to drop aluminum's solidus below 600°.

(Although silicon is commonly used to reduce the melting points of metals, I've omitted it entirely here on the theory that very few metals are able to reduce silicates, which are the only water-soluble silicon-bearers that occur to me.)

Indium, bismuth, and gallium might also be effective at reducing metals' melting points, but often the resulting alloys don't have desirable properties. Also, bismuth doesn't seem to have any water-soluble salts, while indium and gallium salts are mostly soluble but don't decompose on heating.

Ammonium dihydrogen arsenate (48.7 g/l) might be a useful source of arsenic for fluxing copper.

Alternative solvents might include carbon tetrachloride, ethanol, carbon disulfide, chloroform, supercritical carbon dioxide, dichloromethane, and ammonia. Carbon disulfide in particular can dissolve sulfur, which forms low-melting sulfides with a huge number of metals; these can then be reduced back to the metal by roasting.

Other ways to get fine particles to not stick together

Flowability has been a major concern for pillmaking for a long time, along with rules of thumb, like a 31–35° angle of repose and under 16% compressibility for good flowability, and 56–65° angle of repose and over 32% compressibility for very poor flowability.

What can we do to get even flow, other than trickler-style rotating tubes?

- Ultrasonic vibration, as I've suggested previously. Even non-ultrasonic vibration is commonly used in pillmaking, with accelerations in the tens of gees; above five or ten gees even relatively stubborn particulates may flow, though if the angle of repose is higher than 65° or the compressibility over 37% even that may not be enough. Particulates with "good flowability" as described above do not need vibration.
- Dilute them with coarser inert carrier particles, or grow coarser particles on them. If each 10-micron-diameter particle of binder is attached to a 200-micron-diameter round particle of ammonium chloride, it will flow easily; mild heating of the particle bed will "sublimate" the ammonium chloride. Many other possible alternatives exist for ammonium chloride: sulfur, dry ice, paraffin wax, other polyolefins, and so on. Other possible ways of removing the inert carrier include reacting it with a gas to form another gas or dissolving it in a solvent, which might permit the use of even table salt. The crucial fact is just that the carrier particles must somehow be made to disappear without disturbing the binder. (Solvent removal ought to use low-surface-tension solvents.) In dry inhalers, lactose is the typical carrier.
- Premix them with some other particles that they don't stick to, but

which form part of the final result. In the case of fluxing an iron particle bed with 3% carbon, you could mix each part of the carbon particles with 3 parts of additional iron particles, so you'd have to add 12% instead of 3%. In systems where a little graphite inclusion would be harmless, you might be able to mix inert graphite particles with the flux particles to keep them flowing freely. Taken to the extreme, this approach amounts to depositing the particle bed like a stack of sand paintings, depositing different mixes in different parts of a layer to form a layer of constant thickness.

• Coat the particle surfaces with something that sticks together poorly; in the case of waterglass particles used to flux quartz, for example, treating the surface with the sorts of silanes used to enhance adhesion of glass fillers to nonpolar polymers might work well, so that when the flux particles touch one another, they encounter only alkane moieties and very little adhesion. Magnesium stearate is commonly used for this purpose in pillmaking, simply by dry-tumbling it with, for example, a microcrystalline cellulose excipient, for a few minutes. It might work even better to coat *some* flux particles with an alkane moiety like the tail of stearate and *others* with a fluorocarbon moiety, so that they will tend to have even less affinity for the foreign half of their neighbors. Other stearates commonly used are that of calcium and of zinc.

• As an alternative to *adding* a low-surface-energy coating it may be possible to *transmute the surface into* one in some cases. Fluorinating the surface of a polymer or metal is one example, although this shades into the sort of silane surface treatment suggested above, and fluorinating some things will *increase* their surface energy rather than decreasing it.

• Keeping them very dry will help in many cases. Materials commonly added to table salt for this purpose include calcium silicate, sodium aluminosilicate (a zeolite), sodium ferrocyanide, and potassium ferrocyanide; other common anticaking food additives that work by similar absorption include bentonite and tricalcium phosphate.

• Presumably there's a temperature effect, but whether being cold or being hot is better, I don't know.

• An ion emitter (or the foil used on those antistatic phonorecord brushes) may help to reduce electrostatic forces that tend to cause clumping. Maybe also using conductive particles or a conductive coating.

• If you run the whole apparatus inside a centrifuge, so that the small flux particles have proportionally more mass relative to their surface forces, that should help.

• Generating the flux particles as smoke generates them out of contact with one another, so they cannot stick to one another. The smoke can be generated in a stream of plasma or gas, which is then sucked the gas through the particulate bed so the flux can deposit. Or it can be generated simply by heating and deposited on the particulate by diffusion.

• Of course to the extent that the fine particles can be spherical rather than irregular, acicular, or platy, they will tend to clump less. The lower the aspect ratio, the better.

• And to the extent that the particles are nonuniform in size, I think the smaller particles will provide more adhesion among the larger

ones.
• On the contrary, though, a small quantity of very fine irregular particles should help to keep a much larger quantity of much larger particles apart; "glidants" in pills commonly work in this way, including silica gel, fumed silica, talc, magnesia, and even cornstarch. Magnesia was what made Morton Salt pour when it rained.
• Porous and soft particles will adhere to one another at lower pressures than hard ones.

It occurs to me that if a particulate is gently tumbled in a closed drum that has pinholes in its walls, clumps that fall down and impact a pinhole may not fit through, but may be able to eject some particles through the hole. If this can be observed it may be a solution to the problem of depositing ten or fewer particles in a given position.

Topics

• Materials (p. 1138) (59 notes)
• Digital fabrication (p. 1149) (31 notes)
• 3-D printing (p. 1160) (17 notes)
• Powder-bed 3-D printing processes (p. 1226) (5 notes)
• Particulates

A four-dimensional keyboard matrix made of linear voltage differential transformers (LVDTs) to get 30 or 180 keys on five pins

Kragen Javier Sitaker, 02021-05-12 (updated 02021-12-30)
(4 minutes)

Suppose you charlieplex blue LEDs on four pins ABCD of a microcontroller. So you have 6 pairs of lines AB, AC, AD, BC, BD, and CD. To light a forward LED you bring an earlier line high and a later line low, and the voltage is below twice the threshold voltage, so when you bring A high and C low, LED AC lights, but the LEDs AB and BC (which you might suppose would supply a secondary current path) do not exceed their threshold voltage and so do not light.

You can get 12 LEDs on 4 pins by charlieplexing by adding reverse LEDs BA, CA, DA, CB, DB, and DC.

Consider LVDTs. At the most basic level, an LVDT creates or destroys an inductive coupling between two coils depending on the location of a sensed element, typically a magnetic core. The "L" means that the coupling is linear in the position of the core, and the "D" means that the way it is destroyed is by precisely balancing couplings in two opposite directions.

Between these 12 current paths there can exist 66 pairwise inductive couplings; in series with the AB LED we can have inductive elements that couple that current path to current paths BA, AC, CA, AD, DA, etc. We can literally create these couplings by putting 11 LVDT windings in series with each of the 12 LEDs. But can we sense them independently?

I think the answer is "not quite". If we bring A high and B low, then we cannot sense voltages on those pins (unless we are using weak pullups and pulldowns like those in the STM32). But maybe if flowing 20 mA from A to B causes C to be pulled up or down from either A or B, we can detect this, and likewise for D. Moreover if the induced voltage is at some point predictably less than the voltage being applied across A and B, as it surely will be if the AB current tends toward an asymptote, we can distinguish C being pulled up from A (which will be limited by the input protection diodes to 3.9 volts or so) from C being pulled up from B (which might be, say, 1.5 volts). So that gives us 12 keys: two for each of our 12 current paths, but each key appears twice, since the coupling between an AB winding and an AD winding will appear both as an AD voltage on an AB current pulse and an AB voltage on an AD current pulse. Maybe you can increase this further with different polarities and coupling strengths, although that would seem to eliminate the possibility of using continuously varying coupling strengths to, in effect, charlieplex analog sensors.

If C is being pulled up or down from D, this may be more difficult

to distinguish if both C and D are floating. We might have 1.5 volts of difference between the two, but does that mean C is at 0 and D is at 1.5 volts, or that C is at 1.8 and D is at 3.3? This is maybe complicated by diode blocking. But maybe twiddling weak pullups and pulldowns at C and/or D can distinguish.

Unlike just charlieplexing a switch in series with each LED, this approach allows you to use the LEDs independently for output.

With five pins the picture is rosier still: instead of 12 current paths we have 20, and each of those 20 can be sensed on any of 3 other pins, for a total of 30 keys. Or maybe each of those 20 current paths can be usefully coupled to any of 18 others, and we can distinguish all of them, in which case you have 180 keys.

Topics

- Contrivances (p. 1143) (45 notes)
- Electronics (p. 1145) (39 notes)
- Human-computer interaction (p. 1156) (22 notes)
- Input devices (p. 1252) (4 notes)
- Keyboards (p. 1289) (3 notes)

Planetary screw potentiometer

Kragen Javier Sitaker, 02021-05-12 (updated 02021-12-30) (1 minute)

You could integrate a multiturn potentiometer into a planetary screw without occupying extra space, though plausibly this is a bad idea. One or more of the planets is metallic or otherwise strongly conductive, one of the threads on the continuous track (the nut for an inverse planetary screw, the screw for the normal type) is resistive, and the other linear element (the screw for an inverse planetary screw, the nut for the normal type) is conductive. This gives you ratiometric positional feedback and a potentiometer without sliding contact (and thus plausibly longer life than potentiometers normally have, though I suppose that if this is beneficial then there must be existing ball-bearing or gear potentiometers).

This might involve undesirable compromises to mechanical properties. Probably the best material for the planets is either conductive or non-conductive, so making one of them conductive represents a compromise. And needing to make most of one of the long linear elements insulating may be a drawback, since metals are plausibly better materials for them.

Topics

- Contrivances (p. 1143) (45 notes)
- Electronics (p. 1145) (39 notes)
- Mechanical (p. 1159) (17 notes)
- Input devices (p. 1252) (4 notes)
- Roller screws (p. 1275) (3 notes)

3-D printing in carbohydrates

Kragen Javier Sitaker, 02021-05-16 (updated 02021-12-30)
(10 minutes)

I was thinking about simple sugar syrup, which has a glass transition around room temperature or even below depending on the water content (the eutectic point is 60% sucrose and -9.5°, but at that concentration the glass transition is about -90°, rising above 0° around 85% sugar and increasing to 52° at 100% sugar), and which can either be easily crystallizable or stable against crystallization depending on its specific composition. (Using isomalt, which normally only crystallizes in a hydrated form, rather than or in addition to sucrose is one approach popular for keeping sugar art from crystallizing; hydrolyzing some of the sucrose with lemon juice is another, and mixing in some high-fructose corn syrup is a third.)

The first 3-D printer I ever saw was the CandyFab 4000, which melted sugar with hot air, which happens at 160° to 186° in a very complex way, at which temperature the sugar caramelizes fairly rapidly, enough to produce a noticeable discoloration in the few seconds the CandyFab 4000 kept the sugar molten.

But in sugar art, sugar glass is maintained in its rubbery, plastic state at room temperature with the addition of water.

Glasses drop dramatically in viscosity above their glass transition temperatures. So, for example, syrup of 60 wt% sugar has 113 centipoise at 10°, but 56.7 cP at 20°, 34.0 cP at 30°, 21.3 cP at 40°, 14.1° at 50°, and 4.17 cP at 90°. Moho says that, at 76%, the viscosity at 30° is 1200 cP, dropping to 510 cP at 40°, 250 cP at 50°, 130 cP at 60°, and 47 at 80°. He gives no viscosity at 20°, so presumably it's effectively no longer a syrup for confectionery purposes.

However, PLA extrusion normally happens at 3-20 kilopoise, i.e., 300-2000 Pa s. This is in the range Moho gives for toffee fondant mass (Table A1.34, p. 572): 17%-water Kis-Kis toffee mass is 2.48 Pa s (2480 centipoise) at 60° and 0.26 Pa s at 100°, while 8%-water Kis-Kis toffee is 487.8 Pa s at 70° and 43.0 Pa s at 90°. Wikipedia claims toffee is more like 1% water, the "hard crack" stage (boiling at 146°-154°), so perhaps "toffee mass" is a different substance.

Regardless of exactly how much water is needed to plasticize sugar to an extrudable stage, it's clear that such a level does exist, and the resulting substance hardens rapidly as it cools.

Pok Yin Victor Leung investigated hard-crack candy printing in 02017 as a more accessible model for printing optics in soda-lime glass, using sucrose and high-fructose corn syrup gravity-fed from a reservoir maintained at 98° with a PID controller with only a manual valve. He got beautiful results but reports that the shining golden objects thus printed were deliquescent.

A standard candy sealing process is "hard sugar panning", in which hard sticky candy balls are rolled around in syrup which crystallizes on the surface as it dries, sometimes in many layers added over weeks; this is how "jawbreakers" and M&Ms are made. If the syrup used is a non-crystallizing syrup like glucose syrup, the process becomes "soft

panning", and powdered sugar can harden it, producing jellybeans. Such crystallization would be undesirable for Leung's purpose of 3-D printing optics, but it would solve the deliquescence problem.

There are a variety of other possible ways to harden such a surface besides dusting it with sugar, though. Shellac, for example, is commonly used in candymaking, with zein as an up-and-coming alternative that also leaves the result edible. You could also include sodium alginate in the syrup and harden the surface with calcium ions, or vice versa, or include something that hardens instead of deliquescing when it reacts with water from the air. Or perhaps you could wash the surface with a desiccant such as ethanol or a strong solution of muriate of lime.

Of course, if edibility is not a requirement, there are lots of coatings you can use. Possibilities for hardening systems to use as coatings include molten wax, cyanoacrylate, plaster of paris, spray paint, resins that polymerize on the object (such as silicone, epoxy, or acrylic), polymers dissolved in a solvent (such as acrylics, ABS, or polystyrene, dissolved for example in acetone or gasoline), lime concrete, OPC concrete, geopolymer concrete, or soluble silicates such as that of sodium (perhaps desolubilized by polyvalent cations added to the syrup). In addition to using these hardening systems simply as coatings, you can also just pour them over the printed sugar object, embedding it in a block; once this block has solidified, you can dissolve the sugar out of it with enough flowing water, ideally warm.

Some of these coating systems include free water, which poses a potential problem: at the interfacial layer between the hardening system and the sugar object, water will be migrating out of the hardening system and into the sugar, swelling and liquefying the sugar, while diminishing the water available to the hardening system, potentially impeding its hardening. This may be actually desirable, acting as a sort of inbuilt mold release and avoiding the need to melt or dissolve the sugar out of the hardening system's product; even if not, the affected layer may be thin enough to be acceptable.

In other cases, it may be possible for the hardening system to actually *extract* the water it needs from the sugar object; for example, methyl cyanoacrylate or the usual silyl acetates that comprise acetoxy-cure silicones (largely methyl triacetoxysilane, I think) will happily steal water in such a situation, and I think plaster of Paris can too. So it may be possible for them to be applied as a bath or powder coat and selectively harden on the surface of the printed object.

A totally different approach is to postprocess the print to get rid of the sugar, which is especially appealing if the sugar syrup is mostly used as a plasticizing and sticky carrier for a solid particulate "filler" that is the real printing payload, much like epoxy is used in JB Weld or PLA is used in brass-filled PLA filament. (Such fillers, in addition to adding numerous useful properties to the resulting object, may help to make the melt thixotropic, easing the compromise between flowing easily through the hotend nozzle and staying in place once extruded.) The easiest way to remove the sugar is to heat the piece to caramelize it, eventually producing carbon. If it's thin enough and it's heated slowly enough, this can be done without provoking water-vapor bubbles.

(Are we blowing hot and cold with one breath here? Why won't the syrup clog up the extruder if heating it caramelizes it? It might not work, but my thought is that in the extruder it's potentially only hot for a few seconds, during which time it is under a lot of shear stress, while we can keep it at a lower temperature overnight or longer with very little stress in order to caramelize it.)

I haven't had much luck getting caramelized sugar to stick quartz sand together, but it's well known how difficult it is to get it off steel or stainless steel, often requiring lye.

In addition to solid fillers, another possible additive to improve thixotropy of water-plasticized sugars is emulsified oil, as in mayonnaise, though of course in mayonnaise it's soluble protein that's being plasticized by the water, not sugars. Oil droplets dispersed in an emulsion can give rise to quasi-elastic behavior.

I thought the same was true of dulce de leche, but that turns out to be wrong. Dulce de leche is an emulsion, but it is also a much more complex system containing proteins and polysaccharides which form a gel structure; it's normally 6–8% fat and 31–34% water. This is not enough fat to give the emulsion quasi-elastic behavior; instead it is pseudoplastic like molten polymers made of long linear molecules that form ephemeral entanglements, and also forms a gel.

A different family of carbohydrate 3-D printing is suggested by pasta and flubber, which are made out of starch granules and water; the starch granules can be suspended in water by purely mechanical means. Heating the water enables it to dissolve the starch granules, in a process called starch gelatinization, and the resulting viscous solution of amylose and amylopectin (boh polysaccharides) behaves much like a sugar syrup.

There are various ways to crosslink these starch molecules to reduce their solubility, including using glow discharge plasma, phosphorus oxychloride (POCl3), citric acid (with a sodium hypophosphite catalyst), sodium trimetaphosphate, boric acid, formaldehyde, and sucrose oxidized by periodate cleavage with sodium periodate to form random aldehydes; and polyols like glycerol or sorbitol can be used to plasticize the resulting insoluble plastic. Glyoxal, the simplest dialdehyde (and essentially nontoxic, 3300 mg/kg) is consumed in mass quantities to thus crosslink starches for sizing paper and textiles. Glutaraldehyde (plantar wart remover, also used in tanning leather, and as a biocide in fracking) seems like it should work.

Topics

- Materials (p. 1138) (59 notes)
- Digital fabrication (p. 1149) (31 notes)
- 3-D printing (p. 1160) (17 notes)
- Sugar (p. 1271) (3 notes)
- Glutaraldehyde (p. 1294) (3 notes)
- Cross linking (p. 1377) (2 notes)

Clay-filled PLA filament for firing to ceramic

Kragen Javier Sitaker, 02021-05-17 (updated 02021-12-30) (1 minute)

Curiously, I just realized I haven't heard of clay-filled PLA filament before, but it turns out that it does exist; the clay increases shear thinning, thermal stability, crystallinity, and rigidity, and at 40% clay the print can be fired to ceramic, though melt viscosity increases. Other mineral fillers currently on sale include zircon and sapphire, both of which claim to be sinterable in the same way. I'd think that including clay would also lower the cost and increase strength without firing, but that doesn't seem to be the case yet; in this experiment the clay actually decreased tensile strength in most cases by 5-20%.

They bought a prefunctionalized clay whose preparation I don't understand at all:

> For nanocomposites preparation, an organo-modified layered silicate (Cloisite 30B, Southern ClayProducts Inc., Gonzales, TX, USA), modified by methyl, tallow, bis-2-hydroxyethyl, and quaternary ammonium chloride, having a basal interlayer spacing d001 = 18.5 Å, was used.

Topics

- Materials (p. 1138) (59 notes)
- Digital fabrication (p. 1149) (31 notes)
- 3-D printing (p. 1160) (17 notes)
- Filled systems (p. 1161) (16 notes)
- Clay (p. 1179) (10 notes)
- Ceramic (p. 1193) (8 notes)
- Poly(lactic acid) (PLA) (p. 1281) (3 notes)

Multicolor filament

Kragen Javier Sitaker, 02021-05-17 (updated 02021-12-30)
(5 minutes)

A YouTuber named Sunshine has demonstrated an interesting technique for varying colors in an FDM 3-D print with a single hotend, using the technique with PLA vaguely similar to knitting with variegated yarn. This also permits you to produce an unlimited variety of colors within the color gamut spanned by two or more of your filaments.

He first prints out a spiral on his printer bed using two or more filament colors in two or more printing passes; then he pops the spiral off the bed and uses it as the filament for the final object. Unless you do an additional filament-spiral-printing step, the resulting color depends on the direction of movement during extrusion, providing a color gradient.

At this point Sunshine is only printing a filament that remains constant in composition from beginning to end, varying only laterally, but it's obviously easy to vary the composition as the filament progresses, producing temporal variation during the print.

It occurs to me that your slicer knows within a few millimeters how much filament will be extruded at various points in any given print. So you could actually synchronize the color changes to different parts of your print, so that instead of just the smooth color gradients Sunshine demonstrates, you can perform arbitrary multicolor printing in this way. (Marco Reps covers a multi-material filament splicer called the Palette Plus from Mosaic Manufacturing designed with this purpose, but concurrent with the actual 3-D print itself, using periodic pauses to fix desynchronization.)

Doing this straightforwardly will suffer from some imprecision in the color change, as the hotend's melt chamber gradually changes from one color to another, and also due to unavoidable imprecision in the precise timing. When this isn't desired, you could insert G-code into the print that moves the hotend off to the side of the print and extrudes enough spaghetti to achieve the desired sharpness of color change, wasting a little plastic, or you could choose to spend the transition zone on inside perimeters or infill, where it won't be visible. Typically you would have to do this two or more times per layer.

By synchronizing the color changes to the amount extruded on a given circumference of the print, you can achieve smooth gradients more intense than those Sunshine achieved.

This filament-mixing process is not limited to color; you can use it to achieve customized material properties, including gradient properties. For example, you could use filament with a metal filler such as brass in areas of the print that need extra strength, density, conductivity, rigidity, or shininess, or mix varying amounts of thermoplastic elastomers into your ABS or PETG to give continuously varying rigidity, or mix acetal into ABS, PETG, or PLA to improve mechanical properties, or print polycarbonate or nylon fibers inside a print to improve impact resistance, or polypropylene to

provide a chemically resistant surface, or include other fillers or additives to make part of a print more malleable, translucent, electrically permittive, phosphorescent, levorotary, permeable, hygroscopic, flame-retardant, bacteriostatic, abrasive, hydrophobic, abrasion-resistant, electronegative, ferromagnetic, high in refractive index, ferrimagnetic, high in specific heat, diamagnetic, fragrant, thermally expansive, flavorful, incompressible, acidic, porous or otherwise permeable, optically dispersive, more dielectrically stable, viscoelastic, high-melting, fluid when liquid, fluorescent, or possessed of some other material property. (Or less so: while flame retardants will make a filament more flame-retardant, oxidizer fillers will make it less so.) And of course you can print water-soluble supports with PVA. This kind of processing has thermal limits, since you can't mix filaments that need incompatible hotend temperatures (for example, PLA and nylon), but those can be extended somewhat with additives such as plasticizers or antioxidants.

Even linear reinforcement like carbon fiber might survive this kind of process.

Some kinds of additives might be harder to come by already mixed into printer filament, but can be coated onto the surface of a filament once it's printed. The geometry of the printed filament can be adjusted to facilitate this kind of surface adhesion by having more surface area. It can also be adjusted to improve the grip of the extruder, which is especially important for filaments that are highly brittle or have high melt viscosities.

The filament in this process makes two trips through the hotend. This will have some good effects, such as boiling out any water it has absorbed while in storage during the first trip; but it also increases the strength loss from hydrolysis. And of course it doubles the wear on the hotend.

Topics

- Materials (p. 1138) (59 notes)
- Digital fabrication (p. 1149) (31 notes)
- 3-D printing (p. 1160) (17 notes)
- Filled systems (p. 1161) (16 notes)
- Poly(vinyl alcohol) (PVA) (p. 1245) (4 notes)
- Poly(lactic acid) (PLA) (p. 1281) (3 notes)

Acicular low binder pastes

Kragen Javier Sitaker, 02021-05-19 (updated 02021-12-30) (1 minute)

By using a platier or more acicular or fibrous granulated filler, you can decrease the amount of binder needed, although at some point you start having the stuff stick together without any binder at all. Using smoother surfaces and larger particles also decreases the amount of binder needed as well as the tendency to stick together without binder.

This has implications for powder-bed 3-D printing: a small amount of binder can bind a large amount of powder into a continuous solid network, although the bound powder may be porous and weak.

Promising fillers that come to mind include talc, mica, milled clays such as kaolin, acicular hydroxyapatite, asbestos, jade, graphite, mullite, rockwool, glass fiber, carbon fiber, acicular wollastonite ($CaSiO_3$, the calcium analogue of enstatite, which has a totally different crystal structure), ceramic fiber, steel fiber, acicular boehmite (AlOOH, which more typically crystallizes in a massive habit),

Topics

- Materials (p. 1138) (59 notes)
- Digital fabrication (p. 1149) (31 notes)
- 3-D printing (p. 1160) (17 notes)
- Filled systems (p. 1161) (16 notes)
- Anisotropic fillers (p. 1218) (6 notes)
- Powder-bed 3-D printing processes (p. 1226) (5 notes)

Cutting clay

Kragen Javier Sitaker, 02021-05-19 (updated 02021-12-30)
(10 minutes)

I've been interested in sheet-cutting automated fabrication for a while as a more practical alternative to 3-D printing for many purposes, and I just realized that there may be a considerably more accessible variant of this technique right under my nose.

I was watching a video on "clean cutting with polymer clay cutters ". The author gets 3-D-printed cookie cutters to cut a previously rolled sheet of "clay" cleanly by first sticking it to something it will stick to, like glass or ceramic, then scraping it off afterwards, a process which may distort it. However, they also point out that putting plastic wrap over the top of the "clay" will keep the cutter from sticking to it, and then the wrap can be peeled away; and then the bubbles trapped under the wrap produce little shallow hollows in the surface.

This wrap-layer approach provoked many thoughts about automated fabrication. This probably works with pastes such as real clay in its plastic state too, and maybe even better in its leather-hard state, I'm not sure. In the leather-hard state adhesion is not much of a problem.

Also, it probably works with molten thermoplastics (rather than plasticizer-impregnated "polymer clay" plastics, as shown in the video). A thermoplastic of a higher melting point can serve as the protective anti-adhesion wrap; for example, nylon oven bags for baking chicken would work adequately for a lot of common plastics that won't dissolve them, or won't dissolve them too fast.

Also, it probably works with soda-lime glass, although I'm not sure what the backing plate needs to be. The anti-adhesion layer in this case might be aluminum foil, but probably a more effective approach is to keep the surface of the glass cool, using forced air or water if necessary.

You could probably use a rolling wheel to "cut" the plastic sheet through the wrap, giving you a vinyl-cutting-plotter-like capability. Certainly you can repeatedly push something like the end of a butter knife or tongue depressor through it. Making multiple trips along the cutting contour at different heights should handle the cases where this doesn't work as well as you'd hope. So this is an appealing technique for sheet-cutting automated fabrication.

Also, you can push a thin rod into the material in many places, for example a chopstick, to be able to cut arbitrary shapes without the limits on sharp corners imposed by the wheel-cutter method.

You can use this approach with a rolling ball (like the spherical casters sometimes seen on office chairs) to *form* the clay or other plastic material instead of *cutting* it. This requires good simulation of the plastic deformation to plan the forming toolpath. The wrap will reliably keep the plastic sheet or other plastic workpiece from adhering to the forming tool.

Of course it also works with stamps pre-designed to form the

surface of the material rather than cutting all the way through like the cookie cutters. (Some cookie cutters have such stamps already incorporated.)

For many sticky plastic materials, there exists some kind of powder that can be dusted on the surface to prevent adhesion: talc, cornstarch (suggested in the video, which claims it doesn't prevent sticking in this case), quartz flour, zirconia dust, whatever. In other cases a liquid coating will prevent sticking and may also serve as a lubricant. In some cases you would like this to be removed or fuse with the workpiece after it has its shape, and with the appropriate choice of powder this can be achieved by means such as washing, raising the temperature further, solvent vapor smoothing, or liquid solvent smoothing.

Instead of a film, powder, or nonplasticized part of the underlying plastic slab, fibers may be a feasible alternative: woven, felted, or especially knitted. This is appealing because cloth can remain flexible and resilient over a wider temperature range; even fairly minor plastic or elastic deformations in the fiber material can permit gross deformation of the cloth, protecting the "cutter" from adhesion as it deeply penetrates the surface of the slab. It can also provide better thermal insulation than powders or films. For example, knitted fiber of silicon carbide or zirconia could permit pizza-cutter action on glasses whose glass transition is at 1500° or more.

Other kinds of filled systems are appealing, too. For example, filled system consisting of 65 vol% stainless steel powder, 5% of a sintering aid (such as borax, potassium bisulfate, boric acid, fluorspar, sulfur, or brass), and 30% poly(lactic acid) or delrin would probably become plastic at the usual temperatures for PLA (around 185°–230°) or delrin, at which temperatures the sintering aids should be inactive. Heating to a much higher temperature should activate the sintering aid and bind the particles together, as well as burning off the organic polymer; sufficient time at sintering temperature would result in a porous sintered stainless steel body. However, such a the low-temperature plastic matrix will likely burn off long before the sintering aids can activate, so you might need a small amount of a fourth ingredient to add enough green strength by holding the particles in place; candidates include soluble silicates, clays (especially highly plastic clays like bentonites), carbohydrates such as sugar, and organic thermosets.

Sulfur is a particularly interesting possible binder/matrix for such filled systems because at one temperature it is very liquid, and then at a higher temperature it polymerizes into a plastic form which can be molded; if cooled quickly enough (traditionally by water quenching), it then gradually hardens through further polymerization. Moreover, it reacts with many possible metal fillers to form low-melting sulfides, which can then be transmuted back to the original metals by further heating to drive off the sulfur.

Water-soluble carbohydrates such as sugars, isomalt, carboxymethylcellulose (a popular choice for pottery), or gelatinized starch have the potential advantage that, like real clay, you can plasticize them with a small amount of water so that they flow readily at easily accessible temperatures (0°–100°), but upon being maintained

warm for a longer time in contact with air, they will lose water and in some cases can be induced to crystallize by such dehydration. (This is the principle behind making fruit leather, for example.) Once their water content is low enough to avoid forming bubbles that would disturb the form, they can be caramelized at somewhat higher temperatures (150°–250°) to form a heat-stable green body that will carbonize, and the carbon will survive for a substantial period of time even at metal-sintering temperatures, before finally debinding the sintered piece through oxidation.

Such carbon-containing binders/matrices or sintering aids may add carbon to the final product, for example if the "filler" contains iron or silicon.

What should you use as the backing sheet? In the case of real clay or similar preceramics which will be fired afterwards, one possibility is to use sacrificial organic material or or sulfur, either of which which will fully gasify during firing in an oxidizing atmosphere; this will avoid distortion from the peeling process used in the video. Sulfur or zinc will also simply boil away, even in a reducing atmosphere, but the gases may cause problems for things other than the workpiece, for example by forming metal sulfides.

In the case of molten thermoplastics, experience with RepRap-derived 3-D printers has shown that many thermoplastics will adhere nicely to a sheet of warm glass, then delaminate from it without breaking due to thermal contraction when allowed to cool. This may work for soda-lime glass on sheet steel too, I'm not sure. Another popular option which will also work here is to use an elastic flexible backplate made from something like thin sheet steel; this can also be peeled off the finished object once it is cool.

Another alternative may be backing sheets that can be destroyed, or whose workpiece-contacting surface can be destroyed, by a reagent that will spare the workpiece itself. In semiconductor fabrication, for example, hydrofluoric acid with a little nitric is routinely used to remove exposed silicon dioxide without attacking the silicon, aluminum, copper, and I think hafnia parts of the chip, while caustic potash is routinely used to etch away silicon (along certain crystal planes! and only if it doesn't contain enough boron!) and aluminum without affecting silicon dioxide or nitride; in jewelry, hot aqueous alum solution is routinely used to dissolve broken steel taps from workpieces made of other metals; and caustic soda will rapidly dissolve aluminum and etch amorphous silica, and with sufficient heat will dissolve crystalline silica, but leaves most metals untouched.

The air bubbles in the video suggest another fascinating possibility: forming the surface of a sheet of material by injecting pressurized gas or liquid between it and some kind of substrate, whether flexible like the wrap or rigid like the glass.

And of course the overall process is not so far different from things like hot-needle cutting of foam sheets, or hot-wire cutting.

Topics

- Digital fabrication (p. 1149) (31 notes)
- Filled systems (p. 1161) (16 notes)
- Clay (p. 1179) (10 notes)
- 2-D cutting (p. 1201) (7 notes)
- Self replication (p. 1204) (6 notes)
- Sugar (p. 1271) (3 notes)
- Forming (p. 1295) (3 notes)

Scaling laws

Kragen Javier Sitaker, 02021-05-19 (updated 02021-12-30)
(8 minutes)

Galileo's square-cube law explains that a cylinder over a given dimension will collapse under its own weight:

> From what has already been demonstrated, you can plainly see the impossibility of increasing the size of structures to vast dimensions either in art or in nature; likewise the impossibility of building ships, palaces, or temples of enormous size in such a way that their oars, yards, beams, iron-bolts, and, in short, all their other parts will hold together; nor can nature produce trees of extraordinary size because the branches would break down under their own weight; so also it would be impossible to build up the bony structures of men, horses, or other animals so as to hold together and perform their normal functions if these animals were to be increased enormously in height; for this increase in height can be accomplished only by employing a material which is harder and stronger than usual, or by enlarging the size of the bones, thus changing their shape until the form and appearance of the animals suggest a monstrosity.

Different properties

Suppose we scale some physical system up by some factor f, or down if $f < 1$. Not only will some of its static properties change, as Galileo points out above, but also some of its other behaviors will speed up or slow down. But by how much?

Mass

The mass and thus weight of a body will tend to scale as f^3.

Tensile failure

The tensile strength of a member will tend to scale as f^2, so by smallifying an object we make it stronger ($1/f$ times) relative to its own weight. If we consider stress from accelerations, well, this means we can accelerate it faster ($1/f$ times) before it fails in the tensile mode; relative to its own dimension, this is an advantage of a factor of $1/f^2$ in acceleration, but this only works out to a factor of $1/f$ in frequency.

That is, suppose we have some sort of machine in which a weight is being jerked back and forth along a course inside the machine by a rope, and if we run the machine too fast, the rope will break. We smallify the machine, say by a factor of $1/f = 3$. Now the weight is being moved a 3 times shorter distance, it has 27 times less mass, and the rope can withstand 9 times less tension. So the rope can withstand $27/9 = 3$ times greater acceleration, but at a given jerking frequency, 3 times less acceleration would be needed to jerk the weight the same distance. But the length of the weight's course within the machine has also diminished by a factor of 3, so at the same operational frequency, the stress on the rope has actually diminished by a factor of 9.

However, if we run the machine 3 times as fast, so that it jerks the weight 3 times as often, the velocity increases by a factor of 3, but the acceleration increases by a factor of 9. So, in the mode of tensile failure, smallifying does increase the maximum frequency, but only

proportionally, not quadratically as one might hope.

Buckling failure

XXX Euler columns

Shear failure

XXX

Compressive failure

XXX

Elastic beam bending

The usual expression for rectangular beam bending stiffness is $k = Ebh^3/4L^3$, where E is the material's modulus, h is the thickness of the beam along the direction of bending, L is its length, and b is its width transverse to the direction of bending. If we scale the beam uniformly up or down by some factor, then the changes in bh^3 and L^3 leave the stiffness varying directly with the scale.

Resonant tines

The general expression for the resonant angular frequency of a sprung-mass system is $\omega = (k/m)^{½}$. So, if a mass is at the end of an elastically bending beam, the mass m changes by a factor of f^3 while the stiffness k changes by a factor of f, so its resonant angular frequency changes by a factor of $1/f$.

tensile-mass oscillation

shear-mass oscillation

Dennard scaling

XXX

Resistors

A resistor's resistance is $\varrho L/bh$, where ϱ is the resistivity. If it becomes f times longer and f times wider and deeper, it will thus diminish in resistance by that same factor f. Smallifying resistors thus makes them proportionally higher in resistance; we might say that *conductance*, rather than resistance, is proportional to scale.

A thin resistive film whose thickness does not change is well known to have a characteristic "resistance per square": its resistance over 1 mm × 1 mm is the same as its resistance over 1 cm × 1 cm or 1 μm × 1 μm.

At ordinary (macroscopic) sizes it is easy to achieve an enormous range of resistances at any scale, because in a given volume you can either make a long, thin conductor or a short, thick one, and moreover available resistive materials themselves cover several orders of magnitude of resistivity; resistors from 0.001Ω to 1GΩ are all staple articles of commerce, and resistors from 1Ω to 1MΩ are common, and all of those common ones are available at any size from 0402 up to many watts. At the lower end these resistances are limited by the resistances of the commonly employed wires (i.e., not superconductors, but materials like copper) and at the upper end by the leakage currents of materials such as glass, polypropylene, and Teflon, used as insulators.

On chip, the story is different, because although low resistances are easily accessible (just use a metal layer) high resistances such as 100kΩ are not; they occupy an inordinate amount of space and are a nuisance.

Capacitors

A (planar) capacitor's capacitance is $\varepsilon A/d$, where ε is the permittivity of the dielectric. This is proportional to f, so smallifying capacitors thus makes them proportionally smaller in capacitance.

Common macroscopic capacitors cover an even wider range than resistors, from 10 pF to 50 F, or 10000 μF if we exclude supercapacitors.

RC time constant

Perhaps the most common way to mark time in an electrical circuit is with an RC circuit with an exponential decay time constant $\tau = RC$; this is how RC filters and 555 timer circuits work, for example, and how the internal oscillators in microcontrollers work. Since R is inversely proportional to f while C is proportional to it, smallifying an RC circuit will not change its frequency response.

This is an astonishingly different result from Dennard scaling.

Inductors

A cylindrical air-core coil has an inductance of roughly $\mu N^2 A/L$. Scaling it by f increases A by f^2 and L by f, so inductance scales with f. So inductance, like capacitance and conductance, is proportional to scale.

LC frequency

The resonant angular frequency of an LC circuit is $\omega = (LC)^{-½}$. So multiplying both L and C by a factor f will multiply ω by $1/f$; larger LC circuits oscillate proportionally slower.

RL time constant

etc.

electromagnetic relays

electrostatic relays

crystalline structure

heat transfer characteristic time

turbulent vs. laminar flow

matter diffusion

composite materials

aerostats

heat exchanger design

mass transfer design

liquid friction

Topics

- History (p. 1153) (24 notes)
- Physics (p. 1157) (18 notes)
- Small things (p. 1223) (5 notes)
- Galileo (p. 1367) (2 notes)

Selectively curing one-component silicone by injecting water

Kragen Javier Sitaker, 02021-05-19 (updated 02021-12-30)
(2 minutes)

I understand that the usual single-component RTV acetic-cure silicone cures by hydrolyzing silyl acetates, consuming water in the process, normally from the air (and producing acetic acid). That's why it doesn't cure in the tube and why it takes a long time to cure if you spread it too thick.

It occurred to me that this affords a simple silicone 3-D printing process: first, you make a big block of the liquid silicone in a cup, and let it skin over the top. Then, you insert a hypodermic into the silicone through the skin to the bottom of the cup, withdrawing the needle while injecting water. By repeating this process you can create water bubbles at different locations in the silicone mass, close enough together that the silicone around them will join into a continuous object when enough of it is cured by the water. Once the curing has proceeded to this point, you must remove the uncured silicone; Smooth-On recommends acetone or mineral spirits (naphtha or Zippo fuel), while others report that ethanol works too, and some even report success with ordinary aqueous detergents (Dow Corning DS-1000 Aqueous Silicone Cleaner is such a formulation). GE recommends mineral spirits or "rubbing alcohol" (probably 70% isopropanol).

This kind of needle injection of reagent into a gel or viscous liquid is much more broadly applicable. You can thus inject dyes into jello, for example (according to *Soonish*); or you can inject carbon dioxide or calcium chloride into a green body of quartz sand lightly moistened with soluble silicates, which will harden the silicates immediately; or you can inject cold plasma (of air, for example) into a wide variety of powders to activate the surfaces of powder particles to get them to clump together;

Topics

- Materials (p. 1138) (59 notes)
- Digital fabrication (p. 1149) (31 notes)
- 3-D printing (p. 1160) (17 notes)
- Silicone (p. 1329) (2 notes)

Clay wire cutter

Kragen Javier Sitaker, 02021-05-21 (updated 02021-12-30)
(2 minutes)

A clay wire cutter makes curved cuts. If it enters a planar surface of a clay block all at once while it is pulled taut, the cut starts out straight, but if it enters incrementally, the cut can start out curved; and of course the path on which it is pulled through the clay can also be curved.

The cut path has a tendency to have negative Gaussian curvature as initial wiggles in the wire tend to straighten out; I am not sure if it the cut surface through homogeneous clay is necessarily a minimal surface.

By moving the wire ends under automatic control it should be possible to make preprogrammed cuts, and by using a mathematical optimization algorithm such as Nelder-Mead or gradient descent on either a simulation or experimental data, it should be possible to design those preprogrammed cuts to produce a wide variety of geometries in the clay. Simulation parameters can be tuned from experimental results, for example using a mathematical optimization algorithm such as gradient descent.

The same thing is true of hot-wire foam cutters that contact the foam, but the process has an additional degree of freedom, the speed. The trajectory of the wire through clay is almost insensitive to speed because the clay is almost purely plastic, but that is clearly not the case for hot-wire cutters, which encounter not only viscous resistance but time-varying viscous resistance as the molten plastic around them heats up. By adjusting the wire current up and down, they can even vary this variation with time.

Feedback from the wire cutting process can come in different forms: tension, electric resistance, and electric impedance tomography through the clay, for example. This can improve the quality of the control algorithm by improving the picture of what is going on during the cut, permitting incremental replanning to improve the quality of the final result.

Topics

- Digital fabrication (p. 1149) (31 notes)
- Clay (p. 1179) (10 notes)
- Foam (p. 1185) (9 notes)
- Control (cybernetics) (p. 1262) (4 notes)

Electroforming rivets

Kragen Javier Sitaker, 02021-05-22 (updated 02021-12-30)
(2 minutes)

By discharging a capacitor through a coil you can electromagnetically form metal nearby the coil; this is most famous in the form of "quarter shrinkers", but those exert so much force that they destroy the coil. But electromagnetic forming can be applied in less violent ways that permit the coil to be used repeatedly. I have seen spectacular demonstrations by a scholar at OSU using pulses of only 2kJ and 4kJ.

Unlike with physical impacts, there's no inherent limitation to how fast this can happen; a hammer moving at 100 m/s can't push material ahead of it faster than 100 m/s (though bouncing off it can accelerate hard material to faster than that). But electromagnetic forming is limited only by the inductance and voltage of your coil; submicrosecond rise times should be feasible, although I think there is a tradeoff in which larger numbers of turns produce slower rise times (at a given voltage) but higher peak forces. This might make it possible to do cold welding, particularly of thin foils.

One use I was thinking about just now is fastening hollow rivets, similar to pop rivets, through holes. By inserting a coil into a metal tube and discharging a pulse through it, the tube can be plastically expanded, and its ends can be flared, forming a permanent connection. If the center of the coil is either a ferrite (ferromagnetic or ferrimagnetic) or some other kind of hard ceramic such as alumina or zirconia, it will tend to protect the coil from being shrunk in the process.

The same coil may be used to preheat the rivet through electromagnetic induction in order to soften it before the electroforming operation; induction heating is feasible for any metal, but is easier (can be done at lower frequencies) for ferromagnetic metals.

The same kinds of forming can be used to seal together pipes or structural tubing end-to-end, as mentioned by the OSU scholar.

Topics

- Manufacturing (p. 1151) (29 notes)
- Welding (p. 1181) (9 notes)
- Forming (p. 1295) (3 notes)
- Electroforming (p. 1372) (2 notes)

Metal welding fuel

Kragen Javier Sitaker, 02021-05-23 (updated 02021-12-30)
(6 minutes)

I was watching an Abom79 video about spray welding (or “thermal flame spraying”) and it occurred to me that maybe you can dispense with the welding gas entirely.

Suppose you wanted a welding torch, or better a metal-cutting torch, that used *only* iron filings as fuel, perhaps only for emergency situations. Clearly burning iron or steel can produce a high enough temperature to burn iron or steel in a stream of oxygen, and I think compressed air would also work, though it isn’t the usual practice. You could think of this as a thermic lance adapted to use powder.

You could imagine that the “flame” it produced could be quite small, perhaps submillimeter in size. Perhaps initial ignition could be provided by an arc igniting iron filings suspended in air within an ignition chamber; once ignition was achieved, though, it would probably be more practical to continue combustion outside the torch body in eddies of more-slowly-moving gas, just because otherwise the rapid heat production seems certain to destroy the torch, even if the combustion chamber is pyrolytic graphite. Maybe if you water-cooled the walls or something.

The enthalpy of formation of formation of Fe2O3 is -824.2 kJ/mol, and its molar mass is 159.687 g/mol, of which 2×55.845 = 111.690 g is the iron fuel. Unlike the case with things like propane, the combustion product is liquid rather than gaseous (Fe2O3 melts at only 1539° but doesn’t boil until 2664° according to PubChem; magnetite, which melts at 2623°, oxidizes to hematite upon sufficient roasting in air, while producing magnetite from hematite normally requires reduction with hydrogen, though presumably a much higher temperature would also work) so heat is not carried away nearly as rapidly. At ordinary temperature its heat capacity is some 103.9 J/mol/K, which would give us an extrapolated “maximum flame temperature” of some 7900 K above ambient (824200/103.9); NIST gives 68.2 J/mol/K in liquid form, giving an even higher maximum temperature.

The stoichiometric mixture of iron filings and oxygen, which is probably reasonably close to optimal for this sort of thing, would be two moles of iron (111.690 g, as explained above) to three of atomic oxygen (48 g) or 1.5 moles of oxygen molecules. Solid iron weighs 7.874 g/cc, so this is about 14 cc of iron; a room-temperature mole of an ideal gas is about 24 liters at atmospheric pressure and 20°, so to burn those 14 ml of iron we would need 36 STP liters of pure oxygen or 172 of ordinary air. A more practical point of view is that 111.690 *mg* of iron would occupy 14 μl and need 172 *ml* of uncompressed air to burn it, yielding 824 J.

Probably about as far as you can compress air in practical terms is some 4000 psi, in medieval units, or 28 MPa or 270 atm; if it were to behave as an ideal gas at this concentration, then when cooled to room temperature, it would have a density of some 0.33 g/cc; the 172 ml of

room-temperature air would then occupy some 640 μl.

If this mixture of air and iron were burned and squirted out in a 100-micron-wide burning jet, I feel like it could get pretty hot, but how hot would I guess depend on the equilibrium between the process of combustion generating heat; the process of radiation, which would cool the jet; and the processes of of expansion and mixing with cooler air, which would tend to expand the jet out pretty rapidly.

The maximum energy density of this mixture would be 824 J per (640 + 14)μl, which is about 1.3 MJ/l, similar to primary-battery levels and almost up to rocket-propellant levels. It isn't obvious to me how to calculate the maximum power per unit area that could be supplied by the hot jet; that would seem to depend on the achievable gas velocity and combustion velocity of the jet.

The 78% nitrogen mixed into air has a heat capacity of 29.124 J/mol/K, and there would be 5.6 moles of nitrogen per mole of Fe2O3, adding a considerable extra "thermal mass" of 162 J/mol/K to the 103.9 J/mol/K of solid Fe2O3 or 68.2 J/mol/K of liquid Fe2O3. Still, 824200/(162+104) is still 3100 K, which is still way hotter than we need.

The relatively small amount of carbon mixed into steel will produce carbon monoxide or dioxide; in the context of burning in highly compressed 78%-nitrogen air this is not significant, but of course it has a very visible effect when steel sparks are burning on their own.

You'd probably want to either sift the metal powder down into the gas jet, as modern spray-welding torches do, or ensure that the gas blowing up through the metal powder reservoir was traveling fast enough to prevent flashbacks into the metal reservoir.

Speaking of water cooling, some metals can burn in steam instead of air; this has the potential advantage that steam is 89% oxygen rather than 21% and wouldn't need to be compressed the way air is. However, I don't think iron can burn this way. Water's enthalpy of formation is -285.83 kJ/mol, so stealing the three oxygen moles for a mole of Fe2O3 would take 860 kJ. So I think the reaction would be endothermic. Powdered aluminum or magnesium would work, though, and although aluminum oxide would be a nuisance if you were trying to weld or cut steel, magnesium oxide might be tolerable.

Topics

- Materials (p. 1138) (59 notes)
- Manufacturing (p. 1151) (29 notes)
- Physics (p. 1157) (18 notes)
- Aluminum (p. 1180) (10 notes)
- Welding (p. 1181) (9 notes)
- Magnesium (p. 1213) (6 notes)

Aluminum foil

Kragen Javier Sitaker, 02021-05-24 (updated 02021-09-11)
(14 minutes)

Kitchen aluminum foil is a remarkable material.

It's typically 10 μm thick and 400 mm wide, giving it an aspect ratio of 40000 in that dimension, and rolls are commonly some ten meters in length, for an aspect ratio of 1 000 000; heavy-duty versions can reach 30 μm or more. Despite their thinness, foils of 25 μm or more are impermeable to oxygen, water, and light, though Wikipedia claims thinner foils typically are plagued with pinholes. It comes in a fully annealed state, so it rapidly work-hardens when bent, and because of its thinness can be bent at deep submillimeter scales to form metamaterials. It's highly reflective (88% on the bright side across the visible spectrum and even higher in the infrared) and conductive, rivaling copper. It resists corrosion for years in weather, it's nontoxic, it's light (2.71 g/cc), and it's *damn* cheap, under 50¢/m².

Robert Lang recommends laminating tissue paper on one or both sides of kitchen aluminum foil to make "tissue foil", which for years he considered the ideal origami material. Notably, he uses a weak sacrificial adhesive layer to hold the foil in place for the lamination process.

Typical alloys include especially 1100 and 1200, but also 8111, 8015, and 8006, with 0.06%–0.6% silicon and 0.4%–1.6% iron, and in some cases also some copper or manganese, under 0.5%. (1100 is sometimes described as an "unalloyed aluminum grade" but it's specified to contain 0.05%–0.20% of copper, and it unavoidably has other impurities.) Room-temperature yield strengths of these alloys range from 30–170 MPa, with ultimate tensile strengths of 70–200 MPa, and of course they all have a Young's modulus around 70 GPa. Because its crystal structure is fcc, it remains ductile down to absolute zero, making it suitable for cryogenic applications; indeed, aluminum becomes *stronger* at cryogenic temperatures. And, although it weakens dramatically at higher temperatures, it doesn't melt until almost 650°, enabling it to be used at higher temperatures than organic materials.

If oxidized (for example, with a soda solution, an arc, or anodization) it yields amorphous sapphire, which if crystallized is an excellent insulator, refractory, and abrasive. The oxidation process produces a great deal of heat, making aluminum a very-high-energy-density fuel, and, thanks to aluminum's sternly trivalent nature, electrical current; aluminum-foil fuel cells are routinely produced by amateurs, though these typically oxidize the aluminum to the chloride rather than the hydroxide or the oxide.

50¢/m² is 50¢/kWp in a solar concentrator, or 0.05¢/Wp, which is noticeably cheaper than photovoltaic cells, currently around 18¢/Wp, 360 times more expensive. (However, the foil number there is sunlight watts; if you're making a PV solar concentrator you have to divide by the efficiency of the solar cells, say 21%, which gives you 0.24¢/Wp electric.) A large aluminum-foil assembly would be vulnerable to significant deflections, but many small assemblies could

be placed on a hard, stable surface such as a rock or an adobe wall.

Alternatively, though, it might be possible to stiffen the foil by making the equivalent of corrugated cardboard out of it, maybe using aqueous boric acid (US$1.70/kg according to Potential local sources and prices of refractory materials (p. 566)) or borax as the glue. The surface tension of water is ample to hold aluminum foil in place until the water dries.

The feature that currently attracts my attention is the possibility of work-hardening, which suggests the tempting possibility of making tooling from aluminum foil that can itself work aluminum foil at room temperature, a possibility reinforced by the immense aspect ratios routinely available. As a simple example, you can in theory roll some foil into a cone, and the point of this cone can dent, form a rib in, or even pierce more of the same foil; but this is much easier in practice if you first fold the foil 16 layers thick, form ribs converging to a point on the last-formed fold, then roll the cone around that point. If the last-formed fold is reversed, the aluminum along the outer edge of the fold is the aluminum that was most strained previously, having been bent double with as small a radius as possible, and so will be the most work-hardened.

I was able to use such a cone to pierce not just aluminum foil but the skin of an apple. I folded it from some foil which, folded 256 layers thick, measured 2.57 mm in my shitty digital calipers; the resulting square measured 27–29 mm on each side and weighed 1.8 g, giving a density of only 0.8–1.0 g/cc, so it's probably more than half air, though it rapidly sinks in water, so probably the density is a *little* higher than that.

Using such a cone point to form ribs without piercing foil is tricky, because it tends to have significant asperities around the tip, which tend to tear the foil if it is unbacked. These can presumably be removed, permitting traditional SPIF processing of raw foil by sliding the point over the foil; a better alternative might be to produce a sequence of dents in the foil, then add new dents between them, eventually producing a continuous groove in a way analogous to how chain drilling cuts through a block of metal. However, when the foil "workpiece" is backed by something reasonably hard (I've used corrugated cardboard and the above-mentioned packed 256-layer aluminum-foil square) and I'm using one of the other point types described below, tears are relatively uncommon; in this situation it fairly reliably just forms ribs. (I need to test more rigorously to find out if the point type, the backing, or both is relevant here.)

Because such ribs are work-hardened, they are able to imprint their shape on fully-annealed foil repeatedly. I wrote a short word in cursive on foil using a layered aluminum-foil point ("single-point incremental forming"), with the foil simply backed by the somewhat-hard 256-layer square, then pressed this master against another piece of foil in several places, pressing the two foils between my fingers in each position ("stamping"). This resulted in very readable copies of the word in several locations, although I'm guessing there was substantial springback, so repeating this sort of stamping through multiple generations would make the stamping shallower at each generation.

I've tried smoking and annealing this foil with candle flames and butane lighter flames, but so far I've only managed to melt it (in under a second, usually) without ever smoking it. Maybe if I put water in it I could get it to smoke up so I could tell when it was on the point of overheating, but probably a different method of temperature control would be more practical to anneal such a thin material, such as a temperature-controlled heat gun.

A more reproducible point construction with a sharper, lower-volume point was able to pierce the foil and apple even more easily. I folded the foil three times to get 8 layers with a right-angle corner; bisected the corner twice to get a 22½° angle; formed a rib bisecting that angle with thumbnail pressure; then opened the final fold to about 30° so that the two sides of the point would stiffen one another.

By laying the foil into a form with a 90° valley in it and dragging such a point over it, I was able to get a bend into the foil. When there were ribs running perpendicular to the bend, this required multiple passes in one case; a second attempt resulted in neatly cutting through the foil at the intended bend location.

Another way to look at the 40 000:1 aspect ratio is to consider making a tight cylindrical roll from a strip of the foil, 400 mm long and, say, 10 mm wide, comprising 40 mm^3, a cylinder whose ends are 4 mm^3. The cylinder thus has radius 1.13 mm and diameter 2.26 mm, so a section through the center of it will go through 226 10-μm layers of foil. That is, instead of being 40 000 as you'd expect, it's about $\sqrt{(40000)} \cdot 4/\pi$.

The significance of ribs for folding is not that the ribs themselves become more flexible — the material in the rib is work-hardened and thus *less* flexible in plastic deformation, though its elastic properties remain unchanged — but that they prevent curvature of the material around them in any *other* direction, so if it's going to bend, the bend will be parallel to the ribs.

By making many parallel slits in the foil (with a steel box-cutter blade, backing the foil with cardboard), I was able to make expanded sheet metal, expanding a bit of foil by more than a factor of 2.

I was also able to fold a rather ugly origami crane by hand from the foil, about 700 mg and 70 mm wingspan.

This assemblage of techniques seems promising for matter compiler bootstrapping, although it's clearly just a beginning. Many of the problematic aspects of kitchen aluminum foil result from trying to work with it at the 10-mm scale rather than the 10-μm scale. Wrinkles, rips, and so on are going to happen unintentionally when trying to manipulate 10-μm foil with 10-mm human fingers.

(Also, the natural frequencies of such macroscopic objects made by folding such foil rarely exceed 100 Hz. The wing of the foil crane resonates at around 100 Hz.)

As a test of alternatives, I also folded an origami crane from a square cut from an aluminum Monster can, which is normally expected to be about 100 μm thick. The square was about 125 mm on a side, and the crane weighs about 3.8 g. One layer of the square measured 0.12 mm; two layers 0.33 mm; three layers 0.38 mm; and

four layers 0.52 mm. We can conclude from this that (a) my caliper technique is shitty, (b) the can (including paint) is about 120 μm thick, and (c) the actual aluminum part of the can is more like 90 μm thick (3.8 g / 125 mm / 125 mm / 2.71 (g/cc)).

It's a fucking miracle that I didn't cut myself on the damn crane. It was all knife edges and burrs, and every time I folded the damn thing it cracked and ripped more, exposing new cutting edges. Aluminum-can bodies are typically aluminum 3004, hardened with manganese and magnesium, and work-hardened from the deep-drawing process rather than annealed, so it's not a perfect analogy, but it seems at least suggestive.

Aluminum flashing for roofing is 0.024 inches, or in modern units, 610 μm, but I think it's annealed; aluminum is sold as sheet metal down to 0.004 inches, 100 μm in modern units.

If we figure that the foil can meaningfully change direction every 20 μm, then we might think of an aluminum-foil machine as being made of "moving parts" on the order of 1000 μm² (50 μm × 20 μm), 1000 "parts" per square millimeter of foil; a roll of kitchen aluminum foil is enough to fabricate some 4 billion "parts". A bootstrapping compiler might require 100 000 parts and thus a square centimeter of aluminum foil, cut and folded around into a shape a couple of millimeters in diameter. If it were doing only one thing at a time, and needed 10 seconds to construct/assemble each moving part, it would take about 12 days to recompile itself. This is probably adequately fast, barely, but probably not adequately robust against errors. It would probably be better to design it to have more parts and do many things at once, enabling it to be faster and correct errors.

It would be astonishing if *no* other materials were needed: you can't build anything electrical out of aluminum, at least at sub-microwave frequencies, because the whole device is at the same electrical potential. Similarly with getting mechanical power from thermal expansion and contraction: it would just expand isotropically rather than bending or sliding to do useful work. It might be possible to use just aluminum foil coated on one side with something else, such as glass or a few microns of aluminum oxide.

An interesting way to think of the density of aluminum foil is that 10 μm of 2.71-g/cc aluminum foil is 27.1 g/m², which is the same areal density as a 23-mm-high column of air.

Other processes that may be very interesting to apply to aluminum foil include electrolytic machining, electric discharge machining, scanning probe microscopy, and anodization. Electrolytic machining might make it possible to use an aluminum-foil tool to cut arbitrary shapes into metals such as steel, invar, brass, inconel, monel, or tungsten, and also to transform a scrap of aluminum foil (either flat or of a known geometry) into a white-light hologram of an arbitrary optical system, Fresnel-reflector-style.

Topics

- Pricing (p. 1147) (35 notes)

- Digital fabrication (p. 1149) (31 notes)
- Electrolysis (p. 1158) (18 notes)
- Experiment report (p. 1162) (14 notes)
- Strength of materials (p. 1164) (13 notes)
- Machining (p. 1165) (13 notes)
- Aluminum (p. 1180) (10 notes)
- ECM (p. 1186) (9 notes)
- Solar (p. 1203) (6 notes)
- Optics (p. 1209) (6 notes)
- Aluminum foil (p. 1237) (5 notes)
- Electropolishing (p. 1371) (2 notes)
- Origami
- Alloys

The nature of mathematical discourse

Kragen Javier Sitaker, 02021-05-27 (updated 02021-12-30)
(5 minutes)

As a kid I was always confused by the requirement to "show your work" on math tests, which is to say, demonstrate how you derived your answer. Why did it matter how I got the answer? What mattered was whether the answer was right or wrong, wasn't it?

This comes out of the street-fighting approach to math commonly taught in elementary schools, in which math is treated as a skill used to come up with answers to potentially difficult puzzles --- or, worse, merely a means to pass math tests. (And surely one motivation for demanding the "showing of work" is to reduce cheating on tests.) One alternative approach is to see math as a medium of creative expression, as explained in Lockhart's Lament, in which the tools and materials are abstract ideas rather than clay or paintbrushes. But another alternative is to see math as a form of argument, whose quality is to be judged by its convincingness and fallibility.

That is, although I could tell you that 48303 / 27 = 1789, even if you trust me, it may not be immediately obvious to you whether I am mistaken or not. If you are going to rely on this fact for some purpose, such that you will put yourself in a position to be harmed if it turns out to be false, you might want some sort of stronger assurance than merely my fallible assertion. And this is the objective of "showing your work" if you write down the partial sums:

```
   1789
  × 27
  -----
  12523
+ 3578
-------
  48303
```

This is an abbreviated notation for a syllogistic argument for the truth of my original assertion, which we could partly unpack as follows: 1789 × 7 = 12523; 1789 × 20 = 35780; 12523 + 35780 = 48303; therefore 48303 / 27 = 1789. (There are several other implicit premises as well, such as the distributive law of multiplication.)

Although it happens to be correct, this is not a very good argument, because each of the three premises I stated explicitly above is less than obvious. If I had written 1789 × 20 = 35870, for example, it might take you a while to spot the error. I claim that a principal objective of math is to *state arguments in such a way as to make any errors obvious*. Such an argument can be far more convincing: if it contains no obvious errors, then it contains no errors at all. Then, if its premises are correct, so is its conclusion, even if its author is untrustworthy.

I think this is a better argument for the same proposition: 1789 + 1789 = 3578; 3578 + 1789 = 5367; 53670 - 5367 = 48303; therefore

48303 / 27 = 1789. These calculations are simpler and so if there is an error in them it should be easier to spot, although perhaps the reasoning requires a little more explaining (30 - 3 = 27, so 30 × 1789 - 3 × 1789 = 1789 × 27).

In practice, though, I checked these calculations mostly by doing them with computer programs that I believe are unlikely to produce wrong answers, and it's common nowadays for people to use spreadsheets, cash registers, or pocket calculators for this purpose. Arguably, repeating a calculation a few times with different calculators is more trustworthy than mental checking. But there's still a great deal of potential for error in the process of invoking the calculator, as well as from hardware and software bugs.

This mathematical form of argument is the central ratchet that has allowed human knowledge to advance rather than falling backward over the last few centuries, because, just as money permits us to gain safety and sustenance by the efforts of not only honest hardworking folk but even treasonous cutpurses and greedy misers, math allows us to gain true and trustworthy knowledge of the universe from the reasoning of half-mad alchemists and deluded fools, because we can sift the occasional flake of gold from the mountains of superstitious dross they produced; by mathematical argument we can recognize a truth even when beset on all sides by nonsense, and often we can perfect a near-truth into a truth, and just as easily we can spot a flaw even in the sweetest honey of theory, dripping from the mouth of the finest of philosophers.

Unfortunately, at present we cannot do the analogous operation for results produced from a computer program rather than a mathematical formula. Often not enough information is published to even allow us to reproduce the published results by re-executing the program used by the original researcher; when such reproduction is possible, often the results diverge, and the error is quite frequently a different environment on the computer of the researcher attempting the reproduction, a situation more closely resembling chemistry than mathematics. Even if the results reproduce the original results correctly, they may well be due to a bug present in software installed on both computers.

Topics

- History (p. 1153) (24 notes)
- Math (p. 1173) (11 notes)
- Reproducibility (p. 1277) (3 notes)
- Ontology (p. 1350) (2 notes)
- Epistemology

Designing curiosity and dreaming into optimizing systems

Kragen Javier Sitaker, 02021-05-30 (updated 02021-12-30)
(6 minutes)

Watching a talk by Deepak Pathak about “Learning to Generalize Beyond Training” where he’s talking about helping reinforcement learners perform better in the world by making them do non-goal-directed exploration.

Pathak shows two photos, one of a toddler playing with her toy plane wearing goggles, another of a young woman standing in front of a fighter jet, and says, “In the real world, the reward could be delayed by days, months, or years, such that it’s hard to project back to where you are. So how does this child over here know how should she [sic] act to become pilot [sic] 20 years later? Does she optimize any reward and backprop all the way to her childhood? Well, not quite.” He cites Alison Gopnik’s work claiming that children are not driven by extrinsic goals, but by intrinsic curiosity. “Maybe by not giving goals to the agent you are making it not overfit to the task.”

(And that explains why people learn so poorly when motivated by extrinsic rewards, probably. They’re overfitting.)

He then outlines some approaches for choosing what actions a reinforcement learner should take to do “goal-free exploration”, which in a sense is experiment design.

Pathak actually cited a dozen papers on curiosity and intrinsic motivation (Poupart et al. 02006, Lopes et al. 02012, Bellemare et al. 02016, Oh et al. 02015, Tang et al. 02016, Ostrovski et al. 02017, Schmidhuber 01991, Schmidhuber 02010, Stadie et al. 02015, Houthooft et al. 02016, Mohamed et al. 02015, Gregor et al. 02017) and said that the originality of his approach was that his agent has no extrinsic goals at all.

It occurs to me that there’s probably some kind of way to bend gradient descent and its children to this task. If you have some kind of differentiable model (an ANN or whatever) of cause and effect in your world, you can use it to maximize a reward (and children do of course take goal-directed actions, for example to get food or to earn their parents’ approval) by using gradient descent to seek the optimal action: you compute the gradient of utility with respect to your vector of planned action parameters, then revise the plan to increase utility. And you can optimize it to be a better fit to your existing database of real-world experiences: you compute the gradient of prediction error with respect to your world-model parameters (ANN biases and weights or whatever), and adjust the parameters to decrease prediction error. So what does curiosity look like in this framework?

I think you can handle this with gradient descent as well. If you dream up scenarios in the world, for example by generating plausible predictions forward from some arbitrary state, then you can ask your model what will happen in those dreams, and perhaps in particular what actions would be best. In cases where your world model

provides very vague predictions (this may require that in some sense it gives you Bayesian probabilities) you can compute that you are ignorant, and you can use automatic differentiation to figure out which of the parameters of your world model are responsible for that ignorance --- dimensions in which your existing prediction error has a very small gradient, but there is a large gradient in the dream. Then, to be curious, you can try to create situations in the real world that you find unpredictable in the same way the dream was unpredictable, where there are plausible outcomes that maximize the magnitude of the resulting update in those ignorant parameters.

Or you could perhaps just choose actions whose results you cannot predict. But that might be more difficult: if you weight by utility, you will be choosing the actions that are the most unsafe, and if you don't weight at all, you will just be choosing the actions that will produce the brightest colors or loudest noises or most edges, whatever your input feature space is. So it might be best to leave the generation of unsafe scenarios to your dreams, then permit the dreams to inform you of which parameters of the world to be curious about so that you can design experiments to investigate them in a safe way.

This formulation of dreaming or fantasizing is vaguely similar to the concept of a generative adversarial network as a dreamer.

The phenomenon of "flow" suggests that there's a nonmonotonic aspect to intrinsic motivation in the humans: when prediction error is really high, they lose interest in the task (it's too hard) and when it's really low, they pay attention to other things (it's too easy). Pathak's formulation seems to lack that nonmonotonicity: his agents get more interested when things get more unpredictable.

Entropic formulations of the prediction problem (like, I think, about half the papers Pathak cites) offer a different candidate for the goal of curiosity: you want to improve your model of the world by reducing the entropy of the past, so that it now seems obvious that the things that happened were going to happen ("hindsight bias"). But does that mean you should look for parts of your model that are training slowly because their gradients with respect to your training data are very low, and try to do experiments that impact them? It seems that perhaps those are the parameters least likely to help you at re-encoding the past with lower entropy.

Topics

- Artificial neural networks (p. 1307) (3 notes)
- Mathematical optimization (p. 1348) (2 notes)
- Gradient descent (p. 1364) (2 notes)
- Dreaming (p. 1373) (2 notes)
- Flow

Omnidirectional wheels

Kragen Javier Sitaker, 02021-05-30 (updated 02021-12-30) (1 minute)

I was just watching a YouTube video by James Bruton of xrobots.tech on spherical wheels composed of independently rotating hemispheres in order to be able to roll in one direction controlled by a drive axle and a second direction freely; he built a vehicle with three such wheels that could move in any direction. In the center of each hemisphere he included an additional wheel, copied from a design at Osaka University. And it occurred to me: what about a toroidal wheel to be able to do the same trick?

Not such a novel idea, I guess, with the diagonal rollers. But what if the rollers around the edge of the wheel simply rotate perpendicular to the torus's axis? The torus can be driven on that axis and consist of four or more rollers around the outside whose axes of rotation form a regular polygon. Perhaps if there are many of them they can even overlap one another. Bruton's spheres contain sort of a degenerate version of this: the torus has been stripped of all but two rollers, and the rest of the torus is covered with the hemispheres.

Topics

- Contrivances (p. 1143) (45 notes)
- Mechanical (p. 1159) (17 notes)

Ghetto electrical discharge machining (EDM)

Kragen Javier Sitaker, 02021-05-31 (updated 02021-12-30) (5 minutes)

In theory you could cut any shape into steel, tungsten carbide, or diamond by electroerosion with nothing more than the tip of a copper wire or graphite point, a cup of diesel fuel, a capacitor or inductor sufficient to power an arc, and some way to supply power to it (originally, in 01943, a resistor). All you have to do is touch the wire to the steel (or other workpiece) wherever you don't want steel, and stir the dielectric around to sweep away the swarf. The trouble is just knowing where the cutting tip is, where steel is, and also that it's a slow process.

Nowadays you should be able to use the cutting tip itself to probe where the steel is, at least relative to the tip; a secondary tip operated at lower voltage and therefore not suffering from sparking can be used for touch-off to correct for tip wear. This should enable you to get an accurate picture of where the workpiece is and what remains to be cut, as well as the cutting rate of the spark parameters.

This kind of feedback should permit much higher material removal rates than are typical for wire EDM or die-sink EDM, because the feedback system compensates for the tool-electrode wear, permitting the use of much more aggressive sparks like those used in tap-burning EDM equipment.

To the extent that you can use a hollow needle electrode with dielectric squirting through it, you should be able to get higher cutting rates due to better swarf clearance.

Diesel fuel is cheaper than kerosene, lamp oil, vegetable oil, transformer oil, or laxative mineral oil, although any of these would also work, and these oils are much easier to keep in a dielectric state than water, which is very sensitive to slight amounts of ionic contamination. The oils only need to be circulated and have particulates strained out of them. Machines using water, even plain tap water, seem to have higher MRRs in practice, but I don't know why.

One inventor creating unorthodox devices in his shed reported success with die-sink EDM with an RC circuit with a 60V supply and a 400V electrolytic 100μF capacitor switched at 5kHz and a 10% duty cycle with an IRF IRG4PC40S IGBT. For his resistor he used a cartridge heater of "a few ohms". He used a water dielectric and got an MRR on aluminum of 18 mm³/minute (a 3-mm hole through a 1-mm aluminum sheet in 23 seconds) or 310 nl/s, and his brass tool electrode had visible pitting after even this brief use. This works out to a maximum of about 400 mJ per spark and theoretically up to about 1800 watts, but I suspect that in practice the power was much lower because he was getting more like 50 sparks per second than 5000.

Later he added an aquarium pump to recirculate the water through

a paper coffee filter, switched to a lower-ESR 300V 160μF electrolytic, and added a servomechanism which simply feeds the electrode down when the current spikes of the sparks didn't reach a preset set point. At times, his aquarium-pump setup squirts water *onto* the cut rather than immersing the whole apparatus in water. With an additional patch to lift the electrode periodically to improve flushing, he was able to cut steel.

Another machinist reported similar success on aluminum, but used a tubular brass "die" tool electrode to cut his round hole. A third amateur used diesel fuel as his dielectric and reported very slow cutting on steel, limiting his current with the series resistance of a quartz-halogen worklight at 120VDC.

A much simpler approach uses the traditional solenoid buzzer circuit, with the sparking contacts being the workpiece and tool and the solenoid itself holding the energy of the spark, powered from merely 24VDC 1.8A. Thus the servo, energy-storage, and feed mechanisms are all provided by a single simple mechanism. He reported extremely slow cutting and applied a light oil dielectric occasionally by hand; I think it was slow because his cut was getting filled up with swarf.

A review of a Luoyang Xincheng SFX-4000B tap remover reported burning through a 50-mm-long broken tap in 8 minutes with about 10 amps at about 40 volts (average, not peak), using a brass electrode in tap water. My best guess is that it was making a 2 mm cut through a 4 mm wide tap flute, so about 400 mm^3, or 50 mm^3/minute. The wear on the brass tool electrode was visually about 15% of the wear on the hardened steel workpiece.

Topics

- Materials (p. 1138) (59 notes)
- Pricing (p. 1147) (35 notes)
- Manufacturing (p. 1151) (29 notes)
- Ghettobotics (p. 1169) (12 notes)
- Electrical discharge machining (EDM)

Broken hard disks are the cheapest source of ultraprecision components

Kragen Javier Sitaker, 02021-06-02 (updated 02021-06-12)
(3 minutes)

Ooh, broken disks are AR$1200 (US$8) for a ten-pack. $120 (80¢) (or US$1 in the EU in 02013) gets you a voice-coil actuator built around two pyrophoric neodymium magnets plated in nickel mounted on permalloy or mu-metal brackets, a 5400rpm long-life bearing, and a BLDC motor with a controller, plus some extremely flat (Ra 120 pm) first-surface glass-ceramic mirrors with 80% reflectivity and a low thermal coefficient of expansion (7.4 ppm/° for TS-10) and high modulus (100 GPa), about 400g of castable aluminum between possible platters and the case (maybe A380 or ADC12 or 6061 or 5052) some Torx screws, a SATA power connector, some machined 6061-T6 aluminum spacers, and some jumper blocks. Oh and maybe an accelerometer and temperature and pressure sensors. As scrap metal this totals about US$1.66, mostly from copper and gold from the PCB, but obviously you can't get voice-coil actuators or bearings that cheap.

Oh, and an ARM core accessible over JTAG, using external RAM and Flash. And it talks to the controller that controls the spindle and head over SPI. The Flash format has been largely reversed, but the chipmakers don't publish datasheets.

The whole precisely balanced platter assembly with the platters, motor, and bearings is nowadays almost invariably 5400rpm or 7200rpm; 10krpm disks are exotic rarities, and 3600rpm disks are antique. A 7200-rpm 3.5" desktop drive has a rim speed of 5.3 m/s, while a 5400-rpm 2.5" laptop drive is 2.9 m/s. Either of these is a fairly respectable speed on its own for things like fine grinding, but also you can also overclock them quite a bit. They're normally only operated at about 3 watts, so you probably can't get more than 30 watts out of them no matter how much cooling you add.

The motor might be suitable as a hand pullstring generator.

Desktop disks still use aluminum platters, typically 635 μm thick. These are also 6061.

For small machinery the cobalt alloy used for the magnetic medium might be worth extracting and refining. I'm guessing it's on the order of 10 mg per disk, assuming 25 nm thickness, 3 platters, and 3.5", and ignoring the center hole; there are also two layers on top of it and three underneath.

Reputedly the head arm bearing is also very high precision. Micah Elizabeth Scott built a 4kpps laser projector in 02008 using two voice-coil actuators from hard disks.

Not sure how to use the actual GMR sensors themselves; presumably they can detect small magnetic fields reliably.

Topics

- Contrivances (p. 1143) (45 notes)
- Electronics (p. 1145) (39 notes)
- Pricing (p. 1147) (35 notes)
- Frrickin' lasers! (p. 1168) (12 notes)
- Ghettobotics (p. 1169) (12 notes)
- Precision (p. 1183) (9 notes)

Micro impact driver

Kragen Javier Sitaker, 02021-06-02 (updated 02021-06-12)
(2 minutes)

I watched a Marco Reps video last night when I should have been working and was surprised by his overpowered screwdrivers with goofy animated OLED screens and 1500-watt quadcopter motors and the like. It led me to thinking about what you could use the modern cheapness of control electronics for in hand tools.

One semi-obvious example is impact drivers, which avoid disastrous screwdriver camout by simultaneously exerting force axially "down" onto the screw. Traditionally these are operated manually with a hammer, but there's no particular reason not to operate them internally with a spring, like an automatic center punch. And if you have electrical power, you could power it internally, like a hammer drill or rotary hammer.

In an impact driver it may be necessary for the axial force to be large: it must be larger than the camout force generated by the inclined planes, and the rotary force normally needs to be large enough to overcome the screw's stiction, which is possibly larger than normal due to the axial force (so increasing the axial force without bound makes things worse). However, the requirement on the impact *energy* is minimal: it needs to be enough to overcome stiction. But you could imagine that, at least for some screws, a very large number of impacts with very high peak force but low energy would work adequately. In the limit you're approaching ultrasonic sonication of the screw as you twist it, making this a real-life "sonic screwdriver".

Topics

- Contrivances (p. 1143) (45 notes)
- Electronics (p. 1145) (39 notes)
- Mechanical (p. 1159) (17 notes)
- Pulsed machinery (p. 1167) (12 notes)
- Frrickin' lasers! (p. 1168) (12 notes)
- Hand tools (p. 1197) (7 notes)
- Sonic screwdrivers (p. 1324) (2 notes)

Minkowski deconvolution

Kragen Javier Sitaker, 02021-06-02 (updated 02021-12-30)
(6 minutes)

Suppose you are moving a tool of some unknown, but static, geometry around a workpiece that also has unknown and static geometry, and you have a thing rigged up that stops motion when they touch, preventing damage, and tells you. And you have a high-resolution positional feedback system that tells you what the position of the tool is, relative to its starting point; and there are no other important sources of motion or unknown geometry in the system. What can you learn from this?

Well, if you have only one degree of freedom, you can tell the initial distance, or angle or whatever, from the tool to the workpiece. Or the distance at any later time, which comes to the same thing when you always know how far you've come.

If you have two translational degrees of freedom, you can trace out a curve in the plane that is sort of the Minkowski sum of the tool and the workpiece; if the point of the tool is not too large, this is an approximation of the workpiece's shape. In particular, if the tool is convex, you will never see tighter convex curvature on this workpiece approximation than the tightest curvature on the tool. This can allow you to bound the error due to the tool being non-pointlike.

In three translational degrees of freedom a similar property holds, but now you're tracing out a Minkowski-sum surface rather than just a curve.

But how can you separate out the contributions from the tool and the workpiece?

One piece of information comes from recognizing motifs in the surface: every point and edge in the workpiece surface manifests as a copy of part of the tooltip shape, like the camera bokeh in a photo, but the Minkowski sum is more similar to grayscale morphological dilation than like the convolution with a bokeh. So in theory from a purely information-theoretical perspective you ought to be able to recognize that these repeated motifs are generated by the tooltip and infer enough about the tooltip shape to give you a more random-looking workpiece landscape, though with holes in it where the tooltip didn't fit. But another approach comes from having more degrees of freedom.

Suppose you have redundant degrees of freedom. The simplest example here is having two translational degrees of freedom, plus the ability to rotate the tool around the axis perpendicular to them. This enables you to sort of measure the shape of the tool, by rotating it and measuring it against the same position on the workpiece. If you're measuring it against a flat on the workpiece, you can use this to find the shape of the convex hull, though only up to some constant radius offset. If you measure against a needlelike point on the workpiece, you can use it to find the shape of the tool to high accuracy, again up to some constant radius offset. Combining this with the workpiece

motifs, which tell you how big the radius should actually be (at least if the workpiece has a few asperities on it somewhere), you should be able to infer quite precisely what the tooltip shape is, and thus a great deal about the workpiece shape.

This extends into three translational dimensions as well, if you have two separate rotational degrees of freedom.

A sort of intermediate case here is where you don't have four or five degrees of freedom, but the degrees of freedom you do have admit multiple solutions to inverse-kinematics problems. Consider an XZC* setup with a tool that can scan back and forth past the center of a turntable, and can also be raised and lowered. This only has three degrees of freedom, only enough to bring the end effector into contact with any position on or in the workpiece, but for every point that isn't on the center axis of the turntable, you can establish contact in two separate ways, with relative orientations 180° apart. These two orientations give two separate Minkowski-sum surfaces; the difference between them tells you something about the difference between the tooltip and a half-turn-rotated version of the tooltip.

This isn't necessarily enough information to tell you anything interesting; it's impossible to distinguish any two half-turn-symmetric tooltips that are capable of producing the same Minkowski-sum surface. By contrast, a redundant rotational degree of freedom enables you to distinguish any two tooltips that differ by anything other than a circular dilation, or in the 3-D case, a cylindrical or toroidal dilation.

This is a particularly interesting problem to me because, when you're cutting or forming any kind of workpiece, there is usually tool wear, and avoiding tool wear requires tradeoffs that may be unappealing for other reasons, like lower material removal rate or more expensive tool materials. So to the extent that it's possible to automatically and continuously compensate for that, it may be an extremely worthwhile ability to develop, providing order-of-magnitude advantages in speed/precision tradeoffs. Probably the right way to do this is to start by cutting or forming one or more reference points on the workpiece to have favorable geometry in a way that is resilient to tooltip shape errors, such as a sharp-edged circular hole or a sharp cone, and then using those reference points periodically thereafter to measure the tooltip. The reference points can be eliminated at the end of the process, or they can be placed on things that aren't part of the workpiece proper (but are rigidly fixed in relation to it), or they can be placed in places where their shape doesn't matter.

* The X and Z axes can also be rotational rather than purely translational; it makes no difference in this case.

Topics

- Math (p. 1173) (11 notes)
- Precision (p. 1183) (9 notes)
- Sensors (p. 1191) (8 notes)
- Scanning probe microscopy (p. 1242) (4 notes)

Greek operating systems

Kragen Javier Sitaker, 02021-06-04 (updated 02021-06-12)
(4 minutes)

Because it's the most common masculine singular nominative suffix, Greek is full of nouns and adjectives that end in "-os" ("-ος"), including many people's names, although in many cases English got them by way of Latin, which usually substituted its own masculine singular nominative suffix, "-us". And there's also an "-ιος" suffix (I think that's the singular genitive of the same declension?) which gives us names like "Apollonios" and "Dionysios" and Hephaestus's epithet "Aitnaios", as well as highly productive suffixes like "-ασσος", for places, and "-ισμος", for procedures or beliefs. Of course, to a programmer, the "-os" suffix sounds like an operating system, like MacOS, AmigaOS, MS-DOS, RISC OS, iOS, IOS, BeOS, GEOS, GECOS, SunOS, Unicos, ReactOS, FreeDOS, TempleOS, KeyKOS, Palm OS, ChibiOS, and the like, and there is a long tradition of naming computer systems after figures from Greek mythology, such as Kerberos and Project Athena.

So multilingual puns seem to be called for.

This is not an entirely new idea; CDC named an OS "Kronos", there was an MS-DOS alternative called "THEOS", of course EROS, and there has probably been more than one OS called Cosmos , DEMOS/ДЕМОС, and at least three named Minos.

Adding to the potential for multilingual puns, masculine *plurals* of nouns and adjectives in Spanish and Portuguese *also* commonly end in "-os", and "Unicos" and "AROS" are already Spanish (and Portuguese) words, meaning "only" and "rings/earrings" respectively; Liddell and Scott additionally define ἄρος as "use, profit, help". Too bad it's already taken!

But we could imagine, for example, the cult of HefaistOS AitnaiOS in LemnOS, OdysseOS, DionysOS and his thiasOS and SeilenOS drinking from a cantharOS, carrying a thyrsOS (with which fire was stolen) and suffering a sparagmOS, hermaphroditOS, FalOS, OceanOS, TyrnavOS, PriapOS, GlaucOS, LycurgOS, ScyrOS, KotylOS, AscOS, AryballOS, CyprOS, TheofrastOS, DiodorOS Siceliotes, HerodotOS, FlaviOS PorfyrogenitOS, HalicarnassOS, HomerOS, ChiOS, ChaOS, EustachiOS, DemetriOS, ParnassOS, PeisistratOS, SamOS, LogOS, StrategOS, HyperbolOS, PeloponnesOS, ArgOS Panoptes, HeliOS, Hermes TrismegistOS, AtreOS, Eos, OrpheOS, TartarOS, BriareOS, AnytOS, IapetOS, OlympOS, ApollodorOS, OuranOS, KratOS, TheOS ProterOS, FolOS, ThespiOS, ErymanthOS, PithOS, PylOS, KnossOS, AngelOS, TyphOS or TyphoiOS, AischylOS, PatroclOS, FaidrOS, OrthOS (which is "assholes" in Spanish, though misspelled), KadmOS, AigyptOS, NarkissOS, IkarOS, TalOS the gift of HefaistOS KholOS to MinOS, KokalOS who gave DaidalOS refuge in KamikOS from MinOS, PegasOS, OrfeOS, AisculapiOS, MinotaurOS, CetOS, AgriOS, EnkeladOS, HippolytOS, SisyphOS, NessOS the son of KentaurOS who slew Herakles the lover of IolaOS and slayer of AntaiOS, AkhaiOS, NostOS, LykourgOS the

law-giver, LesbOS, BarbarOS (which in colloquial Argentine Spanish meaning “excellent”), DemosiOS (“public”, of the δημος), HeraklitOS, OstrakismOS, ByblOS, PapyrOS, BybliOS, DiOS (“of Zeus”), and so on.

Several other promising names turn out to not end in “-ος” in Greek: Prometheus and Theseus, for example.

Really, though, you can probably find a Greek name ending in “-ος” for any concept you care to name an OS after.

Topics

- Programming (p. 1141) (49 notes)
- Humor (p. 1292) (3 notes)
- Puns

The algebra of N-ary relations

Kragen Javier Sitaker, 02021-06-14 (updated 02021-07-27)
(4 minutes)

(Based on Jamie Brandon's Imp and some unpublished work of Dave Long's based on Hehner's *Practical Theory of Programming*.)

Consider N-ary relations like those of the relational algebra, treated as sets of equal-arity tuples. (Maybe they could be bags instead of sets; I'm not sure.) It's convenient to assume some set of atomic items to make these tuples out of, such as words, symbols, or numbers. We can treat an atom as a degenerate relation: `a` is taken to be the relation consisting of a single tuple containing the atom `a`.

From these atoms we can build nonempty single-column relations with a union operator `+`: `a+b+d` is a single-column relation consisting of three one-item tuples `a`, `b`, and `d`, in no particular order, so it's equal to `a+d+b` or `d+b+a`. (If we rule out multisets, it's also idempotent, so `a+b+d+a+b+d` is also equal.)

By adding a cartesian-product operator with higher precedence, written simply with juxtaposition, we can build multiple-column relations: `x y z` is a three-column relation consisting of a single tuple with the atoms `x`, `y`, and `z`, in that order. This operator is associative, so `(x y) z` is equal to `x (y z)`, but not commutative; the order of columns matters, so, for example, `y z` ≠ `z y`, and `x y z` ≠ `x z y`.

We give the implied operator of juxtaposition cartesian-product semantics simply by declaring that it distributes over `+`, so `(a + b) (c + d)` = `a (c + d) + b (c + d)` = `a c + a d + b c + b d`. This also gives us a normalization procedure for relations built up in this way, which makes equivalence decidable.

Now we can introduce identity elements for these two operations; capitalizing the names from Imp, we can declare that None is a relation with no rows, and Some is a relation containing only the empty tuple. (A different choice of notation might use {} for None and () for Some, or α for Some and ω for None.) For any relation X, `None + X` = `X`, and `Some X` = `X Some` = `X`. I think it's provable from this that `None X` = `X None` = `None`, but if not, let's postulate it — it's obviously necessary for juxtaposition to give us the usual kind of cartesian product.

We've said that we're interested in sets of *equal-arity* tuples, but there's nothing stopping us from writing `a + b c`, though that has a straightforward interpretation as a set containing a 1-tuple and a 2-tuple. For the time being we'll just consider such expressions as being uninteresting due to being "ill-typed", but clearly enough if we leave them in with that interpretation, we have the Kleene-closure-free regular expressions.

This is nearly a Boolean algebra, with juxtaposition for ∧, `+` for ∨, None for 0, and Some for 1; but the ∧ of a Boolean algebra must be commutative, and it's not clear to me how to define ¬ such that $a \vee \neg a = 1$ and $a \wedge \neg a = 0$ (`X + !X = Some` and `X !X = None`).

It *is* a semiring, though, at least if we sweep the "typing" problem under the rug; we have associativity of both operators,

commutativity of +, distributivity, identity elements, and annihilation. In fact, it's pretty much just the free semiring on whatever our atoms are. If we rule out multisets, it's the free idempotent semiring, which induces a partial order on the operations.

The two operators above are two of the five primitive operators of Codd's relational algebra; the other three are selection, projection, and set difference (set intersection being derived from union and difference).

Topics

- Programming (p. 1141) (49 notes)
- Math (p. 1173) (11 notes)
- Program calculator (p. 1246) (4 notes)
- Kleene algebras (p. 1287) (3 notes)
- Databases (p. 1376) (2 notes)

Nuclear energy is the Amiga of energy sources

Kragen Javier Sitaker, 02021-06-14 (updated 02021-07-27) (3 minutes)

Nuclear energy is the Amiga of energy sources.

Ahead of its time, it was unjustly rejected and persecuted by the ignorant masses. Its advocates are bonded by the quiet pride that at least they weren't unthinkingly siding with those masses. (And they're right!) Meanwhile, as the Amiga stagnated for terribly unfair reasons, other, scrappier technologies like the i386 and UMG-Si grew from being worthless boondoggles (except in special circumstances, like spaceflight) to being actually far better and cheaper. But the Amiga advocates keep the faith, sharing their suffering and resentment. They inevitably try the alternatives a little and perhaps even start to like them. Gradually their denial recedes, decade by decade.

But they know that however much fab costs go down and leave their beloved Amiga behind in the dust, you'll never be able to run nuclear submarines and Antarctic research stations on solar panels.

— ⁂ —

Not all nuclear-energy advocates are so unversed in the basics of energy as to say incoherent things like "replacing daily energy consumption from crude oil will require 14.5 terawatts per day" or pants-on-head things like "renewable mandates push up electricity prices" (https://freopp.org/why-nuclear-power-not-renewables-is-the-path-to-low-carbon-energy-part-1-c0b66d4b9570) but today they are all suffering from serious fact deficiencies.

— ⁂ —

Wind, where available, undercut the cost of steam power (including nuclear and coal) a decade ago, and PV undercut it in equatorial parts of the world about four years ago, or in even more of the world if you don't include storage. As a result, last year, China, whose electrical consumption has doubled in the last decade, built 48.2 gigawatts† of new photovoltaic capacity last year https://www.reuters.com/article/us-china-energy-climatechange-idUSKBN29Q0JT but only has, I think, something like 10 GW of nuclear plants under construction, scheduled to come online over the next several years. PV installed capacity in China is growing by 23% per year, the same rate it has been growing worldwide for the last few years; with some luck that will return to the 39%-yearly-worldwide-growth trend that has been the fairly consistent average over the last 28 years.‡

(Previous versions of this comment were posted at https://news.ycombinator.com/item?id=26218673, https://news.ycombinator.com/item?id=26674832, and https://news.ycombinator.com/item?id=27503120.)

† China's PV capacity factor seems to be only about 13%, so those

48 GWp probably work out to only about 6 GW average. It would be nice if China managed to site its new PV plants in places that could provide a capacity factor like California's 28%.

‡ Why 28? Because I haven't found figures yet on what worldwide installed capacity was in 01992 or earlier.

Topics

• Energy (p. 1170) (12 notes)
• Humor (p. 1292) (3 notes)

Flux-gate downconversion in a loopstick antenna?

Kragen Javier Sitaker, 02021-06-15 (updated 02021-07-27)
(2 minutes)

Lower radio frequencies like AM radio are typically received with ferrite loopstick antennas rather than half-wave dipoles or similar, because the dimensions of an efficient dipole are totally impractical for a human-scale system. It occurred to me that if you saturate the loopstick with a magnetic field at a different frequency, you could use it as a magamp that does downconversion, either for downconversion to IF or baseband — in a sense, using the ferrite as a flux-gate magnetometer to measure the magnetic component of the radio waves at submicrosecond intervals.

Mediumwave AM broadcasting is 530 kHz to 1700 kHz, so you couldn't just use a single fixed frequency to downconvert to IF — you could imagine using a 400 kHz LO to get 130–1200 kHz, say, but that's useless. If you tried saturating the ferrite with a square wave at the frequency of the desired station, you might get half-wave synchronous rectification of the frequency without so much as a diode, or you might get nothing if you were in quadrature. So you'd probably need some kind of second-order PLL or something to keep your phase in sync, and it would have to tolerate the intermittent nature of AM. This is probably more complexity than the usual tuning circuits, so there may be no real advantage to this design at all.

There are magnetic core materials that work well up to about 10 MHz.

Topics

- Contrivances (p. 1143) (45 notes)
- Electronics (p. 1145) (39 notes)
- Radio (p. 1278) (3 notes)

PEG-like flexibility for parsing right-to-left?

Kragen Javier Sitaker, 02021-06-16 (updated 02021-07-27)
(2 minutes)

Looking at PDF, whose syntax derives from PostScript, I'm struck by the fact that a lot of its constructs are most easily parsed right to left. An object reference like `16 0 R` starts with an integer object, and so you end up having to backtrack if you're using a PEG parser.

If instead you parse it right to left, you don't have this problem as much; the R announces that you are looking at an object reference, and it contains the following two integers. Similarly, in a content stream, you may encounter a text object like Example 1 from §9.2.2 of the PDF 1.7 spec, ISO 32000-1:2008:

```
BT
  /F13 12 Tf
  288 720 Td
  (ABC) Tj
ET
```

As an S-expression, this is `(text (font 'F13 12) (pos 288 720) (paint "ABC"))`; `Tf` is the operator that sets the font, `Td` is the operator that sets the position, and `Tj` is an operator that draws text. Reading this backwards, it's trivial to predict what you're going to have to parse; reading it forwards, you either need to maintain a stack of pending values like 288 and 720, or do a lot of backtracking.

However, individual tokens in here are more easily read forwards; `/F13` looks like an integer if you look at its last 0, 1, or 2 characters, and perhaps like an operator if you only look at its last 3.

Is there a way to get PEG-like memoization for an asymptotic performance guarantee and flexibility, while maintaining a LALR-like stack of bottom-up items that can tell us what parses to even attempt? Is that even a meaningful question?

Topics

- Programming (p. 1141) (49 notes)
- Algorithms (p. 1163) (14 notes)
- The Portable Document Format (PDF) (p. 1227) (5 notes)
- Parsing (p. 1228) (5 notes)
- Parsing expression grammars (PEGs) (p. 1343) (2 notes)

How little code can a filesystem be?

Kragen Javier Sitaker, 02021-06-16 (updated 02021-07-27) (1 minute)

What's the simplest way to support the basic Unix filesystem interface? open, close, read, write, mkdir, chdir, lseek, and fstat, supporting append, random-read, random-write, and random-readwrite modes. If you were willing to sacrifice efficiency for simplicity.

You need file-descriptor objects with offsets and modes, and then the actual directories, and a CWD for your filesystem cursor.

Probably the simplest solution in a garbage-collected language is to make a mutable tree of strings in memory with associated metadata. fstat sort of requires that you keep track of the modification date. Then you might want to be able to serialize this in-memory filesystem, or maybe an incremental update to it. inode numbers are potentially a bit tricky, but they don't have to be assigned sequentially.

One alternative is to use an *immutable* tree and update it functionally, which potentially simplifies the incremental-update logic.

This feels like it ought to be doable in about 200 lines of code in a garbage-collected language.

Topics

- Programming (p. 1141) (49 notes)
- Bootstrapping (p. 1171) (12 notes)
- Small is beautiful (p. 1190) (8 notes)
- Independence (p. 1215) (6 notes)
- Operating systems (p. 1248) (4 notes)
- Unix (p. 1268) (3 notes)
- Filesystems

Notes on the PDF file format

Kragen Javier Sitaker, 02021-06-16 (updated 02021-07-27)
(15 minutes)

I'm reading through the ISO 32000-2008 PDF-1.7 spec, which is about 340,326 words, 60% of the size of War and Peace. But for the time being I'm not interested in all of it:

- §1 Scope (1 p.), yes.
- §2 Conformance (1 p.), yes.
- §3 Normative references (4 pp.), yes.
- §4 Terms and Definitions (4 pp.), yes.
- §5 Notation (1 p.), yes.
- §6 Version designations (1 p.), yes.
- §7 Syntax (100 pp.), yes.
- §8 Graphics (127 pp.), no.
- §9 Text (59 pp.), yes.
- §10 Rendering (24 pp.), no.
- §11 Transparency (42 pp.), no.
- §12 Interactive Features (124 pp.), no.
- §13 Multimedia Features (61 pp.), no.
- §14 Document Interchange (96 pp.), I don't think so.

This works out to only about (+ 1 1 4 4 1 1 100 59) = 171 pages of reading. I don't think I'm going to be able to make it through the whole thing in the next couple of hours...

It's interesting that on p. 251 in §9.4.3 it requires you to backslash all of your special characters in the string, with no provision for nesting parens:

> The strings shall conform to the syntax for string objects. When a string is written by enclosing the data in parentheses, bytes whose values are equal to those of the ASCII characters LEFT PARENTHESIS (28h), RIGHT PARENTHESIS (29h), and REVERSE SOLIDUS (5Ch) (backslash) shall be preceded by a REVERSE SOLIDUS) character. All other byte values between 0 and 255 may be used in a string object. These rules apply to each individual byte in a string object, whether the string is interpreted by the text-showing operators as single-byte or multiple-byte character codes.

I think this is an error because §7.3.4.2 on p. 23 says:

> Any characters may appear in a string except unbalanced parentheses (LEFT PARENHESIS [sic] (28h) and RIGHT PARENTHESIS (29h)) and the backslash (REVERSE SOLIDUS (5Ch)), which shall be treated specially as described in this sub-clause. Balanced pairs of parentheses within a string require no special treatment.

It's surprising to see that "name objects" are apparently new in PDF 1.2:

> Beginning with PDF 1.2 a name object is an atomic symbol uniquely defined by a sequence of any characters (8-bit values) except null (character code 0).

But maybe that isn't really what is meant; maybe they existed previously but could include null or couldn't include, say, DEL.

It's a relief to see that names are interpreted as UTF-8.

Strings and character encodings

The encoding of string contents is tricky. §7.9.2.2 says they're PDFDocEncoded unless they begin with a BOM. But that's only for "structural" strings, not for strings that are part of document content. Actual text strings on the page are decoded by the font:

> With a composite font (PDF 1.2), multiple-byte codes may be used to select glyphs. In this instance, one or more consecutive bytes of the string shall be treated as a single character code. The code lengths and the mappings from codes to glyph are defined in a data structure called a *CMap*, described in [§]9.7, "Composite Fonts".

It isn't really described there. §9.7.5.3 explains that CMaps are *really* described in "Adobe Technical Note #5014, Adobe CMap and CIDFont Files Specification." Although it does give an example CMap that implements Shift-JIS, which is evidently written in PostScript, and there's some further explanation in §9.7.6.2, but it assumes you're already familiar with the aforementioned TN5014. §9.10.3 also suggests reading "Adobe Technical Note #5411, ToUnicode Mapping File Tutorial."

TN#5014 explains further:

> Some CID-keyed font rendering software (such as ATM-J) takes advantage of a particular stylized use of the PostScript language. As a result, CID-keyed font files must also adhere to these PostScript language usage conventions. The syntax resulting from these conventions is considerably more restricted than that of the PostScript language; CID-keyed fonts can be read and executed by PostScript interpreters, but not all PostScript language usage is acceptable in CID-keyed fonts.

Its §5 and §7 explain the CMap in more detail; TN#5014§5 gives what looks like a slightly less abbreviated version of the Shift-JIS CMap given as an example in the PDF spec. The most crucial information is on TN#5014 p. 51:

> the cidrange sections associate the beginning and ending of a range of acceptable character codes, expressed as hexadecimal strings, with the starting CID for that range. ...

```
100 begincidrange
   <20>      <7e>    1
   <8140> <817e> 633
   <8180> <81ac> 696
...
endcidrange
```

Evidently this means that the byte sequence 0x20 maps to CID 1, 0x21 to CID 2, ... 0x7e to CID 95, then 0x81 0x40 to CID 633, 0x81 0x41 to CID 634, etc. Evidently 0x81 0x7f is an invalid sequence in Shift-JIS, and Wikipedia agrees that it is.

Also there are some predefined CMap names, given as the "/Encoding" of a Type0 font, including /Identity-H (which is UTF-16BE for horizontal text) and some UTF-16BE and UCS-2 cases.

The /Type0 font can include, in addition to an /Encoding, a /ToUnicode which points at another CMap which tells how to convert to Unicode rather than indexes into some font. The example given maps the ASCII range, unpacks some ligatures with "basefont ranges", and maps a single character to a surrogate pair with a "basefont char":

```
2 beginbfrange
<0000> <005E> <0020>
<005F> <0061> [<00660066> <00660069> <00660066006C>]
endbfrange
1 beginbfchar
<3A51> <D840DC3E>
endbfchar
```

Nobody ever uses a predefined CMap for /ToUnicode. And they always compress their CMaps. I extracted one of these content streams to a file and read it with Python's zlib.decompress; evidently it was set up using a super dumb ASCII subsetting procedure:

```
>>> print zlib.decompress(open('tmp.flate').read())
/CIDInit /ProcSet findresource begin 12 dict begin begincmap /CIDSystemInfo <<
/Registry (F3+0) /Ordering (T1UV) /Supplement 0 >> def
/CMapName /F3+0 def
/CMapType 2 def
1 begincodespacerange <20> <78> endcodespacerange
6 beginbfchar
<20> <0020>
<2a> <002A>
<2e> <002E>
<41> <0041>
<59> <0059>
<61> <0061>
endbfchar
7 beginbfrange
<31> <35> <0031>
<43> <49> <0043>
<4b> <50> <004B>
<52> <56> <0052>
<63> <69> <0063>
<6d> <6f> <006D>
<72> <78> <0072>
endbfrange
endcmap CMapName currentdict /CMap defineresource pop end end
```

Here's another case from another file that's not quite so innocent:

```
/CIDInit /ProcSet findresource begin 12 dict begin begincmap /CIDSystemInfo <<
/Registry (NDBBAF+ArialMT+0) /Ordering (T42UV) /Supplement 0 >> def
/CMapName /NDBBAF+ArialMT+0 def
1 begincodespacerange <0114> <012a> endcodespacerange
2 beginbfrange
<0114> <0114> <0144>
<012a> <012a> <017C>
endbfrange
endcmap CMapName currentdict /CMap defineresource pop end end
```

A nicer case is this one, from a third PDF file:

```
/CIDInit /ProcSet findresource begin
12 dict begin
```

```
begincmap
/CIDSystemInfo <<
/Registry (Adobe)
/Ordering (UCS)
/Supplement 0
>> def
/CMapName /Adobe-Identity-UCS def
/CMapType 2 def
1 begincodespacerange
<0000> <FFFF>
endcodespacerange
1 beginbfrange
<0000> <FFFF> <0000>
endbfrange
endcmap
CMapName currentdict /CMap defineresource pop
end
end
```

I think this doesn't comply with the explanation of `beginbfrange` from the PDF spec:

EXAMPLE 2 in this sub-clause illustrates several extensions to the way destination values may be defined. To support mappings from a source code to a string of destination codes, this extension has been made to the ranges defined after a beginbfchar operator:

```
n beginbfchar
srcCode dstString
endbfchar
```

where dstString may be a string of up to 512 bytes. Likewise, mappings after the beginbfrange operator may be defined as:

```
n beginbfrange
srcCode1 srcCode2 dstString
endbfrange
```

In this case, the last byte of the string shall be incremented for each consecutive code in the source code range.

When defining ranges of this type, the value of the last byte in the string shall be less than or equal to 255 - (srcCode2 - srcCode1). This ensures that the last byte of the string shall not be incremented past 255; otherwise, the result of mapping is undefined.

But evidently in this case the intent is to increment both of the bytes of dstString, not just the last one.

But /Adobe-Identity-UCS isn't always so nice. Here's another one, from another file:

```
/CIDInit /ProcSet findresource begin
12 dict begin
begincmap
/CIDSystemInfo
<<  /Registry (Adobe)
/Ordering (UCS)
/Supplement 0
>> def
```

```
/CMapName /Adobe-Identity-UCS def
/CMapType 2 def
1 begincodespacerange
<0001> <046D>
endcodespacerange
10 beginbfchar
<005F> <007C>
<0061> <007E>
<0070> <00E9>
<0085> <00A3>
<0087> <2022>
<00A9> <00AB>
<00AA> <00BB>
<00AB> <2026>
<00C2> <2219>
<013C> <2033>
endbfchar
7 beginbfrange
<0003> <0004> <0020>
<0006> <003E> <0023>
<0040> <0042> <005D>
<0044> <005D> <0061>
<00B1> <00B2> <2013>
<00B3> <00B4> <201C>
<00B5> <00B6> <2018>
endbfrange
endcmap
CMapName currentdict /CMap defineresource pop
end
end
```

Strangely enough nobody seems to include the DSC comments in their embedded CMaps.

U+2022 is a bullet, U+2026 is horizontal ellipsis, U+2219 is BULLET OPERATOR, and U+2033 is DOUBLE PRIME. So I think this is specifying a transcoding from some Adobe encoding into Unicode. The font in question unfortunately uses /Identity-H as its /Encoding:

```
344 0 obj
<</Type /Font
/Subtype /Type0
/BaseFont /Georgia
/Encoding /Identity-H
/DescendantFonts [350 0 R]
/ToUnicode 351 0 R
>>
endobj
```

So apparently in the font we will find DOUBLE PRIME at CID 013C, 316 decimal.

Georgia is not one of the PDF base fonts; it's also embedded in the file:

```
356 0 obj
<</Type /FontDescriptor
/FontName /Georgia
/Flags 6
/Ascent 916.9922
/Descent 219.2383
/StemV 133.7891
/CapHeight 692.8711
/ItalicAngle 0
/FontBBox [-490.2344 -303.2227 1796.3867 1074.707]
/FontFile2 357 0 R
>>
endobj
357 0 obj
<</Length1 49484
/Filter /FlateDecode
/Length 29751
>> stream
```

`ttfdump` on the extracted font file actually agrees with my inference above about the encoding; the corresponding feature in TrueType is actually also called `cmap`:

```
'cmap' Table - Character to Glyph Index Mapping Table
-----------------------------------------------------
...

                 Seg   107 : St = 2032, En = 2033, D =  57609, RO =      0, gId# ⚬
⚬= N/A
...
Segment 107:
                Char 0x2032 -> Index 315
                Char 0x2033 -> Index 316
```

The actual text painted on the page using this font happens to be in a ridiculously inefficient form:

```
BT
/F1 11 Tf
1 0 0 -1 31.375 118 Tm
<0031> Tj
1 0 0 -1 39.813 118 Tm
<0032> Tj
1 0 0 -1 47.9985 118 Tm
<0039> Tj
1 0 0 -1 55.3301 118 Tm
<0028> Tj
1 0 0 -1 62.5166 118 Tm
<0030> Tj
1 0 0 -1 72.7163 118 Tm
<0025> Tj
1 0 0 -1 79.9082 118 Tm
<0028> Tj
1 0 0 -1 87.0947 118 Tm
<0035> Tj
```

According to the above CMap, this encodes the text "NOVEMBER". In 283 bytes. I guess it's less, deflated; the content stream for that page is 84551 bytes uncompressed, 16920 bytes deflated, so that's only about 57 deflated bytes.

Annotations and actions

The whole string thing is super confused. Line breaks are permitted inside strings and are 0x0a LF, but paragraph separators in markup annotation text are 0x0d CR (§12.5.6.2, p. 391, 399/756).

The whole annotation spec is a nightmare, and unfortunately a necessary one for including hypertext links (§12.5.6.5). Annotations can have intents, titles, subjects, reply-tos, modification dates, and author-specific states (marked, unmarked, accepted, completed). It even includes a separate richtext format that isn't PDF (§12.7.3.4, "Rich Text Strings"). They can embed arbitrary file attachments (§12.5.6.15), sounds (§12.5.6.16), and videos (§12.5.6.17). Link annotations can either go to "destinations" (§12.3.2) or take "actions" (§12.6), "such as launching an application, playing a sound, changing an annotation's appearance state."

The action spec is 16 pages long and includes halfhearted warnings against infinitely recursive and self-modifying code; in theory you should only be able to program a sequence of actions, but triggers include mouseovers, clicks, and page opening and closing. And the form spec lets you write actions in JS, including dependency-directed recalculation of form fields! And there's a SubmitForm action (§12.7.5.2) that lets you submit a data form to a URL! And ImportData (§12.7.5.4) to load data from a local file! Plus also GoTo actions to navigate around the document (§12.6.4.2, though you can also link within the document without using an action) and URI actions (§12.6.4.7, with an IsMap parameter). And you can hide or show annotations (Hide, §12.6.4.10), which can be of a variety of drawable types, as well as do a display transition (§12.6.4.14, transitions listed in §12.4.4.1 like Wipe, Dissolve, etc., with a duration), make "optional content groups" visible or hidden (§12.6.4.12), or reorient a 3-D view (§12.6.4.15).

It doesn't look like you can do arbitrary drawing from these actions, though, though maybe you could get pretty far with custom fonts. Or modify the document tree, so I'm not sure what's up with the prohibition on self-modifying code.

The JS stuff is specified in a totally separate document, "Adobe JavaScript for Acrobat API Reference".

Streams

A curious thing is that this is, I think, an indirect dictionary object:

```
31820 0 obj
<< /Length 10 >>
endobj
```

While this is not a dictionary object:

```
31820 0 obj
<< /Length 10 >>
stream
helloworld
endstream
endobj
```

That's a stream object. Stream objects must be indirect objects (§7.3.8.1) and cannot be nested within object streams like most other indirect objects (§7.5.7). So when you are parsing either of these, you don't know if you're parsing a stream object or a dictionary object until you reach the `endobj` or `stream` keyword. But you are guaranteed to hit one or the other.

The easiest way to think about this is as a sequence of stack operations: `stream` is an operation that consumes the dictionary on the stack and uses it to parse the following data before returning control of the input stream to the normal PDF parser.

Topics

• The Portable Document Format (PDF) (p. 1227) (5 notes)
• File formats (p. 1233) (5 notes)

Notes on what would be needed to drive a PS/2 keyboard from an Arduino

Kragen Javier Sitaker, 02021-06-20 (updated 02021-12-30) (12 minutes)

Trying to get a PS/2 keyboard to work with an Arduino Duemilanove.

Notes on the PS/2 keyboard protocol

Adam Chapweske's physical-layer guide clarifies the key points:

> I use a few tricks when implementing an open-collector interface with PIC microcontrollers. I use the same pin for both input and output, and I enable the PIC's internal pullup resistors rather than using external resistors. A line is pulled to ground by setting the corresponding pin to output, and writing a "zero" to that port. The line is set to the "high impedance" state by setting the pin to input. Taking into account the PIC's built-in protection diodes and sufficient current sinking, I think this is a valid configuration. Let me know if your experiences have proved otherwise. ...
>
> The PS/2 mouse and keyboard implement a bidirectional synchronous serial protocol. The bus is "idle" when both lines are high (open-collector). This is the only state where the keyboard/mouse is allowed begin transmitting data. The host has ultimate control over the bus and may inhibit communication at any time by pulling the Clock line low.

The PIC approach should work the same with AVRs, I think.

> The device always generates the clock signal. If the host wants to send data, it must first inhibit communication from the device by pulling Clock low. The host then pulls Data low and releases Clock. This is the "Request-to-Send" state and signals the device to start generating clock pulses.
>
> Data sent from the device to the host is read on the falling edge of the clock signal; data sent from the host to the device is read on the rising edge. The clock frequency must be in the range 10-16.7kHz. This means clock must be high for 30-50 microseconds and low for 30-50 microseconds. If you're designing a keyboard, mouse, or host emulator, you should modify/sample the Data line in the middle of each cell, i.e., 15-25 microseconds after the appropriate clock transition.

At the Duemilanove's 16 MHz, 15-25 microseconds is 240-400 clock cycles, so we can probably get by with polling rather than interrupts, at least initially.

Chapweske also wrote a guide to the logical protocol.

Existing Arduino software is worth trying but will be insufficient

There's an LGPL-licensed library called PS2Keyboard, last updated 2 years ago, with the last release 5 years ago, for speaking this protocol; its interface looks very simple. The page on arduino.cc is even more out of date, which I guess makes sense since it's been read-only since 02018.

```
keyboard.begin(DataPin, IRQpin); ...
if (keyboard.available()) {
```

```
char c = keyboard.read();
if (c == PS2_ENTER) {
  Serial.println(); ...
} else {
  Serial.print(c);
}
```

However, this interface has a couple of major drawbacks:

- It doesn't provide key-release data.
- Multiple special keys like F1 and F2 are all mapped to the same character code. (There's a `get_scan_code` function but it's `static`.)

The implementation also has a couple of drawbacks:

- It uses 52 bytes of RAM for buffering the keys.
- It uses an interrupt — though this is probably unavoidable with any decent interface.
- It always puts three separate 409-byte keymaps in ROM (US-English, French, German).
- It doesn't provide any of the messages *to* the keyboard, such as pinging it ("echo", 0xee), setting the scan code set (0xf0), and, most importantly, setting the keyboard LEDs (0xed).

Buffering on the keyboard?

The Wikipedia page has an intriguing comment suggesting a fully-polled mode of operation:

> When the host pulls Clock low, the device must immediately stop transmitting and release Clock and Data to both float high. … The host can use this state of the interface simply to inhibit the device from transmitting when the host is not ready to receive. (For the IBM PC keyboard port, this was the only normal use of signalling from the computer to the keyboard. The keyboard could not be commanded to retransmit a keyboard scan code after it had been sent, since there was no reverse data channel to carry commands to the keyboard, so the only way to avoid losing scan codes when the computer was too busy to receive them was to inhibit the keyboard from sending them until the computer was ready. This mode of operation is still an option on the IBM AT and PS/2 keyboard port.)

This suggests the possibility that you could pull the clock line low all the time except when polling for a keystroke, trusting the keyboard to buffer any keypresses, but I'd be surprised if that worked reliably. In any case, it would require waiting for the minimum timeout to see if the keyboard was going to send anything…

Experiment notes

I have an Arduino Duemilanove here. It can successfully run the Blink and ASCIITable examples, so it's at least mostly working.

The keyboard is an IBM KB-9910, Latin American layout with Windows key, which seems on first glance to be in pretty good shape aside from being dusty. The main board is a single-sided board built around a large Chicony DIP, 40 pins I think, with lots of through-hole parts. The keyboard switch silicone domes are all molded from a single silicone sheet, which fortunately means they can't get lost individually. The main board is connected to the thoroughly-strain-relieved cable with a four-pin Dupont connector,

with four lines on the board labeled G (violet), V (brown), D (red), and C (yellow); presumably these are ground, Vcc, data, and clock.

I reassembled the keyboard, cut off the PS/2 connector, stripped the cable a bit, and soldered pins individually to the wires. Plugging the brown and violet wires into the 5V and GND pins on the Duemilanove doesn't seem to harm the Duemilanove, which has its pin-13 LED blinking happily.

The next step, I think, is to measure sequences of data transitions and send them over the Arduino's serial port. There's an Arduino sketch under a 2-clause BSD license by Andrew Gillham that implements the SUMP logic analyzer protocol and something called Openbench Logic Sniffer at 4 MHz, with triggering support and stuff. This might allow sigrok to use the Arduino to view these signals without needing to write more code. And, to my surprise, sigrok already has a PS/2 protocol decoder in it! So if I can just get some signal data into pulseview, I can see what data the keyboard is sending.

I burned the sketch to the Arduino and connected to it in Pulseview: "Choose the driver: Openbench Logic Sniffer & SUMP compatibles (ols); Choose the interface: /dev/ttyUSB2 (FT232R USB UART - A6008ePZ); Scan for devices using driver above [usually twice]; Select the device: AGLAv0 with 6 channels." I'm trying to do captures at 200 kHz.

(Mark R. Rubin has also written a somewhat more powerful GPL firmware for the Blue Pill called "buck50" that can do 6 Msps, 5k samples, and 8 channels, and also a 1 MHz analog oscilloscope. (Too bad my Blue Pill is at home.) But it doesn't seem to work directly with sigrok; he suggests using Gnuplot to plot CSV files, saying sigrok uses too much RAM.)

Okay, I haven't figured out how to set up triggering, and the sample buffer is only 5 milliseconds long, but by doing a lot of key repeat and capturing over and over again, I think I got it to capture a keystroke. (This happens about one out of every 32-64 tries.) It does show a nice 10.0 kHz clock on the yellow wire and some kind of data on the red wire. The whole packet is 1.020 ms long, with 10 positive clock pulses (and 11 low states) in 1.050 ms. This means 11 falling clock transitions, which seems to be correct.

I get about 9 samples during each clock pulse. Sigrok's PS/2 protocol decoder is silent about what any of this means.

Oh, now I've told it to trigger on the clock going low, by clicking on the little "0" tag in the left margin. I was thinking that it was triggering too late because the data packet is at the very end of the window, but actually I think this is a bug people have reported in the past in the Arduino sketch: the values are time-reversed. So the triggering packet is at the very end of the window instead of the beginning, and it's backwards so the PS/2 protocol decoder can't decode it. Also, after the first triggering, if I do a second capture it never runs again unless I reset the Arduino.

Setting the sample speed down to 50 kHz gives me a 20.5 millisecond window instead of a 5 millisecond window; 20 kHz is too slow and misses some clock cycles. With this I was finally able to see the two-byte sequence that indicates a key-release event; there are

3.92 ms from the start of one byte to the start of the other, so the whole sequence takes 4.94 ms.

A capture at "5 MHz" has 5 full clock cycles in 0.3765 ms, 75.3 us per bit, which is really more like 13000 bits persecond. The whole 1024-sample buffer is some 511 us long, and the samples are evidently two per microsecond, so I guess it's sampling at 2 MHz.

I tweaked the sketch to send back the samples in reverse order from how they were being sent:

```
  /*
   * dump the samples back to the SUMP client.  nothing special
   * is done for any triggers, this is effectively the 0/100 buffer split.
   */
  for (i = 0 ; i < readCount; i++) {
#ifdef USE_PORTD
    Serial.write(logicdata[readCount - i - 1] >> 2);
#else
    Serial.write(logicdata[readCount - i - 1]);
#endif
  }
```

This does not seem to have made any difference, but I think I maybe made the change in the wrong place. This was the right place:

```
    if (trigger) {
      while ((trigger_values ^ CHANPIN) & trigger);
    }

    for (i = 0 ; i < readCount; i++) {
      logicdata[i] = CHANPIN;
      delay(delayTime);
    }
  }
  for (i = 0 ; i < readCount; i++) {
#ifdef USE_PORTD
    Serial.write(logicdata[readCount - i - 1] >> 2);
#else
    Serial.write(logicdata[readCount - i - 1]);
#endif
```

Now time flows forward, but sigrok's PS/2 decoder is still not able to decode the data.

However, I did manage to get something out of its SPI decoder. At first I was trying to decode the bits MSB-first, but that's wrong; they're LSB-first.

Aha, finally I got the PS/2 decoder to decode the F0 of a key-release event! It apparently doesn't decode the *last* byte in a capture (I guess because it's missing the falling clock transition that indicates the end of the bit) and all of my captures are too short so far.

Now I've verified that pressing the "A" key produces the correct 0x1C scan code; the SPI decoder set for clock polarity 1, clock phase 0, lsb-first bit order, 11 bits decodes the sequence 00011100001: start

bit 0, data bits 0011 1000 (0x1C LSB-first), odd parity bit 0, stop bit 1. The SPI decoder renders this as "438" in hex, which is just shifted left by 1 bit (the start bit):

```
>>> hex(0x438 >> 1)
'0x21c'
```

Left shift gives "624", 00100100011, which should be scan code 0x12. This seems to be correct. And sometimes I instead get 0xF0 0x12, which is the correct key release code.

I feel like for such slow signals it ought to be possible to send a continuous stream of bytes over the Arduino's serial port.

So, at least I understand the PS/2 signals, and the keyboard seems to work properly.

Topics

- Programming (p. 1141) (49 notes)
- Electronics (p. 1145) (39 notes)
- Experiment report (p. 1162) (14 notes)
- Real time (p. 1195) (7 notes)
- Protocols (p. 1206) (6 notes)
- Input devices (p. 1252) (4 notes)
- Encoding (p. 1256) (4 notes)
- Keyboards (p. 1289) (3 notes)
- Interrupts (p. 1361) (2 notes)
- Arduino (p. 1388) (2 notes)

Self hosting kernel

Kragen Javier Sitaker, 02021-06-21 (updated 02021-12-30) (1 minute)

Last time I talked to Jeremiah Orians about it, the stage0-posix project required only the following system calls of a POSIX kernel, up to M2-Planet+mescc-tools:

- execve(), fork(), waitpid(), exit()
- brk()
- open(), close(), read(), write()
- lseek()
- chmod()

execve(), fork(), waitpid(), and exit() are used in a stereotyped manner for spawning a child process running a new program, then waiting for it to exit(); and chmod() is only used to make a file executable.

He said mescc-tools-extra+Kaem also needs:

- fchmod()
- access()
- chdir(), fchdir()
- mkdir() (in particular for extracting tarballs)
- mknod()
- getcwd() (traditionally a library function implemented with stat())
- umask()
- uname() (to find out what the architecture is)

Mes proper and probably tcc also needs:

- unlink()
- ioctl() (specifically isatty(), to find out whether a shell is interactive)
- stat()
- fsync(), though of course this can be a no-op

In particular it doesn't require pipes or I/O redirection.

Topics

- Programming (p. 1141) (49 notes)
- Bootstrapping (p. 1171) (12 notes)
- Small is beautiful (p. 1190) (8 notes)
- Independence (p. 1215) (6 notes)
- Operating systems (p. 1248) (4 notes)
- Unix (p. 1268) (3 notes)

Stack syntax

Kragen Javier Sitaker, 02021-06-22 (updated 02021-07-27)
(4 minutes)

You can serialize a data structure in a compact way that can be efficiently interpreted by building it up on a stack, with pointers into older parts of the stack, minimizing parsing and permitting the fixup of pointers on input so that the final structure can be safely traversed with no overhead. This is a potentially interesting middle ground between slow serialization formats like JSON and protobufs and zero-parsing formats like FlatBuffers.

Now, the most basic thing you could do would be to have a big block of literal binary data, then a "linkage table" that just lists all the pointers in it, so that you can iterate through and offset each of these pointers by adding a base pointer to it in place, and maybe verify that it doesn't point outside the block afterwards. (Alternatively, you could do the offsetting when you follow the pointers rather than when you load them.) Import and export tables are a simple addition to such a scheme. This isn't all that compact, it's not self-describing, and it's likely to have backward-compatibility problems, but it sure is fast to load.

A somewhat less basic approach would use nested delimiters like JSON or PDF for dictionaries and heterogeneous tuples, but counted bytestrings for text or numerical arrays. By using both opening and closing delimiters for dictionaries and tuples, arbitrary nested structures can be built in place, without fragmentation. A linked list of delimiter contexts can be interspersed with the actual data on the stack to avoid any further allocation; and processing the closing delimiter of a dictionary can create, for example, an appropriately-sized hash table.

Referring to the same object more than once cannot be done with a pure stack discipline. The simplest way to handle this is probably with a symbol table: a way to define a unique name or ID number as pertaining to the next or previous object, and a way to insert a reference to another object. If this name-binding step is done at the end of deserialization, or lazily when following references, rather than during deserialization, it permits cyclic references as well as multiple references.

In cases where data is being received byte by byte over a network or serial port, it makes sense to allocate all the memory on the stack. But if the data is mapped in from a memory-mapped file, it makes more sense to build only an 'index' structure on the stack, with pointers pointing to raw binary data in the mapping.

This still requires parsing work for each text string, though, as well as each object. A faster approach is to have a single bolus of all the text and numerical data, followed by a list of offsets into it defining where each blob starts, and then the graph structure of arrays, dictionaries, and records represented with lists of object IDs. A final *coup de grace* is to include the typing information needed to decode the graph structure itself as a graph-structure item, thus making the entire

structure self-describing without inflating it with redundant type tags, length fields, and field names. The decoder must traverse the type graph in sync with the object graph in order to know how to interpret the bytes of each item.

But traversing such a file requires not only parallel type-graph traversal, but also an extra indirection through the object ID table on every reference traversal. If the object IDs in the graph structure are allocated x

Topics

- Programming (p. 1141) (49 notes)
- Syntax (p. 1221) (5 notes)
- File formats (p. 1233) (5 notes)
- Stack machines (p. 1320) (2 notes)

Bead hypertext

Kragen Javier Sitaker, 02021-06-22 (updated 02021-12-30) (1 minute)

The PDF standard (since PDF 1.1) has the amusing terminology of "beads" on a "thread" of "articles". I think "bead" might be better than "line" or "card" or "page" for my card-based hypertext thing (suggesting perhaps its intellectual descent from wampum).

And indeed the meaning in PDF is closely related:

> Some types of documents may contain sequences of content items that are logically connected but not physically sequential.
>
> EXAMPLE 1: A news story may begin on the first page of a newsletter and run over onto one or more nonconsecutive interior pages.
>
> To represent such sequences of physically discontiguous but logically related items, a PDF document may define one or more articles (PDF 1.1). The sequential flow of an article shall be defined by an article thread; the individual content items that make up the article are called beads on the thread. Conforming readers may provide navigation facilities to allow the user to follow a thread from one bead to the next.

The beads are "chained" with "N (next) and V (previous) [attributes]", while they link to their actual contents with "P" for the page object and "R" for the rectangle on the page.

My intent, of course, is to eliminate or ephemeralize the physical sequence entirely, rather than to simply superimpose a secondary sequence on it.

Topics

- Programming (p. 1141) (49 notes)
- Human-computer interaction (p. 1156) (22 notes)
- GUIs (p. 1216) (6 notes)
- The Portable Document Format (PDF) (p. 1227) (5 notes)
- Hypertext (p. 1291) (3 notes)

Does USB bitstuffing create a timing-channel vulnerability?

Kragen Javier Sitaker, 02021-06-22 (updated 02021-12-31) (1 minute)

USB adds a 0 bit after 6 (or 5?) consecutive 1s. Does this create a timing channel attack? Is there a two-out-of-5-code-like approach that avoids this?

A timing-channel attack would occur when some kind of private data is being transmitted in cleartext over the channel, such as a password or an encryption key, and an attacker can observe the *length* or *timing* of the transmission but not its content. It would be especially helpful to the attacker if they could somehow send data that was concatenated with, or especially interspersed with, the bitstuffed data, repeatedly, because that would allow them to determine that a particular bit was a 1 by inserting several 1s before it. Although this sounds far-fetched, several vulnerabilities of this sort have been found in SSL and TLS. Still, USB seems likely to be less vulnerable.

Many other line protocols instead use constant-overhead encodings like 8b/10b encoding to ensure adequate state changes and line balance at a modest efficiency cost. This approach is guaranteed to not create a timing-channel vulnerability in this way. FC, DVI, HDMI, DisplayPort, FireWire, SATA, and USB3 all use 8b/10b.

Topics

- Protocols (p. 1206) (6 notes)
- Security (p. 1224) (5 notes)
- Encoding (p. 1256) (4 notes)

Verstickulite

Kragen Javier Sitaker, 02021-06-23 (updated 02021-07-27)
(3 minutes)

Vermiculite is readily available at plant nurseries and garden stores, and withstands temperatures up to 1150°. But it comes as an aggregate of fragile, loose particles; to make a solid object out of it you need to stick them together somehow. The standard approach is to use sodium silicate, but I don't have any.

Some adhesives are only activated by heat, and those would need a low-temperature adhesive to give the "green body" enough "green strength" to survive until firing. For many uses of vermiculite, much of the body would never reach a high enough temperature to activate it.

The most obvious such "adhesive" would of course be a hydrated clay, but low-temperature clays might slump too much at temperature.

Glassy carbon might be a useful, though weak, adhesive for this purpose; it won't flux the vermiculite, won't melt up to much higher temperatures, and can be easily made from sugar. You could tumble damp vermiculite particles in powdered sugar, or spray sugar water onto tumbling vermiculite, so that each particle is coated without penetrating much into its interior. Heat caramelizes the sugar into glassy carbon, hopefully without expanding the aggregate much. The sugar itself can serve as the initial adhesive if enough humidity is available and it is prevented from crystallizing. The greatest drawback of glassy carbon is that in an oxidizing atmosphere it erodes rapidly.

Superficial borax might also work: a thin coating of dissolved borax (31.7 g/ℓ) can dry and crystallize, providing green strength; upon heating, it will produce anhydrous borax at 75° and then boric-acid glass. This should eventually diffuse into the phyllosilicate vermiculite grains and eliminate the low-melting phase.

Soluble sodium donors also seem promising to apply in the same way: sodium hydroxide, carbonate, or bicarbonate on the surfaces of the grains should enable them to sinter together by forming a small amount of sodium silicate in situ. Trisodium phosphate might also work, perhaps forming silicon aluminum phosphates, though its aqueous solution might attack the vermiculite even at room temperature. Phosphates in general may be a useful way to increase the temperature the final mixture can handle, counteracting the fluxing effects of additives like boria and soda.

The most accessible alkali donor is probably wood ash, with its mixture of oxides and carbonates of sodium and potassium, which potters sometimes use for ash glazing.

(Some sodium hydroxide is currently crackling across the room from me as it releases its water of hydration in a pile of vermiculite on top of an electric burner; the temperature of the burner is plenty hot enough to completely melt it, but the vermiculite is slowing the process. Probably a more finely divided form, or a spray coating on the vermiculite granules, would have been better; both the

vermiculite grains and the NaOH grains are about 3 mm.)

(The final result of that experiment: the sodium hydroxide apparently simply disappeared, leaving the vermiculite loose and not visibly changed.)

See also More cements (p. 466) for more notes on mineral cements that might be applicable.

Topics

- Materials (p. 1138) (59 notes)
- Waterglass (p. 1189) (8 notes)
- Refractory (p. 1225) (5 notes)
- Cements (p. 1235) (5 notes)
- Vermiculite (p. 1238) (4 notes)
- Sugar (p. 1271) (3 notes)

Simple linear-time linear-space nested delimiter parsing

Kragen Javier Sitaker, 02021-06-24 (updated 02021-12-30) (1 minute)

Suppose you're parsing a token stream containing nested delimiters, like (a b (c (d) e ((f g))) h), storing the tokens as you go, and you'd like to store information about the nesting structure in the stored tokens themselves, which are of some fixed size.

You can clearly get by with a previous-sibling pointer and a previous-parent pointer, populated online as you parse the string, if you're going to traverse the string backwards; on close delimiters, you could repurpose the previous-parent pointer as a last-child pointer, since if you want the parent you could just follow the previous-sibling pointer to the matching open delimiter and use *its* parent pointer. When adding a close delimiter and thus ending a level, to find the matching open delimiter, simply leap to the last delimiter or delimiter pair within the level now ending and ask them for their parent; when adding a new open delimiter, look at the previous delimiter, and if it was a close delimiter, it is your sibling and tells you your parent; if it was an open delimiter, it is your parent and you have no siblings yet. This permits building the structure in linear time with only a single pointer to leap over the trailing non-delimiter tokens with.

Topics

- Programming (p. 1141) (49 notes)
- Algorithms (p. 1163) (14 notes)
- Facepalm (p. 1199) (7 notes)
- Parsing (p. 1228) (5 notes)

Economic history

Kragen Javier Sitaker, 02021-06-25 (updated 02021-07-27)
(17 minutes)

(Originally posted at https://news.ycombinator.com/item?id=27639177.)

Money doesn't objectively exist; it's purely a figment of people's imagination. This ten-trillion-Zimbabwean-dollar bill isn't money anymore, even though it used to be, and it isn't physically changed in any relevant way. The only thing that changed was how people thought of it.

It might seem strange to say that a purely imaginary thing like money could be "the driver/catalyst for everything" in industrialization, but many strange things are true. I'm writing you this note in letters of lightning far too small to see, which are persistently trembling within a few tiny slivers of quartz that can thus contain the lightning because the vital air has been removed from it—stranger still, but true, because that is how DRAM chips work.

Still, let's see if we can analyze how industrialization happens in terms of objectively existing reality rather than shared hallucinations like money—surely the picture will be partial, but it may still be useful. Industrialization itself seems to be objectively observable: it doubles people's life expectancy at birth.

Let's start with an outside view of economic production as a whole. How would we explain it to a Martian?

People work, planting and harvesting and preparing food. If they do not eat food, they die in a few months from lack of various molecules crucial to their biochemistry and energy (collectively "nutrition"). They expend energy and water and damage their bodies by working, damages which they repair automatically if they have time to rest, but which will otherwise eventually kill them. So their capacity for work is limited. Not everyone has to work, but if the people collectively produce too little food from their work, then they will start dying from lack of nutrition, and pretty soon they will all die. One old book of poetry sums this up by saying, "In the sweat of thy face shalt thou eat bread."

On the other hand, if they can make a lot of food despite their limited work capacity, more than they need to survive—a "surplus"—many of them will live for decades, they will reproduce, and their population will increase, typically by about 4% per year. Groups that manage to do this come to vastly outnumber groups that do not.

(There are some other material necessities for human survival and reproduction besides food, but none of them are different in relevant ways, so for the time being I'll stick to food.)

The other conventional factors besides labor that affect productivity are land and something called "capital". Some land yields a lot of food and is easy to plant and harvest; other land yields very little. By "capital" is conventionally meant not money but durable goods that increase labor and land productivity, such as hoes

and waterwheel-driven mills for making flour. If people don't need to spend their entire labor capacity on making enough food to survive, they can instead devote some of it to making these durable goods: cords, needles, ropes, bows and arrows, shoes, knives, hoes, axes, pots, and so on. Cords and ropes enable you to climb palm trees, hang meat over a fire, or bridle a horse; needles enable you to make clothing and all manner of sturdy, flexible fiber goods; bows and arrows enable you to kill dinner at a distance; shoes enable you to walk or run longer distances; knives and axes make cooking and woodworking much easier; hoes allow you to plant much more land; pots enable not only cooking but food storage and the storage of other goods; and so on.

So these are "capital goods": once you have a surplus, then by spending some of your work creating durable goods instead of satisfying basic necessities for survival, you thenceforth multiply their possessors' future production, or at least until the durable goods stop enduring. But a hoe or a blast furnace is produced in the same way as a vegetable garden: people work to turn raw materials into the desired end product, directed by their skills and knowledge. Instead of hoeing the field today, perhaps they're hammering the hoe blade into shape or molding bricks from fireclay. Most capital goods are more or less specialized to a particular sort of production—you cannot harvest more corn by dumping blast furnaces or gristmills into the cornfield—though some are more versatile than others.

I haven't mentioned skills and energy before, but they're crucially important. A person is skilled when they can direct their work to produce their intended results with ease and precision, and labor specialization develops skill much more highly.

Energy is a fundamental physical quantity which can be converted between many different forms, including heat and kinetic energy; many kinds of work require a lot of kinetic energy and a lot of heat, and this has historically often been the limiting resource for work productivity, particularly when all the relevant kinetic energy came via people's muscles from their food, and all the relevant heat came from firewood.

The available kinetic energy for production has greatly expanded five times in the past: ox-yokes (6000 years ago), horse-collars (1500 years ago, in the Sui), the steam-engine (250 years ago), and electric dynamos and motors (140 years ago). Photovoltaic cells seem at long last to be adding a sixth item to this list now that they have finally become cheaper than steam-engines.

The available heat energy for production has greatly expanded three times: fire (400,000 years ago), oil drilling (2300 years ago), and deep-shaft coal mining (250 years ago). Again, photovoltaics are probably a huge factor here today.

So, in these terms, we can analyze industrialization as a set of several major synergistic shifts.

First, enormously increased specialization, which would have been counterproductive to effective productivity in the absence of the worldwide markets created by the British Empire and by cheaper transport via canals. The ten workers in Adam Smith's pin factory could produce 48000 pins a day, far more pins than a village could

use, or maybe even a city—perhaps each household might ruin five or six pins in a day as they repaired their clothing. Without easy shipping, the tonnes of pins thus produced would be as worthless as an asteroid belt converted into paperclips.

Second, enormously increased use of capital goods enabled by that specialization—only a few decades after Smith observed his pin factories of ten or eighteen men, the whole operation became automated by cam-driven pin-making machinery, further decreasing the labor per pin, and of course the rise of the spinning-machines and jacquard and other automated looms is even more notorious.

Third, as improved capital goods like the cotton gin and the combine harvester became abundant, the planting and harvesting of larger and larger amounts of land required less and less labor, so smaller and smaller amounts of farming work per person sufficed to prevent mass starvation. This surplus manifested as a mass population migration from the countryside to the cities, where increasingly specialized capital-intensive production of all kinds was carried on by large crews of people harnessing steam power, rather than by individual farmers plowing behind teams of horses.

Fourth, *invention* became central to economic production in the newly industrializing countries, as it had been a thousand years before in the Song; we can think of an invention as a skill that has been digitized, expressed in words or pictures so that it can be copied, rather than having to be learned by practice. New notions of precision, innovation, specification, tolerances, and standardization became central to work in a way entirely foreign to previous generations. New notions of measurement and metaphors of clockwork universes allowed the implicit to become explicit, legible, and reproducible.

Fifth, the advent of the steam-engine, which permitted pumping water out of deep-shaft coal mines, which provided a vastly increased fuel supply.

— ⁂ —

In none of the above do we find any money, or for that matter any elites or banks or other corporations. All of it could, at least hypothetically, have happened through Stakhanovite altruism or telepathic mind control, if human nature contained such possibilities. The investment of surplus production capacity in capital goods which thenceforth further increase the production surplus, faster than the human population can increase, combined with labor specialization, urbanization, and invention, creates industrialization, and that is what creates the profligate material abundance and extended lifespans we are struggling to cope with today.

But of course it is not enough that such a pattern of changes improves the overall welfare; if the *status quo ante* is a Nash equilibrium, anybody who unilaterally attempts to automate pinmaking, invent cotton gins, or move to a city might merely impoverish themself and their family. Indeed, if you merely specialize in pinmaking, surely you will have made more pins than your family needs within only a few weeks—and how will they eat this winter if you've spent the whole planting season making pins instead of plowing? Even if your aunts and uncles will plow for you,

they probably don't need that many pins either.

And *this* is where money comes in: trade gamifies work by allowing you to cooperate productively with people you don't trust, breaking the Nash equilibrium that keeps everybody poor, and money enormously simplifies trade. You can ship a box of pins from your French pin factory to Spain or England or some godforsaken place like Connecticut, and receive gold in return, which you can use to buy both potatoes and better pin-making machinery. As you said, it's a lubricant.

Modern capitalism largely arose in the Industrial Revolution as a form of gambling. Many entrepreneurs can bootstrap: a machinist can buy a lathe, a drill, a work-bench, a file, dividers, a square, etc., with his wages as a machinist's apprentice, and as long as he has a "production surplus" (now, as measured in the collective hallucination of money, rather than in the increased harvests of the farmers using his machinery) he can keep improving and buying machinery to increase his productive capacity and thus his surplus, and perhaps begin to hire workers—not apprentices for a limited term, now, as in pre-capitalist craft production, but employees for life. But a competing machinist who sells her business to capitalists (whether a few wealthy investors or many subscribers) suddenly has enormously more money with which to buy such tools, and perhaps she can win customers away from the bootstrapped entrepreneur, producing superior goods at a lower price because of her superior tools.

Effectively the capitalist is offering the worker a deal: the capitalist provides the tools, the worker provides the labor, and each receives part of the value produced. In capitalism *qua* capitalism, the worker is an employee and receives a *fixed* wage, while the capitalist receives a variable profit, but of course there are lots of other variations; the machinist might take a loan from a bank or rent the tools, precisely reversing the roles, or someone who's nominally merely an employee might receive most of their compensation in the form of incentive stock options.

From the point of view of economic development, these questions of how the pie is divided are minor, but they matter enormously to the participants. This division depends mostly on cultural expectations and on the negotiating positions and skills of the parties; in an industrialized economy, the capitalist has a very strong negotiating position, since capital goods like drill presses and rail locomotives are very commonly the critical limiting factor in productivity. For many expensive products that can be cut on a lathe, there are fewer lathes that can turn them than machinists skilled enough to turn them. The machinist without a lathe must resort to backyard aluminum casting and hand scraping before he can start to turn parts.

Ultimately, though, the money is just a scorekeeping device, a lubricant. What makes the machine shop more or less productive is not the money, but the tools it bought, which were necessarily produced by the production surplus of the existing economy. If suddenly machine shops as a whole attract a hundred times as much investment, it won't make their productivity skyrocket by bringing a hundred times as much tools into existence—it will just drive up the

price of the tools. Pretty soon it will drive up the prices of the machine shops' products, too, since if their management doesn't deliver a respectable return on that investment, it will be replaced, and if that doesn't fix it, the company will be shut down. These higher prices will tend to *decrease* the market for the products.

When we're talking about international capital flows, we also have the well-known resource curse or Dutch disease: investing a lot of money from England in US companies will tend to devalue the pound relative to the dollar, which makes US products less economically competitive and/or UK products more economically competitive. This is precisely the opposite of driving US industrialization.

— ⁂ —

So if some Englishman sent a bunch of money across the Atlantic to invest in nascent industrialization, that wouldn't speed up the industrialization—unless a bunch of that money went *back* across the Atlantic to buy the capital goods needed for industrialization. Maybe it did! From 01780 to 01824 artisans were forbidden to emigrate from Britain, and exportation of metalworking or glass manufacturing equipment was likewise forbidden; but these prohibitions were widely flouted, and at any rate they excluded steam-engines, and they ended in 01843. A lot of the machinery and artisans during this time went to the US. The "American System of Manufacturing" was the world's most advanced manufacturing technology by the mid-01850s.

More likely, though, if someone in England was investing in US industrial firms in the 01800-01850 period, they weren't meaningfully promoting a flow of needed capital goods from England to America; they were just giving the American firms they invested in a leg up on competing American firms, and perhaps other sectors of the US economy, like the slave plantations.

But where does Junius Morgan come into all this? Well, it turns out that Junius S. Morgan was born in Massachusetts in 01813; his father was a Revolutionary War veteran and one of the founders of Aetna in 01819. Junius Morgan was a partner in various US banks from 01833 to 01853, inheriting his father's fortune in 01847 and then becoming a partner in London's leading *American* bank in 01854. So *that* was when he started funneling British investors' money into American industrial companies by selling things such as railway bonds. J. P. had nothing to do with it! He started working with his father in London in 01857 and acting as his father's overseas agent in the US in 01860.

Moreover, by this point it was no longer a question of "driving industrialization" in America. The US had already been the world's leading industrial power for decades. The B&O railroad opened in 01830, using a US-built locomotive; by 01861 it had 236 locomotives, 3451 rail cars, and 826 km of railroad, and it had many domestic competitors. In 01830 the Liverpool and Manchester Railway, the first intercity railway and the first to be powered entirely by steam, also opened. The US had power looms from 01815, milling machines by 01816, small arms with interchangeable parts by the 01830s, 2000 steam engines by 01838, cloth exports from the 01830s, electric telegraphs from 01844, and invented the Corliss steam engine in

01848.

So, Morgan was far too late to drive industrialization in the US. He certainly did make a lot of money from it, though. And he arguably got a lot of other people to invest in its expansion—US industry expanded a lot in the second half of the 19th century.

Topics

- History (p. 1153) (24 notes)
- Incentives (p. 1230) (5 notes)
- Economics (p. 1258) (4 notes)
- The United States of America (USA) (p. 1314) (2 notes)

More cements

Kragen Javier Sitaker, 02021-06-26 (updated 02021-08-15)
(5 minutes)

I've been thinking more about chemical reactions that can produce inorganic solids that are hard and/or strong and/or refractory from conveniently shapable precursors. Previously I was mostly thinking of things that react in aqueous solution, but now I've also been thinking about granular substances you could mold into shape (in some kind of carrier, such as a thermoset resin) and then heat up to activate the reaction.

In Dercuano in file `berlinite-gel` I mentioned Grover et al.'s low-temperature synthesis of alumina ceramics bonded by berlinite (aluminum orthophosphate) by partly dissolving alumina in aqueous 50 wt% phosphoric acid at 130° (with a 5:1 weight ratio of alumina to phosphoric acid) for 1–4 days, then heating the resulting a thick puttylike gel of hydrated aluminum phosphate to only 150° for 1–3 days to get a 20%-porosity solid of nearly 50 MPa compressive strength. Similar substances ("MALP" or "MAP") are widely sold as castable refractories. (See Cola flavor (p. 707) for more on this substance.)

With respect to adding phosphate cross-linkers, the ammonium phosphates should not be overlooked; some are deliquescent under ordinary conditions, easily dehydrated to crystallinity with gentle heating, and yield phosphoric acid when the ammonia is driven off with stronger heating.

An interesting benefit of phosphoric acid as a binder (or acid phosphates like monoaluminum phosphate or monomagnesium phosphate) is that it can bind silicates like soda-lime glass, but doesn't destroy them the way lye does. Soda-lime glass fiber is only about US$3/kg, according to Potential local sources and prices of refractory materials (p. 566), which is enormously cheaper than any alternative fiber reinforcement of similar strength. Maybe by combining phosphoric acid, some kind of high-strength foam (see Glass foam (p. 595)) and oriented glass fiber, you could produce a structure analogous to wood. Space Shuttle TPS protective LI-900 HRSI tiles were a felt of fused-quartz fibers, but they were adhered together with, I think, borosilicate.

More discussion of such composite materials is in Fiberglass CMCs? (p. 588).

A different way to get phosphates of aluminum might be to react sodium aluminate, easily prepared by digesting aluminum in lye, with a soluble phosphate, such as one of the phosphates of sodium or ammonium. I seem to have had some success with this with fire; see Material observations (p. 633), 02021-08-06. WP says sodium aluminate is thus used to precipitate soluble phosphates in water treatment plants.

Sources of aluminum may be other useful bonding agents; water-soluble sodium aluminate precipitates aluminum hydroxide in water upon cooling, part of the Bayer process; even the insoluble

hydroxide itself may serve as a useful bonding agent due to its tendency to form a metastable hydrogel which eventually crystallizes. Calcining the hydroxide yields sapphire.

Aluminates of calcium are also potentially interesting, especially for 3-D printing; the highly reactive tricalcium aluminate is notorious for causing "flash setting" of concrete immediately upon hydration, though a small amount of sulfate can prevent this. WP says:

> Water reacts instantly with tricalcium aluminate. Hydration likely begins already during grinding of cement clinker due to residual humidity and dehydration of gypsum additives. Initial contact with water causes protonation of single bonded oxygen atoms on aluminate rings and leads to the formation of calcium hydroxide. 4 The next steps in the sequence of the hydration reaction involve the generated hydroxide ions as strong nucleophiles, which fully hydrolyze the ring structure in combination with water.

But of course if you want to 3-D print a stone object by spraying water onto a powder bed, this kind of instant setting may be highly desirable, and tricalcium aluminate's other deleterious effects on concrete may not be relevant.

My previous thought of precipitating calcium phosphate through a double metathesis reaction still might work as a cement, even with melting rather than in a solvent system. It's similarly fairly instant. I had settled on calcium chloride and agricultural diammonium phosphate, and the thought at the time was an aqueous reaction; but DAP decomposes to yield liquid phosphoric acid (and other crap) at only 155°, which I think can attack the solid calcium chloride (which melts at 772°). Maybe it can also attack solid metallic aluminum to get phosphates of aluminum. Monoammonium phosphate is cleaner, decomposing to yield only ammonia and phosphoric acid at I think a slightly higher temperature.

Reportedly you can react alumina with boria at 800° or 600°–800° or 1200° and get the exotic aluminum borate, which reportedly sublimes at 1050° or above 1300°; reacting aqueous borax with aluminum sulfate below 45° reportedly gives aluminum borate directly. For $Al_{18}B_4O_{33}$, Vickers hardness is reported as 6 GPa, Young's modulus 400 GPa, tensile strength 8 GPa (65% higher than basalt fiber, 15% higher than carbon fiber, three times maraging steel), Mohs hardness 7, crystal habit acicular, and the whiskers have been used to strengthen aluminum. There are a few different aluminum borates or boron aluminates, though.

Topics

- Materials (p. 1138) (59 notes)
- Phosphates (p. 1184) (9 notes)
- Foam (p. 1185) (9 notes)
- Cements (p. 1235) (5 notes)
- Sapphire (p. 1331) (2 notes)

Base 3 gage blocks

Kragen Javier Sitaker, 02021-06-27 (updated 02021-12-30)
(5 minutes)

Suppose you want to make a minimal number of gage blocks (Joe blocks) out of Zerodur or something similar, with precision of 0.1 microns and a range from 2 mm up to 200 mm in increments of 0.5 microns. Conventionally this is done with combinations of 1 to 7 gage blocks from a set of some 80 or 90 pieces. (Also, conventionally, the blocks are made of hardened steel rather than a ceramic, despite steel's inferior thermal stability, chemical stability in air, and tendency to raise burrs when scratched. Zirconia gage blocks are coming into use, but they are no more thermally stable than steel; a zirconia coating on a Zerodur base would probably be better still. Tungsten carbide is an intermediate ceramic that is seeing some use with a lower TCE.)

Well, in theory you could try doing it in binary, with sizes in microns 0.5, 1, 2, 4, 8, 16, and so on, 18 sizes in all to get to 200 mm.

But a gage block that's less than 10 microns thick will be destroyed when you touch it, probably 100 microns if it's a fragile material like Zerodur. If instead you start at 200 microns and go up with these binary increments, you could start with 200.5 microns, 201, 202, 204, 208, 216, 232, 264, 328, and 456, plus ten 200-micron blocks, then you can wring together ten items from this set to get from 2 mm to 2.5115 mm in 0.5-micron increments; at this point you can add blocks of 0.512 mm, 1.024, 2.048, 4.096, 8.192, 16.384, 32.768, 65.536, and 131.072 to get any number up to 200 mm. Up to 264.1435 mm, in fact. But that's 29 blocks instead of 18.

Suppose that instead we use balanced ternary and add an offset of 250 microns. Our first few blocks would be of sizes 249.5 microns, 250.5, 248.5, 251.5, 245.5, 254.5, 236.5, 263.5, 209.5, 290.5, plus 5 blocks of 200 microns. Combinations of 5 of these 15 blocks give us any size from 1.1895 mm up to 1.3105 mm in increments of 0.5 mm, 243 measurements covering a total range of 121 microns. If we offset our next power of 3 around 321.5 microns instead of 250, we have three more blocks of 200, 321.5, and 443 microns, and now we can cover the range from 1389.5 microns to 1753.5 at 0.5-micron intervals by wringing together 6 of these 18 blocks, a 364-micron range. A single additional block of 364 microns doubles our range, and now we have a 728-micron range in combinations of 6 or 7 blocks out of 19. By increasing the offsets mentioned above we can relocate that range to cover 2.000 to 2.728 mm; this is clearly a small improvement over the previous system, which needed 20 blocks to cover 2.000 to 2.5115 at the same resolution. Adding 9 more binary blocks as before gets us to our objective in 28 blocks.

Still, I can't help but feel that this is not optimal. There are 524288 combinations of our 19 initial blocks, but these are only giving us 1457 different lengths within their intended range; many combinations are outside the bounds (including all the combinations with less than 6 or more than 7 blocks), and 5 of the initial blocks are 250 microns in size and thus interchangeable, so there are many equivalent ways to

achieve most of our 1457 lengths by using one or another 250-micron block. Ideally if we wanted to be able to construct 1001 evenly spaced lengths from 2.0000 mm to 2.5000 mm, we could do it with only 10 blocks (from which there are 1024 combinations), or maybe only a little bit more, rather than needing as many as 19.

So, it seems that this solution can be easily improved (it's only about 0.3% efficient) but it's not clear to me what the optimal solution is or how to calculate it.

Making the gage blocks circular rather than rectangular might make it more difficult to wring them together, but it would also reduce their vulnerability to chipping.

If these blocks are themselves microscopic in size and intended to be handled by microscopic manipulators, the problem of very thin blocks may become less severe, as breakage is less of a problem at small scales; the forces needed to lift and manipulate objects are smaller relative to their strengths than at macroscopic scales. But surface effects like adhesion and wear may become more troublesome.

Topics

- Contrivances (p. 1143) (45 notes)
- Manufacturing (p. 1151) (29 notes)
- Machining (p. 1165) (13 notes)
- Math (p. 1173) (11 notes)
- Composability (p. 1188) (9 notes)
- Metrology (p. 1212) (6 notes)
- Length (p. 1356) (2 notes)

Multiple counter-rotating milling cutters to eliminate side loading

Kragen Javier Sitaker, 02021-06-27 (updated 02021-12-30)
(7 minutes)

Milling machines need a lot of rigidity to avoid not only imprecision but even oscillation, or even grabbing in the case of climb milling, under the heavy side forces created by their cutters. Their workholding setups also must withstand these side forces.

The cutters are most commonly end mills doing side cutting. It occurs to me that, when doing a straight cut while side cutting, if you have two counter-rotating end mills close to one another, so that one is climb-milling while the other is conventional-milling, then only the spacing between the cutters needs to be rigid to resist the cutting forces, because the forward force by one cutter can be balanced against the reverse force by the other cutter. By adjusting the two cutters' engagement depths and spindle speeds, you can use a controlled imbalance in these side loads to feed the machine through the material, eliminating the need for a high-power feed motor driving the X and/or Y axis; the feed motor only needs to oppose the side loads *perpendicular* to the toolpath, which will be only partly canceled by this setup but do not require any power, just holding.

The two spindles might be mounted a fixed distance apart on a small turntable rotating around a C axis, for example.

Of course you need active feedback to monitor the resulting positions and feedrates and adjust the angles and spindle speeds between the two cutters to compensate.

Doing this for more complex surfaces might require rotating the carrier for the two cutters around two axes, not just one. If the cutters are not ball-nose endmills, it might then be necessary to rotate the spindles back to verticality.

In face milling, it should be possible to do something similar most of the time while cutting almost twice the width of a single face-milling cutter in a single pass, by overlapping the paths of the cutters slightly, while overlapping the path of the leading cutter with a corresponding amount that was cut in the previous pass. That same overlap will create an imbalance between forward and backward forces which can enable the cutting forces to feed the cutter into the material.

To do the same kind of thing for single-point cutting on a lathe, you could use two cutters on opposite sides of the workpiece, perhaps with a negative rake angle on them so that if one strays a bit closer to the center of the material than the other, the cutting force will push it back out, so that the cut is the same depth on both sides. Using three cutting points instead of two eliminates an undesired degree of freedom in movement.

This is pretty much the same way that dies, taps, and countersinks center themselves on existing round features, and drills follow existing holes, so it could lead to cutting non-concentric features. If

the cutters' rake angle can be varied dynamically during each revolution, you could use position feedback to cut deeper on the high side and thus recenter the cutting; a system of levers referenced to the desired cutting axis can achieve this. The same approach can be used in place of a boring bar: a sort of hole-expanding drill or boring head that self-centers on the desired cutting axis by changing the rake angles of its cutting points.

These drilling-like approaches surely eliminate most of the usual side forces and workholding forces: if you're cutting a vertical cylinder, inside or outside, then all the horizontal cutting forces cancel between the three cutting teeth. It's like a drill press: you still need potentially a vertical force to feed into the stock (and to hold the stock in place as you do that), which maybe you can avoid by tilting the teeth forward or back to pull it into the stock at an appropriate speed.

But the bigger remaining issue is that the *moment* from rotating the teeth relative to the workpiece is still present; this is what potentially causes a poorly held workpiece in a drill press to spin around on the drillbit and break your hand. If you're cutting on a large radius, this moment is potentially very large, and it needs to be resisted (probably quite rigidly) across the whole workpiece-workholding-frame-spindle chain. If instead of using three rotating *teeth* you use four rotating *endmills* with steep enough helices to ensure continuous contact, then this problem can be eliminated; two of the endmills can rotate in one direction while the others rotate opposite them, with the relative depths of engagement determining the overall revolution of the assembly. It might be possible to reduce this to two endmills and a roller.

Using ordinary endmills would give up the ability to control the rake angle, but a more elaborate endmill design with movable inserts could retain that degree of control as well.

For face milling, the equivalent of concentricity is tramming. You could imagine a sort of three-pointed fly cutter which dynamically adjusts the rake angles of its three teeth as it moves over the surface so as to cut deeper on one side than the other, in order to bring the surface into parallelism with the desired plane. If we suppose the surface is horizontal, the cutter's yaw axis is driven by a spindle, while its pitch and roll axes are controlled by its engagement with the surface. Some kind of force is needed in Z to get the teeth to dig into the stock (unless a steep rake can provide that force on its own). The varying degrees of engagement produce side forces in X and Y, which will not in general correspond with the desired direction of motion across the workpiece, so X and Y feed motors are also needed.

If instead of one three-pointed fly cutter we have three, the X and Y side forces from the nine points can be controlled to provide the desired X and Y feed.

Topics

• Contrivances (p. 1143) (45 notes)
• Manufacturing (p. 1151) (29 notes)

- Machining (p. 1165) (13 notes)

Layered ECM

Kragen Javier Sitaker, 02021-06-27 (updated 02021-12-30)
(2 minutes)

Electrochemical machining cuts almost any metal, has reasonably high material removal rates, has no side loading or heat affected zone, and permits cutting fairly free-form parts in 3-D, much like CNC milling. But in its ordinary form it can't cut parts with complex internal spaces, and in a pure 3-axis form it can't even cut overhangs.

Watching demo videos of ZURAD Engineering's ECM jet-cutting machines, though, it occurred to me that we can go further still. If we use ECM cut layers of the 3-D shape we want, then stack them up, we should be able to get not only much freer-form shapes, but also effectively a higher material removal rate, since we only need to remove a perimeter of stock around the shape of each layer, while if we were cutting the same part out of a solid block of stock, we'd have to remove all the material all the way to the surface of the stock.

There are many possible ways to manage the layer stackup, but locator pins for alignment and thin plastic backing for each layer seem like one straightforward option; the pins can instead be replaced with thin strips that are cut out with ECM in the same way.. The plastic backing can be burned away after the layers are stacked up.

After roughing out the shape in layers this way and stacking them up, you can do the final precision shaping work with ECM on the stacked-up part.

ZURAD's soon-to-be-open-source ZURAD Two EC Jet Cutter evidently cuts 1mm-thick sheet steel with a gentle stream of water with 100 grams of NaCl per liter of water (or was it 200?), with the metal nozzle about 2.5 mm away from the steel. For some reason he positions the sheet steel horizontally instead of vertically. The video looks like it cuts at about 2mm/sec, but at one point the guy holds up a part ("a simple mechanical pawl") with a perimeter of about 100mm, and says it took an hour and a half to cut completely through, so maybe it took 50 passes. I have no idea what current or voltage he's seeing; on his die-sink ECM machine he uses 12 volts.

Topics

- Manufacturing (p. 1151) (29 notes)
- Electrolysis (p. 1158) (18 notes)
- Machining (p. 1165) (13 notes)
- ECM (p. 1186) (9 notes)

A kernel you can type commands to

Kragen Javier Sitaker, 02021-06-27 (updated 02021-12-30) (1 minute)

What if your system call interface was usable as a shell?

That is, you might plug a user process into an outlet that lets it ask the kernel to do things and get back responses, but the language that it's speaking to the kernel is a language that a person can also reasonably speak. Or, maybe not a person, but at least a text terminal, if it's plugged into that same outlet. Maybe you can open a file and read some data from it by typing a couple of lines of text into the kernel, and those are the same lines of text a program would send to do the same operation.

In a sense, this isn't such a profound idea: on a modern computer, instead of the shell interpreting user interface events (received through system calls) and translating them into system calls, the terminal emulator would interpret user interface events received through system calls and translate them into system calls. The actual interface between the terminal emulator and the kernel wouldn't have to look like an ASR-33 byte stream punctuated by CRLFs; it could take any convenient form. The key difference from current shells is that the form of the interface is something that could just as well be used to talk to programs other than the kernel, and often is.

Topics

- Human-computer interaction (p. 1156) (22 notes)
- Systems architecture (p. 1205) (6 notes)
- Post-teletype terminal design (p. 1207) (6 notes)
- Operating systems (p. 1248) (4 notes)

Can you use stabilized cubic zirconia as an ECM cathode in molten salt?

Kragen Javier Sitaker, 02021-06-27 (updated 02021-12-30)
(3 minutes)

You ought to be able to use yttria-stabilized zirconia, magnesia, carborundum, or graphite as a cathode for electrochemical machining of metals in a molten-salt electrolyte. I think zirconia needs to be heated up to 500 degrees, maybe 800 degrees, before it becomes conductive, but carborundum and graphite do not.

Magnesia was what Nernst used in his original lamp, with a platinum preheating filament, but it is fragile and presumably requires even higher temperatures to become conductive than zirconia; the temperature for magnesia was originally claimed to be over 3000°, though magnesia melts at only 2852°. Hartman solved the mystery in 01906, finding temperatures from 1780°–2360°, but these were the temperatures at which the globars were operated to provide light, not the lowest temperatures required to make them conductive.

The commercially sold Nernst lamps used platinum filaments heated to redness to preheat the "glowers", which were presumably already zirconia rather than magnesia, though the 01903 manual doesn't say, only that they are "porcelain-like". Hartman said the glowers he tested in 01906 were magnesia. Mills's history of the lamp said magnesia required "above a red heat", and explained that it required a ballast resistor to stabilize it because of its negative temperature coefficient of resistance. Nernst used a red-hot iron wire in a hydrogen envelope for this resistor.

Mills also mentions that early carborundum was insufficiently conductive for use as globars.

Another promising oxide for this sort of thing is bismuth trioxide, which is significantly conductive down to room temperature. The monoclinic room-temperature form is four orders of magnitude less conductive than the delta form that's favored above 727°; analogous to YSZ, doping with other metal oxides can stabilize these more conductive forms to lower temperatures. However, its melting point is only 817°, so while it might be useful for molten-salt electrochemistry, it wouldn't work well as a globar. (It's also nontoxic, insoluble in water (though attacked by CO_2 unless stabilized with silicate), and has an astounding density of 8.9 g/cc; sillenite, a bismuth silicate, is even denser at 9.2 g/cc, and though very soft, it might be an interesting aggregate to add to pottery to increase its density.)

ECM at the higher temperatures enabled by molten salts ought to permit faster material removal than in aqueous solutions, and might be particularly appealing for metals that are difficult to machine electrochemically in aqueous solutions; however, I think only gold and some of the platinum-group metals really suffer from this.

Topics

- Materials (p. 1138) (59 notes)
- Manufacturing (p. 1151) (29 notes)
- History (p. 1153) (24 notes)
- Electrolysis (p. 1158) (18 notes)
- Machining (p. 1165) (13 notes)
- ECM (p. 1186) (9 notes)
- Carborundum (p. 1381) (2 notes)
- Zirconia
- Nernst lamps
- Molten salts
- Magnesia

Electrolytic glass machining

Kragen Javier Sitaker, 02021-06-28 (updated 02021-12-30)
(6 minutes)

Electrolytic machining of glass may be feasible, using a sort of micro-scale variant of the chlor-alkali process in which the alkali-producing anode directs a stream of dilute near-boiling aqueous alkali against the glass, locally corroding its surface into soluble sodium silicate which is immediately washed away, carrying the ionic current toward the anode. This would be a very slow process, but may be useful for precisely shaping amorphous silicates.

As I understand it, the usual overall reaction of electrolysis of, for example, sodium bicarbonate works like this. The current is carried by Na^+ ions migrating from the anode to the cathode (in industrial practice, through a cation-exchange membrane), while the HCO_3^- ions left behind at the anode give up their electrons to the anode and produce CO_2, O_2, and water: $4HCO_3^- \rightarrow 4CO_2 + 4OH^-$; $4OH^- \rightarrow O_2 + 2H_2O + 4e^-$. At the cathode, meanwhile, the Na^+ ions aren't actually taking the electrodes from it; instead, they're taking hydroxyls from it, which the cathode is producing as $4H_2O + 4e^- \rightarrow 2H_2 + 4OH^-$. So the overall reaction is apparently $4NaHCO_3 + 2H_2O \rightarrow 4NaOH + 4CO_2 + O_2 + 2H_2$. This contradicts Simon et al., which says it's $2NaHCO_3 + 2H_2O \rightarrow 2NaOH + 2CO_2 + O_2 + 2H_2$, splitting twice as much water, but I guess you could totally have some arbitrary number of hydroxyl anions additionally traveling in the opposite direction through the cation-exchange membrane and thus totally split twice as much water, or ten times as much. Or maybe I miscalculated something above?

Anyway, so for glass ECM in this context you would have a tiny little "chlor-alkali" sodium bicarbonate cell whose cathode is a metal tube, like a hypodermic, nearly pressed up against the glass. Hot water is being pumped through the tube through a membrane, behind which you have a tiny pressure chamber, made of an insulator, full of ridiculously positively charged water. The water is pumped into it from where it's bubbling out oxygen and carbon dioxide after the anode destroys bicarbonate ions and gives the water that strong positive charge. The cathode neutralizes this positive charge and produces hydroxyl ions and hydrogen gas, which then come out its tip tens of microns away from the glass. After the alkaline water impinges on the glass it spreads out into a rapid current of some kind of buffer or acid, which neutralizes it and keeps it from corroding the rest of the glass. Maybe the acid is also produced by a similar kind of electrolytic cell, from sodium sulfate or something, so you don't need an acid reservoir. Maybe even the same electrolytic cell.

Now, how plausible is this? I was originally thinking it would require a totally implausible degree of macroscopic charge separation, with unrealistically large coulombic forces from the highly charged water on nearby objects. But of course the reason water is a good solvent for things like sodium ions is precisely that it solvates them with its high permittivity, screening all but a tiny amount of their electric field. And nickel, at least, can withstand contact with hot

caustic solutions. You might need to supplement the needle cathode with a high-surface-area reticulated non-graphitizable carbon electrode, and build the anode in the same way, to get high enough current. You might need to use a high overvoltage and thus get poor current efficiency. But it seems clearly feasible.

What if you ditch all the complicated plumbing and just immerse the glass in the hot sodium bicarbonate solution or whatever, and for your cathode use a solid metal needle insulated except for the tip? What happens if you pump a pulse of electrons into the cathode? Won't they produce hydroxyls that attract more Na+ to the area and enable it to etch the glass there? Won't they repel HCO3- ions from the area?

Well, I don't know. Maybe? It seems like it ought to work.

We might be talking about microscopic effects, though; some researchers ran some experiments on glass recycling in 02010 and it took them about 15 days to dissolve 25% of their glass samples, pulverized to 250–800 microns, in 1-molar NaOH at 70°. KOH was somewhat less effective, 5-M NaOH was slightly more effective, using a solution at only 50° was much less effective, and smaller particles sizes were significantly more effective, but overall we're talking about rates in the ballpark of 30 microns a day or 400 picometers per second.

Yet I've seen demonstrations of dissolving silica gel in 30% aqueous NaOH that converted amorphous silica gel to sodium silicate in a few hours. The silica gel in question didn't have any alkaline earth elements in it, I suppose, which Kouassi et al. mentioned as a factor that slowed dissolution for them. And the demonstrations I've seen where lye dissolved glass in seconds or minutes were with molten anhydrous lye, not aqueous lye, which would be something like "25 molar" and is also at something like 350° instead of 70° or 100°.

Topics

- Materials (p. 1138) (59 notes)
- Manufacturing (p. 1151) (29 notes)
- Electrolysis (p. 1158) (18 notes)
- Machining (p. 1165) (13 notes)
- Precision (p. 1183) (9 notes)
- ECM (p. 1186) (9 notes)
- Glass (p. 1254) (4 notes)

Super ghetto digital fabrication

Kragen Javier Sitaker, 02021-06-28 (updated 02021-12-30)
(33 minutes)

Suppose we have positional control precise to within a few microns, but not much force, power, rigidity, money, or access to pure materials. How can we leverage this into a comprehensive flexible digital fabrication capability?

Geometry

The standard ghetto metalworking processes are stick welding, bending, drilling, hammering, and angle grinding.

Digitally-controlled stick welding is maybe a bit difficult, but should be feasible, since "welding robots" have been a thing for decades; controlling the power supply should make it much easier to strike the arc with a TIG-like high-frequency start, you can avoid sticking the electrode because you don't need to enable high current until an impedance measurement shows the electrode has been lifted from the surface; voltage feedback should provide a trustworthy and low-latency indication of the arc length once it's struck; current and polarity duty cycle can be adjusted rapidly to control electrode melt-off rate; and precise toolpath control should dramatically improve weld quality. To the extent that you can substitute a CO2 hose ("MAG welding") for standard stick-welding flux, you can avoid slag that needs to be chipped or ground off; this should be no problem when welding on steel. "Short-circuiting metal transfer" as in MIG/MAG welding should be a possibility. This kind of process might allow "stick welding" using plain steel baling wire, and thus 3-D printing in mild steel.

Model-predictive control of the temperature distribution across the workpiece, with webcam feedback from surface temperatures, is probably critical to this. The dimensional precision of the result will probably be compromised by the contraction of the deposited metal, which is one reason welding is conventionally used for permanent assembly of prefabricated parts rather than large material buildups (though adequate toolpath planning could potentially reduce this problem), and the resolution will be limited by the surface tension of the weld puddle.

Bending of wire, rebar, or sheet-metal strip can be very precise if you have a good model of the material's work-hardening properties, and fairly fast and low-power as well. But this probably requires annealing the metal (to eliminate unknown work-hardened zones) and possibly keeping it hot during fabrication. Bending of sheet metal is normally done with sheet-metal brakes, press dies, or beading machines. Digital fabrication of press dies is now widespread and can often use cheap plastics.

Drilling metal is pretty high power and imprecise; it may not be very suitable for digital fabrication as a general technique, although digitally controlled drilling might be a useful supplement to other processes, for example to do initial piercing of a workpiece before

further processing. Manually drilling metal or wood after holes have been precisely started by a slower digitally-controlled process is also likely useful.

Digitally-controlled grinding should in theory be able to generate very precise geometry (submicron), particularly since the difference between contact and non-contact between a grinding wheel and the workpiece is not at all subtle; it is easy and fast to detect either the light from sparks (maybe 100 microsecond latency) or the sound (maybe 20 microseconds if you have a microphone on the workpiece). This should be able to give a pretty good indication of where the surface of the workpiece is, to within the error of your model of the grinding wheel's geometry, and you can go back and measure points on the workpiece in between grinding passes by detecting its conductivity. Large grinding wheels are high power when grinding continuously (a US$40 Black & Decker G720N 115mm-diameter 11000rpm angle grinder is 820 W) but not intermittently, and small ones need not be high power or even noisy.

Digitally controlled hammer forging of metal seems potentially very promising, especially since it could produce work-hardened results, but it involves a lot of variables and probably requires annealing steps.

Aluminum foil should be easy to cut to desired shapes under computer control with just a razor blade. If it's on a softish surface, running a small hard wheel over it under computer control should be sufficient to put ribs into it to give it a little bit of rigidity and define creases for origami panels. Automatically folding the origami or assembling the panels is probably pretty difficult, but doing it manually is certainly practical for initial prototyping. The precise origami shapes thus formed can then serve as tiny molds for inexpensive lightweight castable materials including candle wax, pine pitch, polyethylene cutting boards, and polypropylene bottle caps and rope. If desired, the aluminum foil can then be removed electrolytically with a saltwater or vinegar electrolyte or non-electrolytically with lye or muriatic acid.

Aluminum foil can also be cut with low side forces and tiny holes, but probably lower positional precision, with sparks from a copper point or graphite electrode. Normally you would do this in air but doing it under water or oil may provide better precision.

Mild sheet steel is easy to come by and should be easy to cut precisely but slowly with electrolytic machining with salt water, vinegar, or sulfates. Such cutting can include living hinges that make it easy to bend the resulting cut sheet into precise shapes.

Shapes roughly fabricated in nearly any metal can be brought to precise geometry by electrolytic machining or electric discharge machining, which produce no side forces and no heat-affected zone.

Electrolytic machining of glass may be feasible; see Electrolytic glass machining (p. 477).

Styrofoam is easy to come by and can be easily cut under digital control with a hot wire. Suitable wire materials that are easy to come by include copper, steel, stainless steel, and nichrome.

Rigid, brittle fine foams such as some of the materials mentioned

later offer enormous advantages for digital fabrication: they are as easy to cut to dimension as very soft materials like paraffin wax (though much more abrasive), or even easier, but retain the dimensional stability and refractory properties of the corresponding solid material, which are superior to even some hard metals, and their Poisson ratio is effectively 0; also, and related to their ease of cutting, they are much more resistant to fracture propagation than the corresponding solid materials. The only traditional materials with similar properties are Mykron/Mycalex and exotic mica-containing machinable glass-ceramics.

Corrugated cardboard is very abundant and has a spectacular strength-to-weight ratio and good dimensional stability in two dimensions; it can be cut with razor blades to form precise 2-D shapes and pressed with roller wheels to form precise crease lines, which can then guide origami assembly. Thicker corrugated cardboard may require higher-powered cutting with a wire saw or hacksaw. Cardboard's worst disadvantage is that it is very vulnerable to water.

Pottery clay bodies are readily available and very easily formed or extruded, although their stickiness can be a problem, and there is currently no software available that can model their behavior well enough to control their forming. If fired, they can produce hard, refractory, dimensionally stable materials with substantial strength, but if not fired, they can easily be recycled.

Digitally guided sandblasting ("abrasive jet machining") requires compressed air and consumable nozzles, but offers the possibility of cutting desired geometry into a wide variety of materials, even very hard ones, as well as selectively abrading surfaces, exposing fresh unreacted material. This produces such low side forces and heat loads that it can be used even on thin glass or eggshells, though getting it to work on aluminum foil might be too much to ask. Although it even works on metals, it is substantially more effective on brittle materials. Careful choice of abrasives can make this a material-selective process; in industry, for example, glass-bead blasting is used to remove paint without damaging steel, and dry-ice blasting is used for degreasing without damaging paint.

The easiest abrasive to use for sandblasting is of course silica sand, but crystalline silica dust causes silicosis, to the point where sandblasting with silica has been outlawed in many countries. (This doesn't solve the problem if you're cutting something like granite, of course, and granite is a very desirable thing to be able to cut because of its dimensional stability.) Sapphire, carborundum, apatite, metal oxides, or powdered waste glass may be preferable abrasives.

A different way to do digitally controlled abrasive cutting might involve passing a fabric loop loaded with loose, perhaps wet and sticky, abrasive across or through the workpiece. This is pretty similar to a wire saw, but I think the flat-tape shape of the fabric potentially provides a better tradeoff between breakage risk and kerf width. To make a given cut, the tension on the leading edge of the fabric needs to be just as high as it would be without all the cloth behind it, but if that turns out to be too high, the rest of the cloth doesn't have to rip as well.

Digitally guided 2-D plasma or oxy-fuel cutting is of course already a widespread process, and is much faster than most alternatives; laser cutting is making significant inroads here in recent years, but would be much more difficult to improvise.

The "Oogoo" Sugru-like silicone putty mix of cornstarch and hardware-store silicone caulk is thixotropic with a working time of minutes to hours (5 minutes with a 1:2 ratio, an hour with a 5:1 ratio), and thus eminently suitable for digital fabrication through extrusion or forming.

Plaster of paris can be foamed with baking powder, then cut to shape. It adheres well to quartz sand, and quartz sand or quartz flour can be useful functional fillers for it.

Charcoal is an easily cut material related to carbon foam (see below), but it tends to suffer pervasive cracking from thermal contraction during its formation, which may limit its uses.

Adding strength to established geometry, or otherwise switching materials

A lot of the processes in the previous section for converting digital data into a physical three-dimensional shape are only applicable to a narrow range of materials, often with fairly poor properties for anything besides being shaped. So it's important to be able to transfer a geometry fabricated in one material into some other material.

Hardware-store silicone caulk is a promising material for molding for a few different reasons. It's capable of holding detail down to the micron level; it's fairly inert once set, withstanding, for example, gasoline; it's thermally stable up to typically 300° (though the red high-temperature formulation commonly used for auto repair as "RTV" goes a bit higher); it's fairly dimensionally stable (though it does shrink a little, unlike some other silicones); and it's elastomeric. Being elastomeric makes it easy to pry it out of molds made of a harder material, and also makes it easy to peel molds made out of it off castings of a harder material. Also, it won't remain stuck to polyethylene or polypropylene, and reportedly also not to PVC or polycarbonate.

Shapes initially fabricated in aluminum foil can be thickened by electroplating/electroforming them, most easily in copper or brass. Although it is important for the copper to form a solid layer, it is not necessary for it to adhere firmly to the aluminum, as is usually desired in electroplating processes. Because the aluminum foil is the cathode in the electrolytic cell, it is possible to use electrolytes such as muriatic acid or lye that would normally destroy it immediately. A potentially larger concern is the risk of deformation from the surface tension of water, which can be reduced with surfactants and the substitution of alcohol for water.

Electroforming on non-conductive shapes is conventionally done with graphite powder dispersed in some solvent and painted onto the object. In some cases you can disperse the graphite in a solvent that softens or dissolves the surface of the object, welding the graphite to it; dichloromethane is reported to work for PLA.

Polypropylene can be sufficiently stiff for molding of plaster of

Paris or portland-cement concrete, which can provide a polypropylene part with much greater rigidity and thermal stability than polypropylene alone. In the case of plaster, it may be possible to later strengthen the plaster by filling internal channels in it with molten metals such as aluminum, brass, or cast iron.

Styrofoam forms, even those that can be cut by hot wires, can be stacked up into forms for low-temperature molding (for example, of portland cement or plaster of paris), or they can be wrapped in papier-mache-family strengthening materials such as cotton cloth soaked in plaster of Paris or fiberglass window screen soaked in non-alkaline sodium silicate.

Plaster shapes have little tensile strength or rigidity, but if they are suitably designed, automatically winding them in pretensioned wire can improve this. Steel wire has higher rigidity and less creep than copper wire but may be harder to find; aluminum wire is available by dissecting window screens, or the woven screening can be applied directly. Winding subsequent layers at different angles, as is done for glass-fiber pressure tanks, can provide tensile strength in two dimensions instead of just one. Brazing or soluble silicates may be suitable means for obtaining adhesion between layers of winding.

XXX stuccoing

High temperatures

A lot of attractive fabrication processes, such as firing clay, require high temperatures at some stage; so, too, does making many exotic materials (see section below). Making equipment that survives these temperatures requires refractory materials, often insulating refractories, although in some cases it's adequate to just use a pile of quartz sand (good up to 1500°, though not very insulating) or vermiculite (insulates better, I think good to 1100°). Aluminum foil can't resist high temperatures itself but is often useful for reflecting back radiant heat, preventing it from being lost.

Ordinary steel works up to about 1200° in a reducing atmosphere, but carbon dioxide is not sufficiently reducing; in air it starts to oxidize annoyingly rapidly above about 900°. Fired clay is the usual resort for temperatures up to 1100° or so; special clays can reach much higher than this but are harder to find and to fire. Fused quartz is maybe sometimes good to 1500° and usually to 950°, and is available for example in broken space heaters and halogen light bulbs, but it's very difficult to cut or form. (In some cases the quartz tubes are adequate.) Plaster of paris is easily formed before hydration, and can withstand a few excursions to over 1000°, but is not durable as a refractory.

Soluble silicates are hard to find (see below about making them), but can serve as adhesives for silicates such as quartz. Typically, in this use, rather than melting at high temperatures and falling apart, they form new compounds with the materials they're uniting.

Carbon foam is an excellent insulating refractory in non-oxidizing atmospheres (good to 3642°) and can be fabricated easily from bread dough or pancake batter, which is first heated to dry it and make it rigid, then heated further to carbonize it. It is very rigid and thus easy to cut, but abrasive. It does not adhere well to untreated quartz fillers.

Thermoplastics alone are not suitable precursors for carbon foam; enough thermoset ingredients such as gluten are required to prevent the object from losing its shape before carbonizing. If heated sufficiently in a non-oxidizing atmosphere it may graphitize and become electrically conductive, depending on its structure. Carbon dioxide is sufficiently non-oxidizing.

The standard insulating refractory for low-tech pottery kilns in an oxidizing atmosphere is a conventional pottery clay body (for example, ball clay tempered with silica and grog) filled with particles of a sacrificial-filler organic matter such as coffee grounds, sawdust, or used yerba mate, which burns out upon firing. In my experiments, material made with 67% sacrificial filler was quite solid but could be cut with a thumbnail, while material made with 89% sacrificial filler was still solid but friable and permitted easy gas passage. The firing process produces terrible odors.

Intumescent moldable "Starlite"-style coatings may be adequate insulating refractories for bootstrapping high-temperature capabilities. The precursor is an aqueous paint or paste of organic polymers (such as cornstarch and PVA glue, or wheat flour) and blowing agents. Sodium bicarbonate is commonly used as a blowing agent. Borax or boric acid substantially increases the strength of the resulting carbon foam, and may also help to cross-link PVA in the paste to prevent cracking from drying.

Silicone caulk may work as a precursor material for composites of graphite and carborundum, foamed or not, when heated. Acetic-cure silicone may cure more rapidly and foam in the process if carbonate or bicarbonate of soda is mixed in; I have not verified this. Oogoo confirms that it does cure more rapidly when mixed with cornstarch.

Even if you have an apparatus that can withstand heat, where do you get the heat?

The traditional approach for millions of years has been fire. Ordinary butane blowtorches can hypothetically reach 1970°, but usually don't, which is why you can't weld steel with them, and they have the inconvenience of producing a lot of exhaust. Oxy-acetylene torches are easy to buy (though expensive to refuel) and can reach 3500°, and oxyhydrogen torches are easy to make and can reach 2800°. Anthracite, and thus presumably charcoal, can reach 2180°.

But electric heating is much more convenient; it can be turned on or off (or anywhere in between) instantly, and it doesn't produce gases. Ordinary nichrome heating elements have maximum service temperatures ranging from 1000° to 1260°, though they don't melt until almost 1400°. Some varieties of Kanthal have service temperatures ranging from 1300° to 1425°, but these are harder to find. Halogen lamps, still available from auto parts stores as headlights even where they've been prohibited for household lighting, and their filaments may reach 2900°, but their envelopes are only designed to operate around 500° and are typically made of fused quartz, which melts at 1600°, or aluminosilicate glasses, which melt at only about 800°.

Carborundum "globar" heating elements are commonly rated to 1625° or 1600°, but are also not common household items, though it might be possible to make one; they consist of a carborundum tube

with a spiral cut in the central portion to increase its resistance, so that the ends that protrude through the refractory wall of the furnace can remain cool enough not to melt the metal wires that connect to them. The "Globar SR" design has a two-start spiral cut so that both electrical terminals are on the same end. Carborundum is seriously allergic to water vapor.

Historically, the fairly expensive yttria-stabilized zirconia was also used for globars; they melt at 2715° and have been experimentally used for heating up to 2100°. Possibly household ceramic knives could be used for this, though they might need to be cut to have a central "hot zone", similar to carborundum globars. One disadvantage is that they need to be preheated (for example with a flame) to become conductive; historically, their negative temperature coefficient of resistance was also a drawback (for example, in Nernst illumination lamps), since it means they require a constant-current source rather than a constant-voltage source to avoid thermal runaway. (Carborundum, by contrast, has a positive TCR above 700°, so this issue doesn't arise at normal globar service temperatures.) Nowadays current regulation, at least, is an easy problem to solve.

Nowadays, high-temperature heating elements are commonly instead the exotic $MoSi_2$ instead, which is serviceable to 1750° to 1850°. These are commonly used, for example, for sintering zirconia itself, which commonly requires 1530°–1700° depending on, among other things, sintering aids.

Alternative methods of electrical heating include arc heating with consumable graphite electrodes and induction heating, neither of which has an inherent temperature limit of its own; arcs in everyday US$200 plasma cutting torches commonly reach 20000°. Induction heating can keep the induction coils outside the hot furnace, and big induction heating coils are commonly made from copper pipe (at the high frequencies used for metals above their Curie point, only the skin of the coil can carry current anyway) with cooling water running through it. Induction furnaces in industry commonly maintain metal molten by heating the liquid metal inductively.

XXX microwave heating

Making exotic materials

Teflon and glass are crucial materials for their nonreactivity at everyday temperatures. Glass is widely available and, though it requires a lot of practice, can be shaped with a US$10 butane torch from the hardware store (or, traditionally, with an oil lamp and a blowpipe); teflon can be obtained from discarded laser printer fuser rollers, and a great deal of electical insulation is also made of teflon, but I do not know how to distinguish teflon insulation in discarded cables from the more common PVC.

Graphite is a crucial material for both electrodes and crucibles, the only viable electrode or refractory for many purposes; welding shops sell graphite electrodes, but they are graphite composites with poor stability in reactive environments. As mentioned above, some organics can be graphitized in a graphitizing furnace made of carbon foam and purged with carbon dioxide. This requires, I think, electric heating elements that can withstand graphitizing temperatures of

3000° (and pure graphite itself is the only plausible option), but are more conductive than the carbon foam itself. Even if the carbon foam is made from non-graphitizing carbon, it will conduct electricity once fired high enough.

Non-graphitizing carbon crucibles, which are more resistant to reactive environments than graphite, have been historically made from phenolic resin, then fired at 900° in an inert atmosphere. Other thermosets would presumably work too, unless they pyrolyze to graphitizing carbon (polyurethane foam, as found in pillows and spray-foam insulation, is a common precursor); if they don't outgas too much they might be able to make non-porous non-graphitizing carbon.

Soluble silicates, especially those that are neutral rather than strongly alkaline, are likely a crucial enabling material for digital fabrication for several reasons: they can be used directly as refractory adhesives (for example, to make a moldable insulating refractory from garden-store vermiculite); they bind very strongly to silica and other silicates; when dehydrated to solidity, they can be expanded from beads into glass foams by the application of heat, foaming as their water boils; and they can be instantly and directly cross-linked into insoluble silicates by the provision of polyvalent cations such as calcium or ferrous ions, as in the traditional colorful silicate garden, or with carbon dioxide, a feature that promises to be important for digital fabrication by selective solidification, perhaps even permitting 3-D printing of soda-lime glass. But soluble silicates are difficult to find, so we may have to make them.

(It may be possible to leach neutral sodium silicate out of corrugated cardboard.)

The most promising route to soluble silicates seems to be the digestion of powdered soda-lime glass with warm aqueous alkali over the course of hours or days. Alkali requires quite careful handling and can itself be difficult to obtain; the chlor-alkali process with graphite electrodes and a porous fired-clay diaphragm can produce it from table salt, salt-substitute potassium chloride, or alkali carbonates, and producing it thus *in situ* may be adequate for digesting the glass, thus avoiding any accumulation of hazardous alkali. Unwanted chlorine may be disposed of by passing it over hot aluminum foil. The traditional source for alkali is to leach it out of wood ash, but this is normally slow, dirty, bulky, and expensive.

Although discarded soda-lime glass is abundant, it doesn't handle thermal shock or reactive environments well; borosilicate glass is much more resistant, but hard to find. Adding borate (in the form of borax or boric acid) to discarded soda-lime glass seems like a promising thing to try.

Sapphire is aluminum oxide; industrially this is produced in the Bayer process by digesting bauxite with alkali to produce a soluble aluminate, then precipitating gelatinous aluminum hydroxide by cooling the solution (neutralizing it also works). This hydroxide (which crystallizes as gibbsite at a few microns per hour) calcines to sapphire, completely if held above 1200° for an hour; it is even possible to calcine the hydroxide gel to a transparent porous ceramic at 500° if you keep the electrolyte concentration near an ideal value,

though perhaps not if the hydroxide is derived in this way. Digesting (readily available) aluminum metal with alkali produces the same soluble aluminates, and so should be an easy route to sapphire for use as an abrasive, as a refractory (melts at 2072°), or for abrasion-resistant ceramics; the sapphire powder sinters to a ceramic around 1600°. Its thermal coefficient of expansion is an astounding 0.6 ppm/K at ordinary temperatures, though some sources give higher value such as 7 ppm/K.

The so-called "sodium beta alumina" that forms when heating sodium-rich gelatinous (?) aluminum hydroxide has fast ionic conductivity for a wide variety of monovalent cations, a property of great interest for its use as a solid electrolyte (and one which may be much more accessible than zirconia), used in the sodium-sulfur battery.

The transformation sequence from the gelatinous hydroxide to sapphire (alpha-alumina) is astoundingly complex (8, figure 4.1); the gelatinous form converts to the poorly ordered eta-alumina (aka gamma-alumina or gamma-prime alumina) around 375° (or 626°?), which converts to theta-alumina ("a better ordered transition form") around 800°, which finally converts to sapphire around 1120°, while if instead it is first crystallized as gibbsite (as is usual in the Bayer process) some of it instead goes by way of chi-alumina and kappa-alumina. Six other oxide and hydroxide forms are also potentially part of the process, depending on impurities and heating rates. Contamination with carbon dioxide in this process may result in incorporation of carbon.

The gibbsite itself is an important functional filler for plastics, providing strength and especially fireproofing.

Lucalox is another potentially important use for sapphire.

Sapphire can also be crystallized hydrothermally with soda ash above 400° and 200 MPa, but such pressures are challenging.

Another interesting material potentially derived from aluminum hydroxide is mullite, the acicular aluminum silicate that accounts for the legendary refractory performance of Hessian crucibles. Crystallization of mullite from amorphous $Al_6Si_2O_{13}$ at 980° has been reported; the preparation was a difficult process involving alkoxides of aluminum and silicon (and the phrase "was put in an open flask and stirred for three months"), but since mullite is the stablest alumina/silica compound, perhaps easier routes exist, such as just firing at 1100°.

Carborundum was initially discovered by heating sand in an iron crucible with an arc from a carbon electrode submerged in the sand, carbothermally reducing some of the silica with some of the carbon from the electrode. Heating mixed carbon and sand with a submerged arc sounds easy, and you don't even need the iron crucible; you just need two carbon electrodes.

HNO_3 was traditionally obtained by XXX

Control and actuation

Above I presupposed we could get precise positional control. But how could we *get* such positional control? Norbert Heinz has

demonstrated a series of excellent homemade CNC machining tools made from hardware-store parts, using the gantry arrangement used by most existing CNC machines. Some of them use H-bridge-controlled DC motors (or steppers) and optical quadrature encoders he has cut out of tin cans (or from paper) with optical sensors from old printers. The shaft rotation is translated to linear motion with leadscrews from hardware-store allthread; but this feedback measures only the rotational position of the leadscrew, so it is subject to errors from deformation of the machine frame and the leadscrew and from backlash, as well as thermal expansion.

To avoid these errors, you need feedback about the actual position of the end effector rather than motors that indirectly drive it. In theory optical mice have a resolution of a micron or two, but Heinz found they lose steps. He has achieved more precise control using the transparent plastic optical encoder strip from an inkjet printer to measure the linear position; these are usually 92, 150, or 300 lines per inch. I can't find Heinz's page on the topic, but in 02010 Michele Lizzit reported 33-micron precision using this method.

Industrial machine tools are now almost universally equipped with a "DRO", digital readout, or are fully automatically controlled. Three common kinds of DROs exist: using optical "glass scales" similar to the inkjet-printer encoder strip, using magnetic sensors that read a strip of alternating magnetizations, and using capacitive sensors that read a strip of alternating electrical connections. Digital calipers with 100-micron precision using this capacitive system are widely available at retail for about US$8, and are commonly equipped with an easily-tapped internal SPI data bus, while the other two systems routinely deliver micron precision.

These approaches can suffer from thermal error when the scale (glass, plastic, or otherwise) expands or contracts under the influence of temperature variations. The traditional solution to this was to keep the temperature of your metrology lab constant to within a tenth of a degree, but an alternative is to use laser interferometry, which can easily deliver submicron precision and is much less affected by temperature.

The kind of swing-arm arrangement used by manual magnetic 2-D profile cutting machines, or by hard-disk platter-and-arm arrangements, is mechanically vastly superior to the gantry arrangement; Melisa Orta Martínez's "Haplink" design demonstrates a promising mechanical design for adapting such a swing-arm arrangement to motor-driven cable drives.

For precise actuation over short distances, flexures and voice-coil actuators are probably the best approach. Hard disks have voice-coil actuators in them.

Differential roller screws ought to enable far more precise linear actuation than currently popular systems, but without digital fabrication, they are very expensive to build.

Topics

- Materials (p. 1138) (59 notes)
- Manufacturing (p. 1151) (29 notes)
- Machining (p. 1165) (13 notes)
- Ghettobotics (p. 1169) (12 notes)
- Bootstrapping (p. 1171) (12 notes)
- Clay (p. 1179) (10 notes)
- Welding (p. 1181) (9 notes)
- Precision (p. 1183) (9 notes)
- Foam (p. 1185) (9 notes)
- Ceramic (p. 1193) (8 notes)
- Refractory (p. 1225) (5 notes)
- Aluminum foil (p. 1237) (5 notes)
- Control (cybernetics) (p. 1262) (4 notes)
- Silicone (p. 1329) (2 notes)
- Sapphire (p. 1331) (2 notes)
- Sandblasting (p. 1332) (2 notes)
- Oogoo (p. 1349) (2 notes)
- Electroforming (p. 1372) (2 notes)
- Carborundum (p. 1381) (2 notes)
- Haplink
- Grinding
- Cardboard

Glass powder-bed 3-D printing

Kragen Javier Sitaker, 02021-06-29 (updated 02021-12-30)
(20 minutes)

In Dercuano I wrote a bit about a family of likely-feasible powder-bed 3-D printing processes where you print the shape by selectively depositing some "flux" in a powder bed (for example, by inkjet deposition of a dissolved aqueous binder, or by trickling granulated binder from a vibrating chute or screw extruder, like making a sand painting), then bake the whole powder bed to activate the flux, forming a solid object. I had done a few manual tests with 100-micron quartz flour and various different candidate fluxes fired at IIRC 1040°, getting some promising results.

Well, now I have a lower-temperature version of the process that I have a lot of confidence in, although I haven't tried it.

Lye-glass 3-D printing

The powder bed consists of sieved pulverized soda-lime glass, which is one of the candidate fluxes I'd tried at the higher temperatures. The flux is lye. The lye becomes very reactive when heated past its melting point of about 320°, capable of attacking soda-lime glass at corrosion rates millimeters per minute, and forms waterglass with the surfaces of the glass particles, which is less reactive and has a higher melting range than the lye; it bonds the glass particles together into a solid mass. This is too viscous and has too much surface tension to infiltrate the rest of the mass, so unfluxed glass particles remain inert; soda-lime glass generally doesn't soften below some 700°, so at this baking temperature, the glass won't stick together.

Generally molten lye is considered a material meriting the sort of respect we accord to molten steel, lava, or RFNA (unless we're cleaning our ovens, in which case we often treat it casually), but in this case we're dealing with very small quantities which exist for only a short time within the hot powder bed. If the glass granules have a size on the order of 100 microns, they might each occupy 0.6 nanoliters and be associated with an additional 0.4 nanoliter void space. If the lye is initially deposited as a 1-molar solution, that's about 4% by weight or 3% by volume, so we have about 0.01 nl of lye for each 0.6-nl glass granule, distributed among the "necks" connecting it to its neighbors; even the mass in the fluxed zone is 98% inert glass and 2% molten lye, and the overall powder bed might be more like 0.2% molten lye and 99.8% inert glass. A 1-kg printed glass piece might include some 20 g of lye before firing.

After baking the powder bed, first at 100° long enough to drive off any water and then at 350°–500° to sinter the lye-fluxed glass particles, depowdering the finished object should be straightforward, and the leftover powder can be sieved and reused. The total kiln time should be on the order of an hour, more for thicker powder beds through which the heat will propagate more slowly; slow heating and cooling may also be necessary to prevent breakage. Pressure will not

be needed to cause sintering, and may not be effective, since the rigid support from the unfluxed particles will prevent any significant compaction of the fluxed particles.

Variations

Although I'm now pretty confident that the above will work, there are a number of variations that might be better; some are sure things, others less so.

Aqueous lye thickeners, including no-bake waterglass

If the lye is deposited as a liquid solution, inkjet-style, it may be helpful to use ethanol as part or all of the solvent to reduce the surface tension and thus the size of the binder droplets, as well as accelerating evaporation. To prevent it from spreading out through the powder bed once deposited, it may be helpful to either use a very high lye concentration, like 50% (12M), or to include alkali-tolerant thickeners (especially thixotropic thickeners) to get high viscosity and avoid filling the void spaces in the powder bed. Such thickeners would become part of the final workpiece, which probably rules out most organic chemicals, since they would char and turn the glass black.

Being able to use lower lye concentrations in the flux would be valuable because concentrated lye solutions are super annoying.

Of course the most obvious thixotropic alkali-tolerant thickener to use would be waterglass itself, in which case you don't need the lye at all and may not need the baking step either. Exposure to carbon dioxide, either as a gassing step after printing or just from the atmosphere, may provide adequate strength.

Pure amorphous silica powder bases

Fused silica, silica gel, or silica fume (perhaps granulated) may be a more suitable powder base than the cheap-as-dirt soda-lime glass discussed above. Their absence of alkaline-earth metals enables them to react more readily with alkalis, they have much lower thermal coefficients of expansion, and they do not sinter on their own until higher temperatures, perhaps 1000°–1600°. But the resulting soda-silica glass is much more vulnerable to corrosion from water than conventional soda-lime glass. You might be able to add borax to the flux, perhaps reducing or eliminating the lye, to get a borosilicate glass instead of a soda-lime glass. But you need a *lot* of boria for borosilicate glass, like 8%–15%, and borax only dissolves at like 25 g/l at room temperature, up to 250 g/l at 70°. Using very porous powder bases like silica fume or dehydrated silica gel might ease this constraint, but will tend to produce a lot of porosity in the final print.

Desiccants in the powder base

Incorporating desiccants into the powder base is another possible way to prevent binder droplets from spreading out once deposited; they don't need to be stronger desiccants than the lye, just strong enough to diminish the free liquid volume somewhat and keep the lye from escaping. Promising desiccants for this purpose include silica gel itself, activated alumina (incompletely calcined aluminum hydroxide), quicklime, and zeolites.

Cristobalite or other crystalline silica polymorphs as the

powder base

A totally different powder base that perhaps could form waterglass with lye more easily than soda-lime glass or quartz sand (see below about the crude sand experiment) is cristobalite; Hachgenei et al.'s US patent 5,215,732 explains that at room temperature, cristobalite (or "tempered quartz sand" containing a mixture including cristobalite, the other high-temperature polymorph tridymite, and amorphous silica) is so much more reactive than quartz that you can convert it completely into waterglass by boiling it in 50% lye at 112°–146° for only three hours, even with a 2:1 ratio of silicon to sodium ("modulus"). Higher ratios of sodium to silicon naturally run faster. He gives the sand grain size as "in general, ... 0.1 to 0.8 mm," so the corrosion was advancing toward the center of the sand grains at around 30 nm/s, times or divided by four. He reports being able to thus "temper" quartz by calcining it "above 1000° C., preferably at 1300° to 1600° C., with the addition of catalytic quantities of alkali", and, presumably, not annealing it back to alpha-quartz.

Blowing powdered flux onto a powder bed bound with a temporary binder

My original flux-deposition notes contemplated depositing the flux as a powder, for a fully dry 3-D printing process. It's desirable to use as fine a powder as possible, because the grain of the powders limits the dimensional precision of the workpiece; even if you deposit your flux with 10-micron precision, each 100-micron-wide flux particle that becomes part of a surface is going to produce roughness on the order of 50 microns. Similar concerns apply to the powder-bed particles: if your flux particles are sticking together 100-micron-wide particles of aggregate, you're going to get surface roughnesses on the order of 100 microns. But using fine floury powders for the powder bed isn't problematic; however, fine floury powders of flux tend to clump up into larger lumps, as the surface forces between particles overwhelm their weight and inertia.

I've written about various approaches to solving this problem in Precisely measuring out particulates with a trickler (p. 384), but a new one occurred to me in this context. If you can blow the flux powder through a nozzle with a compressed air blast, you can do an excellent job of breaking up aggregates, because the aerodynamic forces on the particles tend to scale with their surface area, just like the surface adhesion forces they're fighting, not with their volume. So you can blow very fine dust through a quite fine nozzle, although once you're down in the sub-ten-micron range you start having safety concerns. But of course if you're blowing that air blast onto a powder bed, those same aerodynamic forces will tend to disrupt the powder bed, which is why I'd never considered this solution before.

The objective here is for the powder bed to be an unbound, loose powder after baking, except in the places where enough flux was deposited to enable it to bake to a solid. So you'd think that including a binder in the powder bed would be totally counterproductive. But if we use a *sacrificial temporary* binder that bakes off during the baking process without leaving a residue, it could still work!

Suitable sacrificial temporary binders might include water (as used in traditional fired-clay ceramics), ethanol, gasoline, kerosene,

toluene, turpentine, d-limonene, acetone, ethyl acetate, dimethyl sulfoxide, isopropanol, formamide, hydroxylamine, sulfur, naphthalene, and various salts of ammonium (chloride, carbonate or bicarbonate or carbamate, acetate, sulfate); most of my notes on these are in file `inorganic-burnout.md` in Derctuo.

Some of these candidate sacrificial binders, like water and ethyl acetate, are ordinarily liquids, which means that the powder bed is sort of more of a paste bed; the powder base might be premixed with the liquid, so that the recoater trowels on one layer after another like drywall mud or construction mortar. This could lead to geometric disturbances from subsequent recoating layers, though that might be solvable; the usual practice with construction mortar is to minimize this problem by including barely enough water in your mortar to make it plastic, and as soon as you trowel it onto the wall, it loses enough water to lose its plasticity.

Alternatively, the recoating could be done with a powder recoater in the usual way, which is gentle enough not to perturb previous layers, then misted with the sacrificial binder liquid.

Other candidates, like ammonium chloride and ammonium carbonate, are ordinarily solid, so if the base powder particles are bonded together by them, it becomes a solid object, which avoids the issue of geometric disturbances when recoating, but poses the problem of how to get the sacrificial binder into the powder. If it's just mixed in as separate particles, it won't be form bonds between the base powder particles that are adjacent in their final position. Ammonium carbonate is water-soluble; it could be sprayed onto the base powder once it is in position, or the base powder could be mixed with an aqueous solution of it, and in either case we would need to evaporate the solvent to get to solidity. Ammonium chloride has an additional power: it can be "sublimed" at convenient temperatures, so it could be analogously infused into the powder once it is in place without needing a liquid solvent. Or it can be formed in situ by reacting ammonia with muriatic acid.

See Powder-bed 3-D printing with a sacrificial binder (p. 506) for more variations on the theme.

In-situ temporary binder creation

Some candidate temporary binders, like muriate of ammonia, can be produced in situ in the powder bed by applying two different reagents at different times; in that case, ammonia and muriatic acid can be infiltrated into the powder bed in gas form, one after the other, where they will react to form the salt.

Salt as flux

Some of the alternative powder-bed bases such as cristobalite or amorphous quartz offer the possibility of using ordinary muriate of soda as a less annoying alternative to lye that activates at higher temperatures. It has been used for salt-glazing of pottery for 600 years. At 1100°–1200° in a steam atmosphere it forms muriatic acid gas and lye; the former escapes, driving the reaction forward, while the latter fluxes the silica as before.

Wikipedia claims that silicates of iron are even more effective

fluxes for this purpose, so reduced (ferrous) iron helps even more, and red clays are well-known to be lower-firing; however, ferrous silicate on its own is fayalite olivine, and olivine is the canonical high-melting mineral at the high-temperature extreme of Bowen's reaction series. But I think *fayalite* olivine may indeed be low-melting; a 01993 abstract in Science gives its melting point as 1478 K, which is only 1205°. So some ferrous iron in the powder bed may help the process along.

Soda ash

Higher-temperature powder-bed bases might also be able to use soda ash as a flux. Soda ash is water-soluble, even cheaper than lye, less annoying to handle, and melts at only 851°. My experience melting it with a butane torch suggests that it has an alarming tendency to bubble, perhaps because it is slowly converting into lye. This produces only carbon dioxide gas rather than muriatic acid.

Iron as flux

If iron silicates are crucial to the fluxing effect in the salt-glazing of pottery, perhaps metallic iron or some iron salt could form these iron silicates directly with soda-lime glass at lower temperatures than those necessary for salt-glazing. Ferric chloride melts at only 308° but has a very narrow liquid range, while green vitriol starts to decompose into high-melting hematite at 680°.

Wood ash

If transparency of the glass produced is not a concern, wood ash might be a possibility; it would surely be the cheapest flux for powdered glass, if it works. It is mostly carbonates, oxides, and hydroxides of alkali and alkaline earth metals, including lye; leaching out the water-soluble components will tend to eliminate the counterproductive polyvalent cations. Traditionally this was done by washing the ash on top of linen cloth, then boiling down the results to reasonably pure potassium hydroxide.

Crude sand experiment

I did a crude and deadly kitchen experiment the other day which seems to have successfully made a little waterglass from quartz and lye. I placed a layer of damp construction sand in a thin stainless (or nickel-plated?) metal bowl, sprinkled a layer of lye flakes liberally on top of the center of the layer of sand, then pressed down another layer of damp construction sand on top. I covered the bowl with aluminum foil, placed a paper towel over the aluminum foil, added an aluminum-foil skirt around the edge to reduce the loss of radiant heat from the bowl's sides, gently heated it on a gas stove burner (maybe 500 W) until the lye flakes stopped crackling from the release of water. Then I turned the burner up to max (maybe 1500 W) for an hour; halfway through I moved the bowl over a bit to ensure that there were no cold spots that never got heated. I left the sliding-glass door open so that the draft would carry any fumes from the bowl away from me.

Unfortunately I don't have a thermometer. Some of the aluminum foil skirt around the sides of the bowl strayed into the gas flame and

melted, but on the various occasions during the hour when I inspected the crude apparatus, no part of the bottom of the steel bowl itself was ever glowing visibly, so it had not reached the 525° Draper point. When I turned the flame off, tore open the aluminum foil, and poked the sand with a chopstick, the end of the chopstick charred and smoked, but didn't burst into flames, so the top of the sand was probably somewhere between 250°–350° at that point. The sand had formed a hard mass, infiltrated by the molten lye, although not, as it turned out, around the edges of the bowl; only in the middle. Presumably the sand in the bowl was hottest on the bottom where it was separated from the flame only by a thin layer of steel, with a net heat flow in at the bottom and out at the top producing a thermal gradient through the sand.

Upon cooling I was able to use the chopsticks to lift up a large monolithic aggregate of sand that extended all the way to the bottom of the bowl; at its bottom and top surfaces its color was the beige of the sand, but in between, where the lye flakes had been, it was much more white. No intact lye flakes were in evidence; they had all completely melted, so that part of the sand had exceeded 320°.

I broke off a small piece of the aggregated sand with the chopsticks and dropped it into a polypropylene bottlecap to which I added a few drops of water; it disintegrated immediately, indicating that the binder was probably mostly lye, not mostly waterglass. A couple of flakes of aluminum foil added to the bottlecap fizzed enthusiastically, confirming the presence of free lye. So at this point I had no indication that any waterglass had been formed.

I neutralized the rest of the sand by soaking it with kitchen vinegar; after letting it stand a while, I added a pinch of baking soda, which fizzed, confirming that the pH had been brought down below neutral. This ensured that any lye had been not only dissolved in water but converted to highly soluble sodium acetate; at the same time, the lower pH would make any waterglass present almost entirely insoluble in water.

Stirring around the sand with my fingers, I found that most of the aggregated chunks had disintegrated immediately upon wetting or were easily broken up. However, one irregular chunk of aggregated sand of a centimeter or three in every dimension remained intact, indicating that it was completely bound together with something other than frozen lye, almost certainly waterglass. Handling it wet did not leave my fingers slippery, providing further confirmation that free lye was not present.

Topics

- Materials (p. 1138) (59 notes)
- Digital fabrication (p. 1149) (31 notes)
- 3-D printing (p. 1160) (17 notes)
- Experiment report (p. 1162) (14 notes)
- Powder-bed 3-D printing processes (p. 1226) (5 notes)
- Glass (p. 1254) (4 notes)
- Fluxing

Rator-port GUIs

Kragen Javier Sitaker, 02021-06-29 (updated 02021-12-30)
(26 minutes)

How do we bring the goodness of RPN and command-line history into a multitouch world?

I've been thinking about this problem for a few years, and I think my thinking has evolved to the point where it's probably reasonable to implement it. I think it will make it easy to implement a lot of really cool and versatile exploratory computation user interfaces.

Basic interaction paradigm

I'm imagining some data values scattered around a two-dimensional workspace, maybe zoomable, each one represented by some kind of graphical display, which I'll call a "port".

Maybe the data are numbers, or formulas, or programs, or images, or files, or data tables, or locations in memory in a program being debugged. One of them is the "focused port" (this has some similarities to the WIMP concepts of "current selection", "current window", "current tab", and "focused widget") and is highlighted to indicate this, and there are some operation buttons and knobs displayed for it, maybe outside of its actual display area, as well as some generic operations applicable to all objects. If you click or tap on another port, the focus shifts to that other port, and the previously focused port becomes the "background port", which is also highlighted in a secondary way.

The ports do not contain other ports; there is no hierarchy of ports. Data values can certainly contain other data values.

I am agnostic as to whether these ports are rectangular, round, or some other shape (although I keep imagining them as being round, with round buttons scattered around them), or whether they are overlapped or tiled or flowed together like sentences in a paragraph, or whether they are all composited together on a single canvas, as in KSeg or Flash. But it does need to be possible to tap or click on individual ports.

The data values themselves are conceptually immutable; at any given time, each port is associated with a single data value, but at different times that value will be different. So ports are stateful but data is not.

Generic user interface operations applicable to all ports include resizing, moving, cloning, closing, and undo/redo. These might be accessible from a menu in a fixed location on the display and/or with fixed reserved keys. Each port has associated with it an undo/redo tree, which you can navigate to restore it to values that it has had at some time in the past. Cloning a port creates a new port with the same data value and a copy of the undo/redo tree, which is then focused, but from then on the two ports evolve independently. Closing a port adds it to a list of recently closed ports, from which it can be reopened in case of regret.

So ports have state, but it is entirely under user control: ports cannot change state irreversibly or destroy information, because you can always undo a state change, and you can copy a port, go back in time, and see what would have happened if you hadn't taken the road less traveled by.

Unary rators

Some of the buttons for the focused port compute a function of the port's current value and change the port's value to the new value, adding a node to its undo/redo tree, and these normally include some kind of preview of this new state in their display. Examples include incrementing or negating a number, deleting a column from a data table or sorting it by a column, navigating to a child or parent node in a filesystem or JSON hierarchy, and perhaps appending a letter to a string (if we consider each of the letters on the keyboard to be a separate "button for the focused port"). If you invoke one of these and regret it, you can go back in the undo tree; sometimes you will then want to clone the port and redo the undone operation, so that you have both the old and the new value visible. Some other buttons similarly compute such a function but open a new port to display the result in; you might have both kinds of button for the same function. These are best thought of as conveniences for common cases, especially where you might want to invoke several such operations in a row starting from the same state; the user can always get the same effect by manually cloning the port before invoking the operation.

(Maybe it would be fun to call these buttons' operators "rators" or "rations".)

Sometimes a rator might return *both* a new data value for its own port *and* a data value to display in a new port — a random number generator, for example.

Rators with continuous parameters

Some other rators interactively adjust one or more continuous parameters of the current port, like a slider widget does. That is, they compute a new value that is a function of the port's current value and some set of continuous quantities. This might be implemented by dragging from a button in one or two dimensions while watching the live preview of the result. Examples include changing a numerical value, changing the brightness or contrast of an image (or cropping it), changing the width of a column in a data table, or rotating a 3-D object. Once the user finishes adjusting the value, a new state is added to the port's undo tree.

It might be better to use a second finger to drag to adjust the parameters while the button is being held, which both permits continuous parameters with more components and reduces the clash with the one-finger scroll idiom in touchscreen interfaces.

Convergent data flow with binary rators combining two ports

All the data flow described so far is divergent: you can make lots of "copies" of a value and "modify" each of them in different ways, but

there's no way to combine multiple values displayed in different ports into a single value. But some rators take a parameter that is another port. For example, the addition or division rator on a number or formula might take another number or formula as a parameter, the differentiation rator on a formula might take a variable as a parameter, the move-to rator on a file might take a directory as a parameter (or alternatively the move-into rator on a directory might take a file as a parameter), the concatenation rator on a program might take another program as a parameter, and on data tables the union and intersection rators take another table.

This requires some kind of data value typing to give the user feedback about which ports are plausible parameters. It might be best to display a preview of the rator's result on each of the possible visible operand ports, graying out those that are not possible, so the user can tap the one they want. If they tap a grayed-out port, they should see an explanation of *why* it is not compatible; in this form, such compatibility checking doesn't require static typing, only dynamic. Normally the whole operand-selection thing should be quasimodal, only active as long as the user is holding down the operation button.

(WIMP interfaces sometimes use drag-and-drop for this kind of two-operand operation, but I think that's clumsy, and it's profoundly incompatible with one-finger scroll on touchscreens.)

Lifecycle behaviors

There are at least five different plausible user interface "lifecycle behaviors" for invoking such a binary operation:

- The focused port transitions to display the result, and the operand port is closed. This adds a special node to the undo/redo tree of the focused port so that undoing will reopen it, and the user need not manually restore it from the list of recently closed ports.
- Similarly, but the operand port remains unchanged.
- The operand port transitions to display the result, but the focused port remains unchanged. This is sort of like the HP-9100A behavior.
- Similarly, but the focused port is closed, although it's hard to imagine when that would be useful.
- The result is displayed in a new port, which becomes the new focused port when the button is released. Both of the original ports remain open and unchanged.

I think in most cases #1 should be the default, beause that's what RPN does, but it seems likely that different options will be best for different cases.

The "background port" mentioned earlier now comes into play; if you press and release the operation button without explicitly tapping an operand, the background port is taken to be the operand if it is compatible. (You can always undo if that wasn't what you wanted.) There's a whole invisible focus history maintained so that, when the background port is closed, the most-recently-focused still-visible port becomes the new background port; this is just the RPN stack. This is also used to set the new focused port when the focused port closes.

Which of the 5 lifecycle behaviors mentioned above is the right default likely depends on what kind of repetition you're most likely

to want when you're invoking the operation. It's very useful to be able to invoke the same operation repeatedly on the same or different operands while still holding down the operation button, but there are lots of possible meanings of "invoke the same operation". Suppose you have the number 1.21 focused and you are selecting its multiplication button. Here are the kinds of repetition you get if you select the operands 2.50, 4, and 2.10:

- All four numbers are collapsed into a single product, 1.21 × 2.50 × 4 × 2.10 = 25.41.
- Same, but the other three numbers remain visible.
- Each of the other three numbers is replaced by its product with 1.21, handy if you're computing a 21% VAT.
- If the operation somehow remains active while the focused port goes away, all four numbers are collapsed into the same single product as in #1, but in a different order.
- Similar to #3, but both the original price and the price with VAT are visible.

In cases #2 and #3, it even makes sense to repeat the same operation with the "same" operands, since one of them now has the result of the previous repetition.

It probably makes sense to offer #5 as an user option with a gesture of dragging from the selected operand off into empty space, particularly in cases where #3 is the default. That is, if you just tap on an operand port, that port transitions, but if you drag from it, the new data value appears in a new port where you dragged it.

Part of the question here is how common various kinds of "linearity" are, linearity in the sense of Girard's linear logic or linear type systems. If a particular kind of value, like a number, tends to be used only once, it is most convenient for operations on it to remove it from the canvas, leaving only the operation result, rather than requiring you to manually remove each intermediate value. On the other hand, perhaps joining two data tables through a one-to-many relationship normally leaves the table on the "many" side of the join intact, waiting for further joins to be applied to it.

(Henry Baker noted the connection between linearity in this sense and the life cycle of values on an RPN stack.)

The gulf-of-execution problem for binary rators

The operand feedback suggested above may not be particularly useful if no compatible ports are visible on your canvas, creating a large gulf of execution. Say you have a formula focused and you try to differentiate it, but the differentiation operation takes a variable as its operand, and there aren't any variable ports visible. So you press the differentiate button, all your formulas go gray for a moment, and then you release it and nothing happens. How are you supposed to know how to open a port of the right type? This is the kind of problem VB-style forms UIs excel at.

In this particular case, maybe you could fix the problem by opening a menu of the formula's free variables, although note that the UI

idioms discussed so far offer no way to open such a menu.

Currying

Computations with more than two inputs can be handled by currying: providing each of the extra inputs in sequence. This potentially worsens the gulf-of-execution problem, because when you provide the first input you don't have any way to see what the type of the second input will be, or that there even is a second input. This is sort of like a "wizard" or a "dark pattern".

So, for example, to equijoin two data tables, you might tap an unary equijoin rator that one of the two tables displays on one of its column headers, which opens a new port for the next step in the join. The new port might display the same data table, maybe with the column in question highlighted and some information about the column's data type and typical values; a binary rator on the join port allows you to select a table to join with, which transitions the join port to displaying data from the columns of the second table, each with an unary rator to select each of them as the join column, and maybe some outer join options. Tapping one of those unary rators transitions the join port to displaying the join result. At any point, even long after completing the join, you can "undo" to go back to a previous step and change the options.

This kind of program-controlled sequencing is anathema to event-driven GUI thinking, according to which the user should be in control of what order they provide the information to complete an operation.

Programming by example with rator names

Suppose that each rator has a consistent name, though perhaps not one that is displayed prominently in the UI. Then we can imagine writing a transcript of a part of a session, automatically assigning a hidden variable name to each port involved:

```
v0 := Number new.   "Create a number for the VAT."
v0 set: 1.21.       "Adjust its value with a slider."
clone(v0, v1).      "Clone the port so we can use it twice."
v2 := v2 * v0.      "Multiply the first price by VAT."
v3 := v3 * v1.      "Multiply the second price by VAT."
v2 := v2 + v3.      "Add them together."
```

This slighly silly transcript contains three free variables --- `Number`, `v2`, and `v3` — which are in some sense inputs; and it ends with one unconsumed result in `v2`. So, when you are interacting with concrete values in this way, you are also "programming by example" or "programming by demonstration", building up a script that could be later applied to different input data, in a way similar to John W. Cowan's system "Mung".

(Maybe `Number` is really a constant rather than an input.)

You could imagine converting such a single-output transcript into a new rator (keyboard macro recording), or automatically snipping it into separate rators at the boundaries where new inputs appear. The standard facilities for output preview and operand compatibility

scanning can apply automatically to these new rators just as they apply to built-in rators.

This may run into some difficulty with the idea that the applicable rators might be derived from the data; for example, a data table view might have a unary rator to sort by the Rating column and another to sort by the Price column, but the same data-table-viewing code will have a different, arbitrarily large number of applicable rators when it's looking at a table with a different number of columns. I think we can solve this by allowing the "rator selector" to include arbitrary immutable data rather than just being a symbol in the way my Smalltalk-syntax example above suggests.

How do you know in what ports a new binary rator thus defined is applicable? At first, I thought this might require static typing, but as long as we can run code peremptorily without harm, I don't think it does. The above script requires inputs `v2` and `v3`. Suppose we should offer it on a port's menu whenever the current state is suitable for use as `v2`, which is to say, it has a `*` rator that can accept our `v1`, which is the Number 1.21; we can determine this by trying to run it, and checking to see where it crapped out. If it croaked trying to read the not-yet-defined `v3`, then we're probably good, but it shouldn't be a menu option if it didn't even get to trying to read `v3` and instead died on a previous line because `v2` doesn't have a `*` (or someone hacked Number and its instances no longer support `set:`).

If you want conditionals in pure programming-by-example form, you can get them with assertions and fallbacks, but I'm not sure how to do loops. So it might be best to reify the macros in the interface so you can apply operations like "if" and "while" to them.

Dependencies and external state via paint functions

Each data value provides not only the transition functions invoked by rators, but also functions that the user interface shell uses to paint its port and rators. So far most of my examples have concerned only painting data pulled from the data values themselves, like the column widths of a table view, or column names and data cells from the underlying immutable table it refers to. But we could imagine that the paint functions can also read mutable state that lives outside the rator-port system, like the list of running processes on the system, or the contents of the filesystem, or the history of IRC messages on a channel. So the rator-port system can give you undo and cloning and stuff for everything inside it, but you might only be using it as a cyberdeck through which you interact with the stubborn and refractory traditional world. (Even the set of available rators on a port might depend on the state of the external world; consider navigating to a filesystem subdirectory.)

If we allow paint functions, or some of them, to read the mutable state of the outside world, then we might as well allow them to read the mutable state of the ports as well, and thus data values to contain pointers to ports. In this case, we could imagine, for example, having some data values representing formulas in which the variables are pointers to ports, and having the increasingly-poorly-named paint function evaluate those formulas recursively. Then, if we change the

formula in one of the ports, all the ports whose formula refers to it will also update their display on the next repaint. (Which we ought to be able to arrange to happen quickly.)

Note also that the rator functions are themselves just pure functions, so it ought to be feasible to incorporate them, or even a PBE script invoking a sequence of them as described above, into a paint function in that way. (Of course that's essentially how the system's results previews work.)

At this point it becomes necessary to cache the evaluation of parts of a paint function that depend on mutable state ("nested transactions"), so that you don't re-evaluate the same formulas exponential numbers of times for each repaint. And this brings us to the question of background computations, progress, cancelation, and futures.

Background processes

Everything above has supposed that all the computation is fast enough that it's always effectively instant, but of course that is a bad assumption for a user interface, especially in latency-sensitive environments like touch and especially VR/AR. In practice we can't miss video frame deadlines simply because a formula is slow to recalculate! But everything described so far is purely side-effect-free, except for consuming input events, port transitions, and updates to the cache; so it's always safe to discard and/or restart a computation in progress, as long as we don't consume whatever input events inspired it. So probably we can keep painting responsive by just having the display of some ports update after several frames, and maybe giving them a "fast paint" function whose results can be used when the slow paint function is slow.

However, we also might want to run background computations that take a while, like solving a large system of equations, maybe even in separate OS processes or on separate machines. I think the simplest way to handle that is to treat the *output history* of such a background computation as "mutable external state", so a port can display that history in whatever way it chooses: scrolling text, progress bars, progressive image enhancement, whatever. And if the background processing happens internal to the rator-port system, well, aborting the computation is just another event to add to the log. It's guaranteed not to corrupt anything else.

Not Actors

These ports are very similar to Hewitt's Actors, but there are some significant differences. Like Actors, ports have a current state, which can include references to other Actors, and a state transition function that specifies how their state should change in response to external events.

But ports cannot send messages, either to other ports within the system or globally, and they cannot create new ports. And they have a fixed set of methods, rather than a single "invoke" operation or an arbitarily large set of methods: one to paint, and one to return a list of currently valid rator selectors and their corresponding closures. (Or maybe there's a third method to invoke a given rator selector.)

Moreover, ports don't manage their own state; their rators return a new data value and some kind of indication of where to stick it, but that new data value might end up being the new state of a *different* port, and at any rate you can time-travel the ports independently to any previous or later state. They can't count on other ports being in a state that is in some way mutually consistent.

(This suggests that maybe another reason that the default lifecycle behavior should be #1 or #2 is that it's maybe kind of bad to set that data value as the new state of a different port, since it will wreck whatever display preferences the user had set up. But how else will we arrange to set the same background color on three different drawing objects conveniently?)

Modality-agnosticism and graceful degradation

This doesn't have to be limited to multitouch; the abstraction level of invoking rators with operands on a bunch of ports to get them to transition to new states means that most of your code would be fine on a terminal.

If you're writing a paint function for something that's basically some text, what you really want is to call a library function with either the literal text or a lazy stream of the text, and have it produce the painted window and maybe some scrollbar interaction stuff, and then just return it. Then you could run that same paint function in a terminal interface by passing in a different library of paint-function building blocks.

It's important to structure the user interface libraries such that most of your code doesn't care about pixels and touches.

Taming external processes with snapshots

If we can take snapshots of external process state and record and rewind and replay it, then we can apply the whole undo/redo/clone thing to computations carried out by external processes as well, as long as we confine their I/O well enough; we can treat their behavior from one input to the next as a "pure function". Batch-mode external processes don't even need snapshots; we only need to "snapshot" the filesystem before they start and after they stop, potentially using something like Docker.

Topics

- Human-computer interaction (p. 1156) (22 notes)
- Composability (p. 1188) (9 notes)
- Real time (p. 1195) (7 notes)
- Higher order programming (p. 1196) (7 notes)
- GUIs (p. 1216) (6 notes)
- Reverse Polish notation (RPN) (p. 1243) (4 notes)
- Zooming user interfaces (ZUIs)
- Rewind replay

Sulfur jet metal cutting

Kragen Javier Sitaker, 02021-06-30 (updated 02021-12-30)
(6 minutes)

Could you cut metal with a jet of hot sulfur vapor in a way analogous to oxy-fuel cutting of iron?

Sulfur boils at under 450° and reaches several hundred kPa of vapor pressure by 600°, and it reacts exothermically with most metals, with adiabatic flame temperatures in the 1000°–2000° range at atmospheric pressure. Sometimes the metal sulfides melt lower than the metals, and invariably much lower than the oxides; ferrous sulfide, for example, melts at under 1200°, much lower than iron's 1538°, wüstite's 1377°, or hematite's decomposition at 1539°. Nickel sulfide melts at 797° and boils at 1388° (though I'm not sure you can just react nickel with sulfur), while metallic nickel doesn't melt until 1455°, and its oxide not until 1955°. Chromium sulfide melts at 1350°, but metallic chromium doesn't melt until 1907°, and its oxide (chromia or viridian) not until 2435°. Tungsten disulfide melts at 1250°, lower than tungsten trioxide's 1473°, while metallic tungsten doesn't melt until 3422°.

Even in the cases where the sulfide is higher-melting than the metal, it's usually not by much. Cuprous sulfide melts at 1130°, slightly higher than copper's own 1085°. Aluminum sulfide melts at 1100°, considerably higher than metallic aluminum's 660°, and boils at 1500°; by contrast, sapphire doesn't melt until 2072°. Even titanium yields a bit: its sulfide melts at 1780°, while titania holds out to 1843°. Tin(II) sulfide melts at 882°, while its principal oxide (the tetravalent) doesn't melt until 1630°.

So maybe you could use sulfur-jet cutting to flame-cut not only iron and steel but also low-chromium stainless, tungsten, copper, brass, bronze, and aluminum.

Generally the metal sulfides transform to oxides when heated in air, not vice versa, so we can't expect the jet of hot sulfur vapor to be very good at removing the oxide. Still, though, it might be good enough; no oxygen atom stolen from the oxide by the sulfur vapor is going to stick around long enough to reform the oxygen.

Unlike oxygen, you could maintain your sulfur in solid form when not using it, and you don't need a separate ignitor or fuel; upon heating past 450°, you would get a jet of vapor which ignites spontaneously in air, though with a low adiabatic flame temperature of something like 1200°, and producing vitriolic fumes. You could perhaps heat the vapor to a much higher temperature by passing it through a heated ceramic nozzle, or perhaps a heated chromed metal nozzle, if it turns out that viridian is sufficiently resistant to sulfidation. If you can convert the sulfur vapor into a plasma (for example, an inductively coupled plasma), you might be able to increase its reactivity and attack oxide coatings that would normally resist it; maybe even just burning a bit in air will be good enough for that. Also, the hot SO_2 in the flame will try to reduce oxide coatings.

If you really want to increase the temperature of the flame, mixing some metal powder into the hot sulfur would do the trick. The easiest way to do this would be something similar to a solid-fueled rocket engine made of solid sulfur with substoichiometric quantities of metal powder mixed in, ideally aluminum. As the reaction proceeded, it would expel superstoichiometric sulfur through the nozzle along with sparks of metal sulfide. Ignition temperatures are in the 350°–550° range.

This has the unfortunate feature that, as with solid-fueled rockets, there's no way to turn it off once it's started, which is less than ideal for flame cutting of metals. Maybe you could do something similar to a hybrid rocket motor, using a combustion chamber containing solid metal fuel in a mostly-sulfur atmosphere. As you pump more sulfur into the chamber, it heats up and expands, traveling out the nozzle; some of it also reacts with the fuel to produce more heat. The chamber can't have an all-sulfur atmosphere, and the sulfur can't be forced to travel through the hot fuel to get to the nozzle; in either case all the sulfur will be consumed instead of producing the desired metal-cutting stream.

In terms of health hazards, the whole system is kind of nasty. You're producing a stream of vitriol as you cut, and the slag that melts out of the kerf is a metal sulfide, which will produce stinky and poisonous hydrogen sulfide in moist air thereafter, possibly for years. And iron sulfide, at least, has been well known for its pyrophoricity since antiquity, but a thing that surprised me is that sometimes it's so pyrophoric that it ignites when exposed to air.

It lacks some of the safety hazards of oxy-fuel systems, though. Unlike with methane or acetylene, it's impossible to have a leak of sulfur vapor that builds up an explosive gas mixture over time before finally exploding; if the vapor is concentrated enough to burn, it's also hot enough to do so spontaneously, because the auto-ignition temperature is 230°, and boy did that surprise me the first time it happened to me. In rare cases, sulfur dust can make air explosive. Aside from the fire risk, cold sulfur and even molten sulfur are relatively safe materials; a ruptured tank would not produce an explosion, asphyxiation, or render nearby objects highly inflammable the way oxygen does. There are no high-pressure bottles in the system that can explode or act as rockets.

Topics

- Materials (p. 1138) (59 notes)
- Manufacturing (p. 1151) (29 notes)
- Machining (p. 1165) (13 notes)

Powder-bed 3-D printing with a sacrificial binder

Kragen Javier Sitaker, 02021-06-30 (updated 02021-12-30)
(12 minutes)

I was writing Glass powder-bed 3-D printing (p. 490) and came up with the idea of "powder-bed" 3-D printing that is really more of a layered paste bed, and I realized that that idea itself is maybe a lot more broadly applicable than to the lye-fluxed glass-powder bed processes with full-bed baking prior to depowdering I was exploring there.

In particular, I was exploring the idea of powder-bed 3-D printing with two binders: a "sacrificial binder" that maintains the integrity of the powder bed during the printing process but is later removed, and a selectively deposited permanent "flux" binder which defines the geometry ultimately produced by the process, and which perhaps is not active until after all the layers are patterned. My objective there was to allow the flux to be blown in dry powder form onto the powder bed without disturbing it, but now I see many other potential advantages of this design approach.

One variant for printing in concrete is to trowel on plastic layers, made of a thixotropic aqueous paste of sand; water; perhaps reinforcers like chopped basalt fibers or straw; a water-soluble temporary binder like sugar, carboxymethylcellulose, clay, or gelatinized starch; and perhaps a plasticizer, which might be the same as the temporary binder. Then, on each layer, selectively deposit a permanent cement such as waterglass, slaked lime, plaster of Paris, magnesium oxychloride, calcium aluminate, or ordinary portland cement. This requires the layers to be thin enough, relative to the mobility of the cement molecules, that the cement can diffuse down to the previous layer before it sets, or they require you to deposit the cement vigorously enough to penetrate the whole layer. In the case of a slaked-lime binder, you'd have to perforate an array of air holes at the end of the process to allow CO_2 to diffuse to all the layers. Such objects wouldn't need to be baked; they'd just need the uncemented material to be washed away with a water hose once the cement was set.

Another variant is to pour layers of liquid into a vat or pit, one by one, allowing hydrodynamics to level each layer rather than using a mechanical recoater; after pouring each layer, you solidify it with the sacrificial binder (for example, by allowing excess water to depart into porous surroundings, or allowing a solvent to evaporate, or by allowing enough time for thixotropic network formation to gel it, or by spraying on a pH-changing agent to activate a pH-sensitive gelling mechanism) and then selectively deposit the permanent binder, which might be a construction cement like those mentioned above, or might be a sintering aid like those discussed in the note mentioned above and in Dercuano, or might be something else.

Another variant is to deposit the layers using spin-coating, permitting extremely fine and precise layer thickness control. In this

case, you could still pattern the layer by selectively depositing a different material on it, such as a catalyst, but another possibility is to pattern it by using light or particle beams to cause some kind of change in it, like a multi-layer version of the standard photolithography process, or stereolithography on a solid substrate rather than a vat of liquid. For example, you could draw a pattern on the layer with a moving laser, move a bar of light sources across it analogous to how a xerox machine or flatbed scanner scans an image or how an LED printer prints one, press it up against an LCD that selectively permits light to pass through, project a pattern onto it optically with a lens from a projector, or press it up against a thin mica window through which an electron beam is passing. Once all the layers have been patterned, maybe you need to bake it, and then you can remove the undesired parts of the block, for example by immersing it in a solvent.

If you were using, say, UV light to pattern the layers in this way, most of the usual stereolithography resin concerns apply: you need a component that reacts to the UV, a possibly different component that does something like polymerize or depolymerize in response, and something that blocks the UV from reaching the next level down, which has already been patterned --- a UV-opaque pigment, but not so much of it that the effect only reaches partway through the layer. The mix might include other components as well, for example to affect the mechanical or electrical properties of the final product, and these might vary from layer to layer.

In the case of patterning with electron beams, which might offer the possibility of deep submicron 3-D printing, adjusting the electron energy may be a more reasonable way to set the penetration depth to the layer height than mixing in varying amounts of opaque pigment.

If you specifically spin-coat potassium silicate rather than photopolymers, your "sacrificial binder" can be the dried potassium silicate itself, while you can inkjet-print the pattern on each layer with an aqueous solution of a polyvalent-cation salt such as calcium acetate, which will cross-link the potassium silicate into insoluble alkali-lime glass. Once all the layers are printed, if it's been adequately protected from CO_2, you can redissolve the dried but uncrosslinked waterglass with hot water, leaving a 3-D-printed object in alkali-lime glass. This may be adequately precise and transparent for fabricating optical lenses; 100-nm resolution in Z should be easily attainable, and resolution in X and Y will depend on the precision of positioning and inkjet deposition, easily reaching 30 microns. Other things to selectively deposit might include aqueous lead acetate, which will not only make the glass insoluble but significantly alter its refractive index and dispersion; water-soluble dyes to incorporate into the final piece; and various kinds of paint that are intended merely to adhere to the final workpiece rather than to react with it, such as conductive copper-filled paint.

For cases like that, where the permanent binder consists of polyvalent cations, you might be able to do your patterning with an array of transition-metal electrodes in contact with the layer being patterned, rather than with actual ink jets. By putting a positive charge on one of the electrodes, you anodically dissolve some of it and pump the resulting (generally polyvalent) cations into the workpiece.

This requires that the workpiece have high enough ionic conductivity for this to be acceptably fast.

Doing things exactly backwards, you could spin-coat layers of some viscous solution of some salt with polyvalent cations, like calcium acetate, and then print "inks" onto them containing water-soluble compounds that abruptly stop being water-soluble once they see polyvalent cations; waterglass is one example, but sodium alginate is another, much gummier example, while e.g., ammonium phosphate is an example that can form extremely strong and refractory minerals with many polyvalent cations, and plain old sodium carbonate can form carbonate minerals with many polyvalent cations. So you could imagine, for example, a single spin-coated layer of zinc sulfate (maybe mixed with other water-soluble solids), on which one of your inkjet nozzles can create the biocompatible mineral zinc phosphate, another can create zinc carbonate (smithsonite), a third can create fluorescent-green zinc silicate (willemite), and a fourth can create the biocompatible antibacterial hydrogel zinc alginate, all with submicron precision at least in Z. After you've finished printing out your array of multi-material objects in a block of mostly zinc sulfate, you can wash away the unreacted zinc sulfate. And maybe you could have other layers that provide other polyvalent cations.

If you're printing in a pH-sensitive gel like some of the carrageenans (or, again, waterglass), you can maintain the pH at a level favorable to gelation of the whole layer during the printing process, but then pattern each layer by loading some places on it up with a bunch of buffering agent to *keep* it favorable to gelation. So, for example, if your gel is stable at acidic pH but dissolves in basic environments, you could dump a bunch of a citrate or acetate buffer into the parts you want to keep; or, if it's stable in basic environments but dissolves in acidic ones, you could dump a bunch of borate buffer in there. Then, once you're done printing all the layers, you can immerse the block in a base (or, respectively, acid), which will immediately dissolve all the unprotected parts, while taking a much longer time to attack the printed object.

Returning to the powder-bed context, we could consider the problem of iron powder metallurgy. We can build up a powder bed of iron, layer by layer, and deposit a binder in it that works as a sintering aid (in Dercuano I suggested graphite, copper, or iron phosphide), and then bake the result to sinter the iron; but iron is not very rigid at sintering temperatures, and will tend to sag. Suppose we mix some dehydrated active alumina in with our iron powder, and repeatedly compact the powder bed during printing enough to kind of squish the iron particles together with the alumina particles, forming a friable but solid block. Now, when we bake this solid block, the iron gets gummy and soft, but the alumina particles remain as firm as ever, and have enough contact with each other to prevent any macroscopic deformation. When the loaf comes out of the oven, the fluxed part of the block is no longer friable; it's a solidly connected network of welded-together iron particles with an interpenetrating network of alumina particles. Light wire brushing removes the unsintered volume, leaving the metal part, and lye, muriatic acid, or oil of vitriol can destroy the alumina within it,

leaving the iron part with great porosity. (Infiltration may solve this, or it may be desirable.)

Alumina particles may not be the ideal "sacrificial binder" here; they're a very poor binder, so they have to occupy a lot of the volume to work at all. There are sol-gel processes for producing alumina gels, and forming some kind of gel to encapsulate the base powder particles could allow firmer holding with much lower volume; the most common sol-gel processes are aqueous, which would be a disaster for iron powder, but some take place in other solvents that won't attack the iron. But there may be better binders available.

For example, silica gel deposited from tetraethyl orthosilicate might do a better job of holding the iron particles in place. In its crude form its polymerization takes far too long, but perhaps if the iron powder is mixed with dry glass fibers of length comparable to the particle size, even slight polymerization would form a continuous network. Amorphous silica can handle iron's sintering temperatures for the little while that's necessary, and then you can remove it with molten lye or hydrofluoric acid without damaging the iron.

Shrinkage is already a big problem in ordinary pressed powder metallurgy, and the sacrificial-binder approach can solve it almost completely, at the cost of increased porosity.

Topics

- 3-D printing (p. 1160) (17 notes)
- Waterglass (p. 1189) (8 notes)
- Powder-bed 3-D printing processes (p. 1226) (5 notes)
- Cements (p. 1235) (5 notes)

Stochastically generated self-amalgamating tape variations for composite fabrication

Kragen Javier Sitaker, 02021-07-02 (updated 02021-12-30) (26 minutes)

Duct tape or electrical tape is a great way to make things; for example, wallets, pipes, connections between pipes, handles, hammocks, shoes, masks, baskets, purses, canoes. It's really easy to make and remake a variety of surface shapes, either with an underlying form or without one, or to seal connections; and the result remains open to revision. But when the tape is only sticky because it uses a constant-tack adhesive, the resulting object never fully solidifies, is subject to creep, and can't handle high loads. There's something called "self-amalgamating tape" which avoids this problem to some extent by bonding to itself and solidifying once it's in place, but I think this is generally a potentially much wider-ranging family of materials.

There's a thing called "UD tape" or "unidirectional endless fiber reinforced tape" which is an existing higher-end version: thermoplastic tape, 50–500 microns thick and 3–165 mm wide, reinforced with lengthwise carbon, glass, or UHMWPE fibers. Typically an automatic tape layup machine ("ATL") automatically laminates together several layers of the UD tape on a flat surface, and then the resulting flat sheet of composite is pressed and heated to fuse the thermoplastic together. Then it can be cut to final 2-D shape, for example with a waterjet, and thermoformed in a hot press. Or, instead, it can be wound around a form to form wound shapes like wound-fiber tanks. The matrices/binders are thermoplastics; common thermoplastics include polyethylene, polypropylene, PET, nylon 6, polycarbonate, PMMA, or, for stronger results, PEEK or nylon 12.

And of course prepreg sheets for fabricating carbon-fiber reinforced plastics or similar materials are pretty much the same thing: a flexible woven fibrous sheet of carbon fiber is pre-impregnated with resin, stored at low temperature to prevent full curing, and activated by bringing it up to room temperature after it's incorporated into a laminate.

There is a very wide spectrum of composites following the general pattern of some sort of flexible sheet (woven or otherwise) combined with some sort of adhesive. As I pointed out in file `globoflexia.md`, it includes the balloon-covering kind of pâpier-maché, as well as most carbon-fiber and fiberglass construction, and it's especially effective if used to add rigid shells to easily-shaped foam cores, forming a sort of sandwich panel which can gain extra strength from surface curvature. If you can somehow cut the sheet into a strip and precombine the adhesive with it, maybe you can use it in the same convenient way as duct tape, but with much more permanent results.

If you're precombining the adhesive for convenience, though, you

need the roll of precombined tape to not self-amalgamate into a solid cylinder before you use it. There are different ways you could potentially achieve this. The adhesive could be activated by contact with air, if you store the tape in an airtight box. It could be activated by spraying it with some kind of chemical. It could be activated by the process of peeling it or stretching it as you unroll it. It could be activated by the humidity cycling of the air. It could be activated by externally applied temperature, as by a heat gun or sunlight. It could be activated by pressure, as scratch-and-sniff books are, for example by breaking open micro-encapsulated reagents in the tape by burnishing it with a burnisher. It could be activated by ultraviolet light, as by sunlight. And (for me the most exciting possibility) it could undergo self-propagating high-temperature synthesis, where once the object is in the shape you want, you light it (for example, with a match, a blowtorch, or a magnesium strip) and a hot chemical reaction propagates through the material, converting it into a different material.

You can use a lot of different possible flexible sheets as bases for the tape: ordinary cotton cloth, cotton duck (as in duck tape!), paper, glass-fiber cloth, fiberglass window screening, aluminum window screening, steel window screening, carbon fiber, basalt fiber, woven or laminated music wire, ceramic fibers like alumina and zirconia, woven polyester, sheet polyester like Tyvek, woven glass rovings as in printed circuit boards, Kevlar or other aramids, UHMWPE fibers, carborundum fibers, aluminum foil, gold leaf, boron fibers, quartz fibers, polyimide film, boPET, burlap, hemp cloth, cotton-candy sugar fiber, flax, silk, glass foil, or foil or cloth made from the common thermoplastics mentioned earlier, for example. For any of these that are fibers, it may be useful to make the fibers unidirectional or nearly so, rather than evenly distributed in two or three directions, as in the case of thermoplastic UD tape mentioned above.

If the permanent binder isn't active until after the tape has been applied, you may need some sort of "sizing" to hold the binder onto the base sheet, or in the case of a base sheet made of unidirectional fibers, just to hold the fibers of the base sheet together. (Often in "unidirectional" fibers for composite panel layups, there is "stitching" or "weaving" of either polyester or the same fiber material for this purpose, so they are only, say, 90% unidirectional.) Such sizings might also be useful for holding layers of the tape in place before the permanent binder is activated, especially if the sizing is intermittently placed, permitting direct layer-to-layer contact, unlike the continuous adhesive layers in duct tape and electrical tape. Sizings might include conventional pressure-sensitive or constant-tack adhesives like those used in duct tape; PVA; other water-soluble polymers such as gelatin, carrageenans, sodium polyacrylate; the common thermoplastics mentioned earlier; soluble silicates; "drying oils" such as linseed or poppyseed oil; soluble polymers such as celluloid and shellac; pine pitch; clays; soluble salts such as the chlorides of sodium, calcium, or magnesium; sugar; gelatinized starch; or thermosets such as epoxies, phenolic resins, or polyurethanes. In some cases only very light bonding might be needed, so even very gentle bonds like sublimed ammonium chloride might work as a "sizing"; this has the advantage that it can be rapidly

infused into a great volume of tape at a reasonable temperature without introducing any water, which is advantageous if water would prematurely activate the permanent binder.

Candidate permanent binders include all of those listed above as sizing candidates, and also geopolymers, plaster of Paris, portland cement, silicones, phosphates, and brass.

Comments on some candidate tape systems of those described above

I wrote this program; it blindly generates random selections from 4,744,806 possible tape systems.

Program to generate random combinations

```
#!/usr/bin/python3
import random

thermoplastics = '''
polyethylene
polypropylene
PET
nylon 6
polycarbonate
PMMA
PEEK
nylon 12
'''.strip().split('\n')

fibers = '''
cotton
glass fiber
carbon fiber
basalt fiber
music wire
ceramic fiber
polyester
Kevlar
UHMWPE
carborundum fiber
boron fiber
quartz fiber
hemp
cotton-candy sugar fiber
silk
'''.strip().split('\n')

fibers.extend(x + ' fiber' for x in thermoplastics)

films = '''
paper
fiberglass window screening
aluminum window screening
steel window screening
sheet polyester like Tyvek
```

```
woven glass rovings as in printed circuit boards
aluminum foil
gold leaf
polyimide film
boPET
burlap
glass foil
'''.strip().split('\n')

films.extend('woven ' + x for x in fibers)
films.extend('unidirectional ' + x for x in fibers)
films.extend(x + ' film' for x in thermoplastics)

sizings = '''
common pressure-sensitive adhesives
PVA
gelatin
carrageenans
sodium polyacrylate
waterglass
linseed oil
poppyseed oil
celluloid
shellac
pine pitch
clays
sodium chloride
calcium chloride
sugar
gelatinized starch
epoxy
phenolic resin
polyurethane
ammonium chloride
'''.strip().split('\n')

sizings.extend(thermoplastics)

binders = '''
geopolymers
plaster of Paris
portland cement
silicone
calcium phosphate
brass
lead-tin solder
silver solder
latex paint
'''.strip().split('\n')

binders.extend(sizings)

def imagine_a_tape():
    tape = random.choice(films)
    if not random.randrange(3):
```

```
        tape += ' and ' + random.choice(films)

    if not random.randrange(2):
        tape += ', sized with ' + random.choice(sizings)

    tape += ', with a permanent binder of ' + random.choice(binders) + '.'
    return tape[0].capitalize() + tape[1:]

if __name__ == '__main__':

    print("A random selection from the %d possible tapes:" % (len(films) * (1+len
(films)) * (1 + len(sizings)) * len(binders)))
    for i in range(16):
        print(imagine_a_tape() + '  ')
```

Commentaries on some combinations

I ran the program to see if it would come up with anything reasonable.

Unidirectional nylon 6 fiber, with a permanent binder of silver solder.

This would not work in its raw form, both because you need some kind of sizing to keep the fiber together and keep the solder on the fiber, and because activating the silver solder requires heating it up far beyond the melting point of the nylon. If you added some kind of refractory sizing, like potassium silicate, then probably you could get the sizing to both stick the powdered silver solder to the tape and roughly hold its shape as the nylon burned out, perhaps even up to the melting point of the silver solder (some 741°), though that's kind of pushing it. Using potassium silicate rather than sodium silicate would allow you to re-wet the tape if it dried out.

The benefit of this sort of thing is that you could make free-form jewelry out of it, then solder it into final shape once you were satisfied; or, you could stick it onto things that you wanted to silver-solder together that were in positions where loose bits of silver solder would just fall off. But I can't help but think that unidirectional nylon 6 is nearly the worst base material for these purposes.

Aluminum window screening, sized with linseed oil, with a permanent binder of clays.

You can definitely get clay to stick to a strip of window screening with linseed oil, and as long as this is kept in an airtight container. If there's a bit of grog in the clay to add porosity, you might even be able to get the linseed oil to solidify all the way through the shape. Then you have an "all-natural" free-form thin clay surface shape which doesn't need to be fired to cure, at least if aluminum is natural enough for you. You might be able to burnish it to a nice finish after it cured.

If you did try to fire it, the aluminum would burn out, but the resulting thin eggshell of fired clay would probably still have enough strength to stand up, at least if it didn't slump too much in the kiln. But if you were going to do that, you'd probably want to leave out

the linseed oil and use water instead.

Woven cotton and aluminum foil, with a permanent binder of polyurethane.

The cotton layer would bear the mechanical load of the tape, while the aluminum foil would make it reflective on one side, especially to infrared. You'd probably have to use a thermoplastic polyurethane and "cure" it with heat somehow, rather than using one of those polyurethanes that polymerizes when it's exposed to air.

Unidirectional music wire and woven hemp, sized with gelatinized starch, with a permanent binder of phenolic resin.

This is probably not a good mix. Unidirectional music wire tape is probably a useful thing to make, and I guess if you wove hemp through it you could keep it from coming apart? The starch would probably interfere with the resin curing, phenolic resin probably would be too brittle for anything you'd want music wire for, and the thermoset curing process for the phenolic would probably soften the music wire.

Woven polypropylene fiber, with a permanent binder of clays.

You could probably use an oil to get the clay to stick to the polypropylene cloth, especially if it was loosely woven, and this could maybe give you a "plasticine tape" that you could make things with, sealing the different layers together with some pressure. It might be hard to unroll from its totally-stuck-together state without ripping all the clay off the fabric. Getting water-based clay to stick to polypropylene would be more difficult, but if you could do it, you could build fairly free-form thin structures that could then be either dried or dried and fired. Normally I'd suggest that you could keep layers of claycloth from sticking together on the roll by putting a thin smooth plastic sheet between them, but also normally that plastic would ideally be polypropylene.

Unidirectional glass fiber, with a permanent binder of latex paint.

No, I don't think that would be a useful combination. You'd need some kind of sizing to keep the glass fiber together, and generally you'd want either a fairly beefy permanent binder like geopolymer cement or epoxy, or a more accessible and convenient fiber like nylon.

BoPET, sized with gelatin, with a permanent binder of PEEK.

Maybe you can get gelatin will stick to PET if you activate the surface with a corona-discharge plasma first? The way you'd activate PEEK would be by blasting it with heat, which would shrink the boPET by making its orientation less biaxial, so this would be sort of like a shrink-wrap tape kind of thing. I guess that would squish the (presumably granular) PEEK around whatever the tape was wrapped around, so that when the PEEK melted it would be in contact with itself. Except that I think the PET and gelatin would just totally melt away, and maybe burn, long before the PEEK started to soften.

Maybe you could activate it with some kind of solvent that softens the PEEK but doesn't attack the boPET? Nothing attacks boPET.

Woven PEEK fiber, sized with clays, with a permanent binder of PEEK.

I guess you could draw PEEK into fiber, though I haven't heard of anybody doing it, and if so you could wrap a tape of PEEK cloth around something tightly a bunch of times and then turn a heat gun on it. Maybe if the cloth was impregnated with clay, maybe like glossy magazine paper, that would increase its viscosity enough to keep it from melting onto the floor before you'd finished melting it together. But probably a better way to thus hold it in place would be to mix the PEEK fiber with some other fiber that was totally unharmed by PEEK-melting levels of heat, like glass fiber. Or nylon? Does nylon stay solid at PEEK-melting temperatures? Polyimide would definitely work but would be expensive. Maybe you could use an oven-bag-style *layer* of nylon (or polyimide) on the back of the tape, but make it full of small holes to permit the layers of PEEK to melt together.

Unidirectional glass fiber, sized with polyethylene, with a permanent binder of lead-tin solder.

This is definitely a thing you could make. In fact, glass-fiber UD tape in a polyethylene matrix is available from multiple vendors right now; all that's lacking is granulated lead-tin solder as a filler in the polyethylene. Although I haven't tried it, I think the solder won't bond to the glass fiber, no matter what the temperature, and heating it up enough to flow the solder onto whatever copper pipes or electronic connections you're interested in will burn up the polyethylene, leaving the glass fibers embedded in this solder mass but not strongly bonded to it. But the glass fibers would still be undamaged.

So maybe you could use this combination to coat an entire vertical surface with solder, or the outside of a copper tank, or something.

Woven PET fiber and unidirectional carborundum fiber, with a permanent binder of carrageenans.

Well, this would certainly be very strong, and the carrageenan could probably stick the carborundum to the polyester cloth well enough to keep the tape intact. You could store the tape in dry form, then spritz it with water to activate the carrageenans once you'd wound it around whatever tank you had. But you'd probably be better off with a stronger binder, and maybe a more refractory one, too, because the virtues of the carborundum are probably going to be wasted with such a weak binder.

Unidirectional polycarbonate fiber, with a permanent binder of nylon 6.

I don't think you can do this because I think polycarbonate melts lower than nylon 6, and I can't think of any solvents that will dissolve the nylon but not the polycarbonate. The other way around would work, though, especially with a little adhesive of some kind to stick layers of the tape together, and it might be a good way to make free-form shatterproof plastic surfaces.

Woven carbon fiber and unidirectional quartz fiber, with a permanent binder of pine pitch.

This is silly: two exotic refractory high-strength low-creep health-hazard fibers and a "permanent binder" of high-creep low-temperature low-strength colophony, whose principal advantages are its low toxicity and wide accessibility.

Woven polyester, with a permanent binder of silicone.

This could work. The commonplace single-component silicone caulk cures by absorbing moisture from the air, so if you keep this in a hermetically sealed container, the tape you pull out will always be fresh and sticky. The polyester cloth (I now realize I have a duplicate in the above list) allows you to wrap a thin layer of silicone around just about anything, and it compensates significantly for silicone's low strength.

Still, I'm not sure what I'd use it for where I wouldn't just use the silicone.

Aluminum window screening and unidirectional carborundum fiber, with a permanent binder of sugar.

Haha, no.

Unidirectional PMMA fiber, with a permanent binder of polyethylene.

This would almost definitely work; you'd hot-press a UD tape layup to get a strong, shatterproof sheet that could absorb an enormous amount of impact energy and wouldn't suffer even in highly reactive environments. Or you could wind it around a form and then heat it up with a heat gun to fuse the layers together. I've never heard of PMMA fiber though, just acrylonitrile.

Woven cotton-candy sugar fiber, sized with common pressure-sensitive adhesives, with a permanent binder of silver solder.

Yeah, no.

Polycarbonate film, with a permanent binder of linseed oil.

How would you cure the linseed oil through the polycarbonate film?

Woven glass rovings as in printed circuit boards, sized with polypropylene, with a permanent binder of sodium chloride.

This wouldn't work; you could spray it with water and dissolve the salt, but neither the water drops nor the recrystallized grains would interact with the polypropylene-coated glass fibers. Without the salt, it would be glass-fiber-reinforced polypropylene sheet, but in a form that was hard to thermoform.

Unidirectional polypropylene fiber, with a permanent binder of lead-tin solder.

Nope, the PP melts too low.

Unidirectional UHMWPE, sized with waterglass, with a permanent binder of linseed oil.

I don't think the waterglass will stick to the UHMWPE, because nothing does. And the linseed oil is too weak to be useful here. This

won't work.

Polypropylene film, sized with common pressure-sensitive adhesives, with a permanent binder of sugar.

So it's tape that you wet to make syrup? No.

Woven carbon fiber, sized with nylon 6, with a permanent binder of shellac.

You could activate it by spritzing it with alcohol, dissolving the shellac out of the nylon, I guess. So it would be a relatively easy way to do a carbon-fiber layup, and squirting alcohol on it is a pretty easy way to "amalgamate" it. The shellac isn't very strong at all, but in some situations it wouldn't have to be.

Unidirectional basalt fiber, sized with sodium chloride, with a permanent binder of gelatinized starch.

Basalt starch tape with salt in it to keep it from growing mold, I guess. All natural! I guess you activate it by getting it wet? And remove it the same way? At a very small scale this might be a good way to repair damaged old books, but maybe with copper sulfate instead of the sodium chloride.

BoPET, with a permanent binder of PMMA.

I'm pretty sure you can activate this by spraying it with DCM if there are holes in the BoPET, which you would want so that the PMMA can weld abundantly from layer to layer of tape. You could probably just deposit a film of PMMA on one or both sides of the boPET.

Paper, sized with sugar, with a permanent binder of linseed oil.

This sounds like decoupage, although I don't know what the benefit of the sugar would be. Maybe it facilitates wet-folding origami?

Nylon 12 film, sized with clays, with a permanent binder of celluloid.

I guess the film would be heavily perforated, maybe in a honeycomb pattern of 1-mm holes spaced 2 mm apart. So you'd get a nice strong biaxial bond, and then you'd wet it down with something to dissolve the celluloid (I forget what dissolves celluloid but I bet it doesn't hurt nylon) and let it redeposit as a very stiff, rigid, lightweight plastic sheet. Which explodes if you get a spark on it. I like it except for the clays.

Woven nylon 12 fiber and glass foil, sized with PMMA, with a permanent binder of pine pitch.

No, that makes no sense.

Unidirectional nylon 12 fiber, with a permanent binder of common pressure-sensitive adhesives.

That sounds like strapping tape, minus the plastic backing.

Unidirectional Kevlar, with a permanent binder of linseed oil.

No, that's ridiculous.

Woven Kevlar, with a permanent binder of sodium chloride.

That's even more ridiculous. You can build furniture and shields with it that withstand bullets, but they fall apart if you spill your Coke.

Woven glass rovings as in printed circuit boards, sized with linseed oil, with a permanent binder of PVA.

None of these three materials are usefully compatible with any of the other two.

Woven polycarbonate fiber, sized with linseed oil, with a permanent binder of carrageenans.

This sounds like expensive and smelly pâpier-maché.

Less random thoughts

In tape systems where you need to apply heat to join the layers together permanently, as in the various kinds of thermoplastic-matrix UD fiber-reinforced tapes, it can be extremely inconvenient to do so externally. A possible solution is to "print" a grid of a self-propagating high-temperature synthesis system on one surface of the tape, a grid of little squares or hexagons whose edges are printed with, for example, a stoichiometric mixture of iron powder and sulfur, or a stoichiometric balance of aluminum foil and nickel foil separated by, for example, a layer of zinc. These systems, once ignited, will burn rapidly, producing a high temperature but no gas. Where they're sandwiched between two layers of tape, each with a thermoplastic surface, they will melt together the thermoplastics in their vicinity. Although the tapes along the grid line itself will be separated by the waste products of the reaction and thus will not form a bond, on both sides of the weld line they will be quite firmly welded together.

Deliquescent substances like calcium chloride may have a useful role in activating water-activated binder systems like calcium sulfate, by absorbing water from the air and making it available to the binder in their vicinity. I'm not sure that will work in that form; it might run into the same kinds of difficulties perpetual-motion machines do. In cases where the lack of a suitable solvent was preventing a reaction, though, deliquescence can definitely bridge that gap. This might work, for example, for producing calcium phosphates from diammonium phosphate and calcium chloride, initially mixed together as dry powders, but gradually, after deliquescing, irreversibly reacting to form calcium phosphates, which can serve as binders under some circumstances.

Water-activated permanent binders like slaked lime, portland cement, and plaster of paris can't be coated onto the backing with water-based "sizings". You need to use some kind of anhydrous or almost-anhydrous approach. Maybe a polymer like polystyrene dissolved in a nonpolar solvent like acetone, for example, or shellac in ethanol, would work for this sort of "sizing", as long as there isn't so much present that it will waterproof the permanent binder particles or keep them from being able to interact. Above I also suggested subliming ammonium chloride into the tape to deposit some relatively inert salt crystals to stick things together.

Topics

- Materials (p. 1138) (59 notes)
- Manufacturing (p. 1151) (29 notes)
- Filled systems (p. 1161) (16 notes)
- Python (p. 1166) (12 notes)
- Composites (p. 1187) (9 notes)
- Anisotropic fillers (p. 1218) (6 notes)
- Humor (p. 1292) (3 notes)
- Tape

Spin-coating clay-filled plastics to make composites with high anisotropic filler loadings

Kragen Javier Sitaker, 02021-07-02 (updated 02021-12-30)
(4 minutes)

Platy clay crystals, for example of bentonite, are common functional fillers in filled polymer systems, among other things because they decrease permeability (if well bonded to the polymer matrix) they provide enormous strength. Each clay grain has quartz-like mechanical properties in its two long dimensions, due to its phyllosilicate structure. The individual clay grains have very large aspect ratios, which allows them to transfer the large loads they can withstand to the polymer matrix over a large contact area. However, they are normally limited to fairly low volume fractions in the filled system because their orientation is random, which limits the strength that can be achieved by the resulting composite material.

One researcher whose name I can't recall at the moment (though he was polite enough to answer my email, years ago) was able to deposit grains of bentonite all in parallel planes, bound together by something like limpet glue, with the result that the resulting composite material had a very high volume fraction of clay particles, and thus a very high strength and stiffness; he had to devise a new means for testing the stiffness by measuring small displacements under a microscope, using a bunch of glass spheres stuck to the film of material as the load was applied, then writing software to analyze the images to measure the displacement. But his procedure was extremely slow, depositing one layer of clay on the surface at a time.

I just thought of a process which might be faster, although it still deposits layer by layer.

The clay particles with appropriate surface treatment are suspended in a resin dissolved in a solvent. This is spin-coated onto a substrate, which orients the clay particles parallel to the surface of the thin layer, then heated to drive off the solvent, leaving a layer of solid resin and clay particles. The heating time needs to be long enough for the solvent to diffuse out from underneath clay particles. Another layer is applied on top of the first, and similarly dried, and the process is repeated many times. If the clay particles are 10% of the original dispersion, but the solvent is 85% of it, then after the solvent is removed, the clay particles will have a volume fraction of some 67%, which should be enough to have clay particles on each layer overlapping particles on the previous and next layers enough to form a continuous filler network.

Two possible improvements are relevant.

First, after evaporating each solvent layer, the new surface is washed before the next spin-coating step long enough to remove some of the resin just deposited. This will preferentially remove resin that is not underneath a clay particle, although if continued long

enough it will remove clay particles too. So there's a certain range of washing intensity within which this change will give a higher volume fraction of filler in the final product.

Second, although the resin used is still solid when the solvent evaporates, it is a photopolymerizable precursor to another resin. This permits a higher-strength final product by photopolymerizing the whole thing at the end, as well as 3-D printing by selective photopolymerization of each layer, which in that case would also need to include UV absorbers to keep the photopolymerization at the surface. See Powder-bed 3-D printing with a sacrificial binder (p. 506) for related 3-D printing processes.

Topics

- Materials (p. 1138) (59 notes)
- Manufacturing (p. 1151) (29 notes)
- Filled systems (p. 1161) (16 notes)
- Clay (p. 1179) (10 notes)
- Anisotropic fillers (p. 1218) (6 notes)

ECM for machining nonmetals?

Kragen Javier Sitaker, 02021-07-05 (updated 02021-07-27)
(11 minutes)

I've looked a bit into electrolytic machining (usually "ECM") of glass, but I think probably it's not a good idea; sandblasting ("abrasive jet machining") is a dramatically more reasonable idea for glass. But I was thinking about marble, and I think electrolytic machining would probably work really well for marble.

Electrolytic machining of marble and similar basic minerals

Suppose we use a neutral, very dilute NaCl electrolyte, saturating the pore spaces of the marble, with a high enough overvoltage that most of the electrolytic product is hydrogen and oxygen (1.23 V) rather than sodium and chlorine. At the anode, we produce (conventionally) H^+ ions and oxygen by stripping an electron from water: $2H_2O \rightarrow 4e^- + 4H^+ + O_2$. If the anode is close to the marble or in contact with it, these H^+ ions (really hydronium, H_3O^+) will immediately react with the $CaCO_3$ to reform water and carbon dioxide: $CaCO_3 + 2H^+ \rightarrow H_2O + Ca^{++} + CO_2$. Thus the marble around the anode will be eroded, neutralizing the acid and producing $CaCl_2$. If the anode is either silver-plated or made of carbon, I think it will not itself suffer erosion.

We can pump a fresh stream of electrolyte constantly through a hole in the center of the anode; if the anode is sheathed in an insulator, such as teflon, the hole in the insulator where the electrolyte squirts out can ultimately determine where the workpiece erosion happens, rather than depending on the geometry of the possibly-eroding anode itself.

I don't think much chlorine gas will be produced, if any, because the electrons being sucked out of the anode are coming from the abundant hydrogen ions, which replace them from the abundant marble, rather than from the scarce chlorine ions. If that does turn out to be a problem, alternative electrolytes exist.

Aside from lime and its carbonates, the same anodic-attack approach should work with other nonconductive minerals subject to easy acid attack, such as dolomite (Mg/Ca), siderite (Fe), smithsonite (Zn), magnesite (Mg), malachite (Cu), azurite (Cu), and perhaps portland cement (calcium silicate hydrate).

A waste product of strong alkali will form at the cathode, though perhaps this could be ameliorated with a suitable buffer, perhaps based on acetate, citrate, or borate, to compensate for the buffering the carbonate provides to the acid produced at the anode; lacking this, ultimately the liberated calcium ions will find their way to the cathode and precipitate slaked lime.

Choice of electrolyte

It's important for the anions in the electrolyte to maintain the calcium ions in solution, unlike phosphate (apatite), pyrophosphate,

oxalate (weddellite/whewellite), fluoride (fluorite), hydroxide (slaked lime), sulfate (gypsum), tartrate (beerstone, barely soluble) or titanate (perovskite). Chloride is a good choice, but other choices include bromide, iodide, cyanide, thiocyanate, nitrate, acetate (34.7 g/100mℓ), chromate (2.25 g/100mℓ), or formate (16 g/100mℓ at 0°).

Calcium borate is a weird boundary case. In dicalcium hexaborate, the least soluble borate of calcium, water can dissolve 202mg/100mℓ of boria, which works out to `(* 202 (/ (+ (* 2 40.078) (* 2 15.999) (* 6 10.81) (* 9 15.999)) (+ (* 6 10.81) (* 9 15.999))))` = 310 mg/100mℓ of the salt, though US Borax gives 470 mg/100mℓ.) Generally, borates are complicated and not very soluble, much like silicates, phosphates, and silicoaluminates, because of the possibility of oligomer or polymer formation.

Choice of anode material

It's simultaneously desirable to use anions that won't form soluble salts with the anode material itself, both so you don't end up with nasty anode salts all over your nice cut marble, and so you don't have to keep feeding in more anode as it's consumed (and suffering imprecision from anode wear uncertainty). A gold-plated anode would permit the use of just about any electrolyte (except maybe cyanides, which have other disadvantages), and even silver should resist chloride and the other halogens (except fluoride). Ordinary copper would permit thiocyanate, and lead might permit the use of iodide and bromide, though the resulting lead salts would be soluble enough to pose real risks of contamination. Because copper is lower in the reactivity series than hydrogen, you'd think it could avoid forming copper chloride in this use, but in fact copper plating using chloride or acetate baths is totally a thing. I have definitely anodically destroyed copper in salty vinegar.

(Of course, graphite or carborundum electrodes will withstand arbitrary acid or base attack at ordinary temperatures, and platinum electrodes withstand nearly any reactive environment.)

Here's a solubility chart formulated for the purpose:

(anion) Magnesium Calcium Gold Copper Lead Silver Tin Iron Nickel

fluoride sS I I sS sS S† S S S

chloride S S S† S S I S S S

bromide S S sS S sS I S S S

iodide S S I I sS I S S S

cyanide S S S I sS? I ??? ??? I

thiocyanate S? S? ??? I sS sS sS? S? S?

acetate S S sS† S S I sS? S S

chromate S S S?† I? I! I sS? R sS

formate S S ?† S S (16 mg/mℓ) ??? unstable S? S S

borate sS? sS ??? I† S? ??? † † †

sulfate S sS R† S sS sS S S S

citrate sS sS ??? sS S I (285 ppm) ??? S S

† indicates compounds that I don't think will form electrolytically from unoxidized metal and relevant anions.

(Ugh, I don't have zinc in the chart. But it's almost the same as magnesium. Also, I don't have tartaric, lactic, and phosphoric acids.)

I tried reducing the above solubility chart to an easier-to-use form a few times, but I never succeeded.

Alternative solvents

Another approach is to make the electrolyte from ions dissolved in a polar solvent other than water; for example, anhydrous ammonia, formamide, tetrahydrofuran, acetone, isopropanol, methyl ethyl ketone, pyridine, DMSO, dichloromethane, or deep eutectic systems; these will yield different solubilities for various ionic substances.

For example, only 0.2 grams of muriate of potassa dissolves in 100 mℓ of DMSO, and 0.013 g of potash, but it can dissolve 20 g of the iodide or 30 g of muriate of Mars, while the muriate of lime is entirely insoluble. In DMSO, hydrated cupric acetate and muriate are insoluble, but the acetate and muriate of zinc are quite soluble, as are the muriates of tin, so a copper anode with a zinc-muriate electrode dissolved in DMSO might be able to electrolytically etch salts of tin or iron with impunity.

Some solvents may not produce electrolysis products of their own that are useful for the electrolytic etching process, the way water does; for example, the anodic reaction converting carbonate ions to oxygen and carbon breaks apart and then reforms water molecules in the process. DMSO in particular seems likely to produce electrolysis products much more noxious for human life.

Etching with a cathode instead of an anode

If the electrolytic cell's cathode rather than its anode were the active tool, it should work for acidic or amphoteric materials attacked by strong bases, most notably sapphire (slowly, at tens of megapascals and 400° or higher), gibbsite, and amphoteric oxides like those of zinc (philosopher's wool, a refractory (1974°) thermochromic transparent piezoelectric direct-bandgap n-type semiconductor with a 3.37-eV bandgap), titanium (rutile, a UV-blocking strongly birefringent photo-superhydrophilic photocatalytic transparent refractory (1843°) n-type semiconductor with refractive index 2.61 and a 3.05 eV bandgap which becomes an excellent dielectric when stoichiometric), tungsten (an electrochromic photocatalytic semiconductor), vanadium (a refractory (1967°) transparent semiconductor with an 0.7-eV bandgap that becomes metallic and IR-reflective in 100 fs above 68°, a temperature that can be adjusted with tungsten doping), and tin (cassiterite, a refractory (1630°) n-type semiconductor with a refractive index of 2.0 and a specific gravity of 7).

The amphoteric oxides can be etched just as well by the anode-acid process described at the start, but etching them with a cathode means you can use any metal for the tool electrode, since it won't be vulnerable to anodic dissolution.

Metal sulfides might be another candidate. Leaching with very dilute alkali has been successfully used to separate antimony from stibnite (antimony sulfide) without affecting other metals, with etching speeds around 10 microns per minute. Alkaline leaching has also been used to extract lead, tungsten, zinc, vanadium, and

chromium from various ores. Mostly, though, these processes are very slow.

These cathodic etching processes, instead of producing waste alkali at the cathode, would produce waste acid at the anode, and the same comments about buffering apply to avoiding undesired acid accumulations.

More speculative directions to explore

A very interesting question for this kind of electrolytic work: these semiconductors, zinc oxide, rutile, tungsten oxide, vanadia, and cassiterite, are all immune to anodic dissolution; but they are amphoteric enough to be unstable for this kind of work. But perhaps other nonmetallic semiconductors other than carbon and carborundum, such as GaN or InP, may be alternative electrode materials.

Pulses of high voltage on small-diameter electrodes should be able to produce plasma discharges to overcome activation barriers, a sort of corona-discharge EDM/ECM hybrid, though this would surely also erode the tool electrodes.

Topics

- Materials (p. 1138) (59 notes)
- Manufacturing (p. 1151) (29 notes)
- Electrolysis (p. 1158) (18 notes)
- Machining (p. 1165) (13 notes)
- ECM (p. 1186) (9 notes)
- Solubility (p. 1273) (3 notes)
- Sandblasting (p. 1332) (2 notes)

Sonic screwdriver resonance

Kragen Javier Sitaker, 02021-07-06 (updated 02021-12-30)
(11 minutes)

I'd previously made some notes somewhere on using mechanical vibration to counter stiction when driving or removing screws, thus preventing screwdriver camout; this would be less annoying if it was above the threshold of human hearing, although that might not be practical, because the energy of each vibrational pulse needs to exceed the stiction energy. It occurs to me that it would also be useful for driving nails, where stiction is pretty much the name of the game.

The ideal graph of force against time of such a sonic-screwdriver system would be the derivative of a sawtooth: a Dirac delta in the desired direction large enough to make progress on the fastener, which necessarily creates momentum within the sonic screwdriver in the opposite direction, followed by a recovery time during which a constant small force is applied to arrest and eventually reverse that momentum, all the while maintaining contact with the fastener. The recovery force is applied against, say, the hand of the person using the device.

A control system can pause if contact is lost, and detect whether progress is being made and increase the impact energy (lengthening the recovery time and lowering the frequency) if not, or decreasing the energy if so, moving the frequency into the ultrasonic; and some degree of randomness in the interval would seem to be desirable to spread acoustic annoyance across the spectrum, also avoiding the danger of dumping a damaging amount of energy into a single resonance mode of the system.

(For screw installation, it would be very useful to be able to set a torque.)

This is all very similar, actually, to the action a hammer drill or rotary hammer uses for drilling holes in masonry. Bosch's 5.7 kg 1500-watt "Titan 5 kg SDS Plus" hammer drill advertises 5 J maximum impact energy and is rated for drilling 32-mm holes in masonry, and other hammer drills from the same "Farmers Weekly" seller advertise impact energy in the 1.7 to 5 J range. Makita advertises their "23 lb. AVTDemolition Hammer" HM1213C rotary hammer as "delivering 18.8 ft.lbs. of impact energy, which is 25 J; it runs on 14 amps, which is presumably 1.7 kW. One shill site advertises the DeWalt DCH773Y2 rotary hammer (US$1099, 21 pounds) at 19.4 J of impact 1000–2000 times a second and marvels at how it "just pulverizes concrete".

For no particularly good reason, there doesn't seem to be a tool that applies the same sawtooth force profile to pulling an actual flat saw blade or wire saw through stone or concrete.

Some of these hammer drills and rotary hammers have a new anti-vibration feature which creates *two* simultaneous impacts with each hammer blow: one on the tool bit to make the hole, and the other on an internal counterweight, so that what rebounds from the impact is only the counterweight rather than the tool you're

struggling to hold in your hands. A different way to accomplish the same thing would be to just impact the counterweight directly against the toolbit: apply the tool's motor to gradually accelerating the counterweight away from the toolbit against the counterweight's spring, then toward the toolbit once the spring wins and the counterweight starts accelerating it back toward the toolbit. Either version of this same feature can be applied to the helical motion of an impact screwdriver (like those I'm discussing here) just as well as to the linear motion of a rotary hammer.

How hard is it to drive screws? Well, how hard can I twist a screwdriver? In a simple test just now with a water bottle and a metal pipe, I was able to rotate my wrist in a T-handle sort of configuration hard enough to lift 5.5 kg with a lever arm of 330 mm (the water bottle) and 430 g at 500 mm (the metal pipe itself), which is a torque of about 20 N m. This is probably about as hard as I can twist a T-handle screwdriver to unscrew a screw, but smaller amounts of torque usually suffice. About a tenth of a turn is pretty much always enough for a screw to make progress rather than springing back to its old position, and that would work out to about 13 J at that torque. (This explains the quasi-unit-compatibility of torque and energy: 20 N m of torque is really 20 N m of energy per radian!)

Normally, of course, screws don't have to turn nearly that far to not spring back, and I don't need a T-handle screwdriver to remove them.

Engineering Toolbox gives withdrawal forces for some nails in spruce ranging from 17.6 pounds to 348 pounds (79–1550 N), with a common 16-penny nail of length 3½" (89 mm) requiring 141 pounds (630 N). Driving such a nail 1 mm would probably be far enough that it wouldn't spring back out, and that would be 0.6 J. Framing carpenters drive them all the way in in three hammer blows, which is about 20 J per blow, disregarding the mass of the nail and the vibrations of the structure, as you should.

So a simple sonic screwdriver would probably need to be able to ramp up to on the order of 1–10 J per impact to make progress in difficult cases, and then it would be able not only to remove screws and drive framing nails, but also drill and saw concrete, wood, and stone, and engrave or center-punch steel. But most of the time 0.1 J would be enough. If it were running at 1000 W, the average impulse frequency would be in the range 100 Hz to 10 kHz, unfortunately all well within the audible range. 1000 W would also drive one of those 60 J nails in 60 ms.

You could power it off Li-ion batteries in the now-conventional way, but it really only needs to contain on the order of 128–256 J at any given time, so it might often be more convenient to charge it with the necessary energy just before use. Even a clockwork spring sort of arrangement might be adequate with a pullstring; I can do two pushups to about 600 mm, lifting half my 110-kg weight, which is about 320 J. So normal people should be able to repeatedly pull a pullstring out to about a meter under a tension of about 50 N, like starting a lawnmower. Easy jobs might need a single pull occasionally, while hard jobs might need five or six pulls before every fastener or whatever, although at some point you just want to plug

the thing in.

The pullstring need not be visible in normal operation; you could split the tool into two parts joined by the pullstring when charging it, then reassemble them once charged.

Clockwork springs have the advantage of having almost arbitrarily high power density, unlike batteries, both for charging and for discharging. Rechargeable lithium batteries typically have a "fast charge rate" of "1 C", meaning 1/1 hour, or less, perhaps 0.5 C, meaning 1/.5 = 2 hours. If you were going to charge them up with a pullstring, you would have to pull the string continuously over the course of that hour or two.

The modulus of resilience of a material is the amount of energy it can store per unit volume as elastic deformation. For tensile elastic deformation of ductile linear materials, the relevant figures are Young's modulus E and the yield stress σ_y, and the integral of deformation from 0 to the yield strain σ_y/E gives us $½\sigma_y^2/E$. Most types of steel have the same Young's modulus regardless of their hardness, about 200 GPa. Soft steels have yield stresses as low as 300 MPa, but normally we make springs from music wire, which is more like 2800 MPa. This gives us a *tensile* modulus of resilience for music wire of some 20 MJ/m³, or 20 J/cc. 256 J then would require 13 cc of spring steel, or 100 g.

Conventional clock mainsprings do in fact deform in tension and compression, but the mainspring is only stressed to its limit at its inner and outer surfaces; on its neutral axis it isn't strained at all. So the situation is actually even worse: you only get ¼ of the possible tensile energy storage that way, and you'd need 400 g of mainspring. This is getting to be a rather heavy screwdriver!

Coil springs instead deform the spring material in torsion, which is to say, in shear, and much more of the material is closer to the maximum shear strain. For *shear* deformation we're interested in the shear modulus G, about 79 GPa for steels, and the yield shear strength τ_y, which for steels is about [0.58 of σ_y][10], or say 1600 MPa; the factor 0.58 comes from $3^{-½}$. So if we just calculate the shear modulus of resilience as $½\tau_y/G$, which I'm not sure is the right thing to do, we get 16 MJ/m³ or 16 J/cc, about 15% lower than the tensile modulus. I guess I should do the integral to see how much the distribution of the shear strain in the circular coil spring cross-section affects the situation, but it seems clear that using shear rather than tension (and compression) doesn't make a huge difference.

By twisting a *tube* rather than a solid bar, the way a lot of torsion bars in car suspensions do nowadays, you can get the full shear modulus of resilience of your metal, but that doesn't get you more energy per volume, just more per mass.

So what about carbon fiber? I hear truck suspensions nowadays are starting to use carbon-fiber-reinforced plastic rather than steel.

sawtooth components

Topics

- Contrivances (p. 1143) (45 notes)
- Pulsed machinery (p. 1167) (12 notes)
- Hand tools (p. 1197) (7 notes)
- Sonic screwdrivers (p. 1324) (2 notes)

Subnanosecond thermochromic light modulation for real-time holography and displays

Kragen Javier Sitaker, 02021-07-06 (updated 02021-12-30)
(8 minutes)

I learned last night that vanadium oxide transitions from IR-transparent to highly IR-reflective at 68° (or a lower temperature if doped with tungsten) in 100 picoseconds. If you wanted to project an IR image onto something, say for an IR camera, what kind of image quality could you manage with that?

Well, suppose you have a line of 64 vanadium oxide pixels illuminated by an IR laser, and you can individually turn them on and off using individual heating elements, and you've arranged the time constant of cooling to be on the order of the thermochromic response time, so maybe you can turn a pixel on or off in 200 picoseconds. This allows you to modulate the laser beam through each such pixel at 5 gigabaud, 5 gigapixels per second, up to 2.5 GHz; the overall system is 320 gigapixels per second. Each pulse of light in the air is 60 mm long.

If you shine light through your modulator strip onto a 30krpm spinning mirror whose axis is more or less parallel to the strip --- say they're both vertical --- then the mirror spins by 630 nanoradians (0.13 arcseconds) per pixel column. If you can manage such a tiny beam divergence, then at 10 meters' projection distance, your pixel columns are 6.3 *microns* wide.

If your beam waist is 200 mm and your wavelength is 0.1 mm (near IR), the usual Airy-disk formula approximate formula $\sin\theta \approx 1.22\ \lambda/d$ gives us 1.22(.1/200) = 0.61 milliradians, three orders of magnitude blurrier. And a 200-mm-wide mirror facet spinning at 30krpm would be something to behold --- ideally from a great distance.

So trying to use such a spatial light modulator in such a way would probably greatly exceed the capabilities of your electronics, your optics, and your mechanics. But it's something to keep in mind if those each improve by an order of magnitude or so.

For LIDAR, you could imagine sending out periodic 200-picosecond pulses by modulating such a mirror to briefly pass light (perhaps in a direction determined by mechanically moving conventional optics) and then modulating a second mirror to briefly pass light some time later; by using analog electronics to vary the phase delay between the two pulses, you should be able to measure the time delay of the reflection to within about 20 ps (6 mm).

I don't know if such fast thermochromic materials exist in visible or shorter light wavelengths, which would be convenient for super-high-speed displays. If you wanted 4000 horizontal pixels spread across half a radian with a 30krpm spinning mirror, 40 nanoseconds would be plenty fast enough, and there are lots of LEDs that can do that.

Real-time holography would be another potentially interesting use. In particular, if you illuminated a grid of such thermochromic pixels with a single laser, you could use it as an infrared phased-array communications transmitter, modulating an arbitrarily large number of separate beams, each at 2.5 GHz. (The divergence of each beam, and thus the number of beams that you could actually separate in practice, would depend on the number of pixels and on the total aperture and thus diffraction-limited divergence.) The pattern imposed on the thermochromic pixels would merely be an approximate linear sum of waveplates, one waveplate pattern for each outgoing beam.

Such a system can be used for multicast communication in an even faster mode. Suppose your laser source can be modulated at 20 GHz, 25 ps per bit. By setting the phased array to direct its light to five specific destinations, after a wait of 200 ps (8 bit times), you can then talk to those five destinations at the full 20 GHz bandwidth.

A perhaps more interesting use of the phased-array approach for optical communication is reception, in which you simply reverse the flow of time; incoming light from a particular chosen source is focused onto your photodiode (or other detector) with a waveplate on the phased-array modulator. You can superimpose several waveplates at once to enable reception from any of several possible sources (each with high "antenna gain", though necessarily lower than you'd have for listening to a single source). Similarly you can have several photodetectors, perhaps at different focal lengths behind the waveplate; their holographic beamforming patterns on the shared phased-array spatial light modulator will add noise to one another, but only mildly so.

Such bidirectional phased-array optics with small arrays of high-speed photodetectors can also be used as "lensless" cameras, whether by physically scanning them over a scene or by changing the holographic beamforming pattern to scan. And of course these also work in reverse, illuminating scenes from a distance.

Alternative means of rapid light modulation, other than thermochromic effects, include Kerr (10 GHz, 30 kV) and Pockels (slower, 10 kV) electro-optic cells. I think these can be used not only to set the phase delay (and polarization rotation) through a material, but also to turn off and on total internal reflection as fast as you can modulate the electro-optic cell. If so, near the critical refractive index that is the threshold for total internal reflection threshold, the transmitted wavefront is still potentially planar (it depends on the phase delay of the electric field over the cell) but has a very large derivative of refraction angle with respect to the applied field, and, thus, potentially with respect to time. If this works, it is a faster alternative to a spinning mirror for scanning a light beam, and perhaps a better alternative to a phased-array transmitter as well.

Liquid crystal pixel arrays, of course, if stripped of their conventional second polarizer, can also produce a spatially modulated optical phase delay, and can thus also be used for such holographic beamforming, especially if the pixel size is not too much larger than the light wavelength.

If the pixel size of any of these spatial light modulators cannot be

kept small, blacking out all but a small window in the center of each pixel may help. A microlens array between the SLM and the focal plane should be able to reduce the resulting loss of gain, for example by focusing all the light that arrives from the laser source onto the small window, or directing all the light that makes it through the small window toward the single photodetector. A microlens array between the SLM and the rest of the world would limit the device's field of view but add "antenna gain".

Whether using liquid crystals, direct electro-optic effects, or thermochromic effects, if great speed in changing the image is not required (because the speed is taken care of by the light source or detector rather than the modulator, which only sends it in an occasionally-varying direction), it may make sense to use a "passive matrix" or "active matrix" like a common LCD display to multiplex a large number of pixels onto a smaller number of control lines.

Topics

- Contrivances (p. 1143) (45 notes)
- Frrickin' lasers! (p. 1168) (12 notes)
- Optics (p. 1209) (6 notes)
- Displays (p. 1261) (4 notes)
- Communication (p. 1264) (4 notes)
- Spatial light modulators (SLMs) (p. 1327) (2 notes)
- LiDAR (p. 1355) (2 notes)
- Vanadia
- Holography

Notes on Richards et al.'s nascent catalytic ROS water treatment process

Kragen Javier Sitaker, 02021-07-07 (updated 02021-07-27) (14 minutes)

Posted at https://news.ycombinator.com/item?id=27763602

(I should preface this by saying that, unlike you, I have no real water disinfection expertise, so I may be overlooking significant fundamentals in what follows.)

Those costs sound reasonable for niche uses (though I do think they would prevent it from "revolutionizing water disinfection technologies" as they claim in the Conclusions), and being able to operate in practical terms at a very small scale seems to me like an advantage rather than a disadvantage. It clearly is more expensive than conventional water treatment and would have difficulty scaling to replace it for reasons of material availability. Other alternative very-small-scale disinfection approaches are probably cheaper than their process in its current form; they mention ozonation, UV irradiation with germicidal lamps, photocatalytic disinfection, and the Fenton process in the paper and supplement.

I don't see how the square-cube law plays into it; the reason they're supporting the catalyst on rutile (anatase?) instead of just using solid bars of metal alloy is precisely to ensure neat linear scaling, and the reason it's 1% metal instead of 0.1% or 0.001% is that the rutile is in the form of 1–100 nm particles. (See p. 8/12 in the article and Fig. 4 on p. 7/12). The only square-cube thing that occurs to me is that a large ice bath requires much less ice input than a small ice bath, but that favors scaling the process up, not down.

A couple of other points:

⓪ Your calculations seem to be correct.

① People's drinking water needs are lower than you suggest by a factor of about 65, although conventional surface-water treatment plants cannot take advantage of this.

② You're not taking into account the supply-chain issues that plague smaller-scale treatment facilities.

③ The cost of consumables that this process would eliminate would still be lower than the cost of the catalyst.

④ There are plausible ways the process might be improved that could make it economic.

Thanks to mkr-hn I found the paper at https://www.researchgate.net/publication/352882305_A_residue-free_approach_to_water_disinfection_using_catalytic_in_situ_generation_of_reactive_oxygen_species; the canonical paper link (unforgivably missing from the original press release) is https://www.nature.com/articles/s41929-021-00642-w, and the supplementary material is at

https://static-content.springer.com/esm/art%3A10.1038%2Fs41929-00021-00642-w/MediaObjects/41929_2021_642_MOESM1_ESM.pdf.

— ⁂ —

⓪ Scaling calculations.

Here are the calculations in more detail, since I misread yours badly at least twice.

You say 0.5 MGD is 6.5 million times higher than the 0.2 mℓ/min (3.3 μℓ/s) in the experiment. 0.5 MGD is 21.9 ℓ/s, which is 6.6 million times higher than 3.3 μℓ/s.

I think you are reading it correctly; they do say their catalyst is 0.5% Au and 0.5% Pd on a TiO_2 support: 0.6 mg of gold and 0.6 mg of palladium to process 0.2 mℓ/min of water. The supplementary material has a plot of different catalyst mixes they tried (Sup. Fig. 14, p. 12, 13/32).

If we normalize that amount of catalyst metal to SI units, that's 180 gram seconds per liter (g·s/ℓ) each of gold and palladium. Your 0.5 MGD (22 ℓ/s) for 5000 people is 4.4 mℓ/s per person, which works out to 790 mg per person each of gold and palladium. Palladium at US$2800 per troy ounce (https://www.kitco.com/charts/livepalladium.html — that *is* per troy ounce, isn't it?) is US$90/g. Gold at US$1800 per troy ounce is US$58/g. Multiplying it out, that's US$71 of palladium per person and US$46 of gold per person, for a total of US$117 of catalyst metals per person. This seems likely to be the dominant cost of the whole shebang at anything larger than laboratory scale; the actual preparation of the catalyst with these metals by the process they reported in the paper might be even more expensive, but presumably cheaper methods are possible.

The 5000-person 0.5 mgd plant you cite as an example would need 3.9 kg of gold (US$230k) and of palladium (US$350k) for its catalysts, a total of US$580k, which is the number you gave for "our small plant".

790 mg of gold per person, times 7.7 billion people, would be about 6100 tonnes, about 3% of all the 200 000 tonnes of gold that has been mined so far. However, the corresponding 6100 tonnes of palladium would vastly exceed the above-ground stocks of palladium, estimated at 4.5 "moz", which I think means "million troy ounces", or 140 tonnes.

However, they also tried 1% gold without any palladium, and although this produced lower H_2O_2 concentrations, they were still high enough to be somewhat effective (≈80 ppm rather than 220 ppm, resulting in a 1.6 $\log_{10}$ reduction rather than the 8.1 they were so satisfied with). So in the face of resource limits you could trade off a larger amount of gold and a longer residence time against scarce palladium.

① Potable water needs are 5.7 ℓ/day/person, not 380.

Your figure of 4.4 mℓ/s/person is 380 liters per day per person (100 gallons per day per person), but Burning Man recommends *1.5* gallons per day per person (5.7 liters/day/person, 0.066 mℓ/s/person), which includes water for showers, for cooking, and for drinking in a very dry environment with extensive physical exertion, though not for

bidets. My experience at Burning Man is that you can get by on less.

That's 67 times less water than your figure.

Maybe your waterworks is supplying not only drinking water but also toilet-flushing water and lawn-irrigating water? Those don't normally need antibacterial treatment. Even the 0.2 mℓ/min benchtop catalyst they used in the paper (containing 5.4¢ of palladium and 3.5¢ of gold) would supply 0.29 liters per day; it would only need to be scaled up by a factor of 20 (to 12 mg gold (US$0.69) 12 mg palladium (US$1.10), 2.4 g rutile) to supply the requisite 5.7 liters per day per person.

② Supply-chain issues with consumables.

This is mentioned in the Conclusions section of the paper.

Conventional surface-water treatment facilities involve treatment with hypochlorite, permanganate, chloramine, and whatnot. These pose some safety concerns (especially at small scales) and in many places are subject to legal reporting requirements. Occasionally there are industrial accidents because a truck driver pumped ammonia into the hypochlorite tank or vice versa; when people try to use the same materials at the household scale, sometimes you get medical problems because somebody put 1000 ppm of hypochlorite into their drinking water instead of 3 ppm. And, especially for individual households in middle-income and poor countries, there are often supply-chain issues with these materials; sometimes shipments don't arrive, or the household doesn't have enough money to buy a new bottle of bleach when they run out, or the products are falsely labeled, or have degraded before delivery—a particular problem with sodium hypochlorite.

For example, the cleaning-products store down the street from me here in Argentina advertises that they sell "100% chlorine", which I found pretty alarming until I saw that they are unpressurized bottles of liquid, not pressurized gas cylinders. Probably it's mostly aqueous sodium hypochlorite, but who the hell knows what the concentration is, and what else they put in it? The supermarket has jugs of sodium hypochlorite solution stabilized with sodium hydroxide, and the label tells you the nominal concentration (usually 66 g Cl/ℓ) but of course the bottles aren't hermetically sealed, may be exposed to sunlight, and may not have been properly quality-controlled at the factory in the first place.

If you're running an 0.5 mgd water treatment plant, you can presumably just *measure* the concentration in your hypochlorite tank, and have someone assigned to do this. And since they replaced the water main last year, we finally do have a reliable water supply 24/7, instead of only at night when the neighbors aren't pumping so much water. Still, the water from the mains has to be pumped up to the rooftop tank, where the chlorine concentration falls before we use it; maybe the water plant is putting way too much chlorine in the water to compensate for that, because it smells pretty strongly of bleach when it comes out of the tap.

If, by contrast, you have a durable catalyst that fulfills the same microbicidal function without needing constant reliable shipments of bleach, all of these concerns go away. It's like the difference between photovoltaic panels and the electric grid: even if a reliable electric

grid might give you energy at a lower cost than having your own solar panels, that doesn't help if you don't have a reliable electric grid. It might be worth spending US$10–US$200 per person for an autonomous germicidal appliance; it might be cheaper than a refrigerator or washing machine.

③ Consumables costs.

That brings us to the question of consumables costs. Obviously enough, I've never operated a waterworks, but for example Callie Sue Rogers's 02008 B.S. thesis on conventional surface-water treatment plants https://core.ac.uk/download/pdf/4276743.pdf has a table surveying a number of Texas "conventional surface-water treatment facilities", concluding that they spend between US$20.21 and US$286.14 per million gallons on "chemical costs", depending on the condition of the water they're starting with; I infer "chemical costs" means things like the oxidants mentioned above, flocculants, and precipitants; this works out to 5.3 to 76 microdollars per liter. She estimates that the total production cost ranges from US$0.31/1000 gallons for small 5 mgd plants down to US$0.13/1000 gallons for larger 130 mgd plants (respectively 81 and 34 microdollars per liter).

The US$120/person figure (US$120 per 100 gallons per day) would be 10400 microdollars per liter if the catalyst only lasted a month, 870 microdollars per liter if it lasts a year, 170 microdollars per liter if it lasts 5 years, or 43 microdollars per liter if it lasts 20 years. It seems almost certain to exceed the cost of buying the necessary oxidants on the open market, particularly on a time-discounted basis.

How long would the catalyst last in practice? The catalyst materials in question are pretty darn inert, so you could probably clean them (with acids and/or alkali) if they get poisoned by some kind of inorganic contaminants in your incoming water. Organic fouling won't be a concern, and organic catalyst poisons would just get chewed up by the H_2O_2. You have to use cleaning agents that aren't so aggressive that they can corrode the porous rutile support, but I think that requires something like hot concentrated sulfuric acid.

Of course, such cleaning might cut into the supply-chain-autonomy advantage a bit. But it might not be necessary at all; this isn't a car catalytic converter, after all, and it's constantly washed with fresh water.

What about the energy cost? Maintaining a 2° ice bath is pretty cheap, but 10 bars (1 MPa, 145 psi) of pressure doesn't come for free. It costs 1kJ/ℓ (a simple unit conversion). Your 380 liters a day per person would be 4.4 watts. At US$20/MWh this is US$0.77 per year of energy per person, probably in practical terms more like US$3 per year once you take into account the inefficiencies of electric motors and pumps. This is small but not insignificant compared to the cost of the catalyst. But it's still a low enough energy cost that you could hand-crank the pump.

④ Process improvements.

So this is an economically feasible way to provide the 5.7 liters per day of potable water a person needs, even if existing alternative processes are cheaper. What are the prospects for improving the process further?

The key findings of this paper, as I read it, are that this catalytic process is resilient to common solutes, and that the witches' brew of reactive oxygen species produced in this process is more effective than commercially purchased H_2O_2—they say by a factor of 10^7, but in Supplementary Table 2 (p. 18, 19/32) I see $\log_{10}$ reduction of CFU/mℓ going from 0.44 to 0.98 (3.5×), from 0.64 to 1.25 (4×), from 0.96 to 1.18 (1.7×), and from 0.84 to 1.48 (4.4×), so I have no idea where 10^7 comes from.

Process intensification is one possibility for making it economic; it's quite plausible that the use of higher pressures or additional alloying elements in the catalyst could increase reaction rates by an order of magnitude or more, and it's possible that heating the water after catalytic ROS production (by running it through a countercurrent heat exchanger into a hot tank) would enable even lower concentrations of ROS to disinfect effectively. (Normally you'd also consider heating the catalyst, too, but presumably the ice bath is necessary to push the equilibrium toward high concentrations of H_2O_2 and other ROS.) Applying light or a voltage to the catalyst are other possible routes to increased free radical production.

Another possibility is lowering the cost of the catalyst; metal-oxide, metal-(other-)chalcogenide, intermetallic, and even transition-metal catalyst systems might work adequately through the same route, and could be much cheaper even if the catalyst leaches at an appreciable rate.

After all, what use is a newborn baby?

Topics

- Materials (p. 1138) (59 notes)
- Pricing (p. 1147) (35 notes)
- Life support (p. 1251) (4 notes)

- Peroxide
- Drinking water

Memory view

Kragen Javier Sitaker, 02021-07-09 (updated 02021-12-30)
(6 minutes)

I was thinking that, for examining data in PDF files, it would be nice to have Python objects that basically represent chunks of bytes; for example, the contents of a file, some substring of the contents of a file, or the output of decompressing them. You ought to be able to slice these objects to create new ones on subranges of them, without copying the underlying data. And these objects ought to have default displays that are useful for debugging, but not only one such display, like the ordinary repr(); rather, several:

- A default text display that represents, in UTF-8 text, the contents of the chunk of bytes, treated as UTF-8 text, up to some limited number of lines, like 8. Like, the first four lines, an ellipsis, and the last three lines. Plus magic symbols to indicate things like control characters, malformed UTF-8, and trailing whitespace.
- A hex dump display, again by default limited to a small number of lines.
- An annotated display that also indicates their provenance.
- Jupyter displays that somehow use Jupyter facilities to display them more richly. For example, Jupyter apparently invokes `_repr_svg_` if present (which the `graphviz.Graph` class uses). Or you can return an `IPython.display.Image(filename)` or `IPython.display.Image(data=somebytes, format='png')`.

Also, I want to have parse-node objects, which are the same kind of objects but with child nodes, trailing context, and computed values, and the computed values are their default display --- but the other forms of presentation are still available. To get one of these, you invoke a .parse() method on one of the raw buffer objects with a grammar argument, and you get back a version of the same object, but with the parse-node stuff associated.

Even the computed values ought to be able to have multiple kinds of displays.

Ideally I'd like to enable clicking around the graph in an object-inspector sort of way, and clicking on Jupyter output to select different views and follow links to child nodes.

IPython facilities

Aha, and I see IPython also supports customizing tab-completion by defining a `__dir__` method, which is also what the builtin dir() uses, and for tab-completion of mapping keys or sequence indices foo[key], there's an `_ipython_key_completions_` method.

For customizing display, there's not only `_repr_svg_` but also png and jpeg representations, and in the HTML notebook, html, javascript, markdown, and latex. If you define more than one of these I don't know how IPython determines which one to use, but there is a `_repr_mimebundle_` that supersedes them, and an `_ipython_display_` function which I guess can call whatever IPython methods it wants to draw

stuff.

For interactivity, there's IPywidgets, which is simple enough to use for simple cases (sliders, dropdown selections) and will use the above means for output. For example:

```
from ipywidgets import interact
class Circle:
    def __init__(self, r):
        self.r = r

    def _repr_svg_(self):

        return """<svg width="256"><circle cx="128" cy="128" r="%s" stroke="#339"○
○ fill="#c6c" /></svg>""" % self.r

@interact(r=(0, 181))
def circle(r):
    return Circle(r)
```

It's no ObservableHQ, but hey, it gives you a form-based GUI to any function. There's an `interactive` which creates the widget that `interact` displays, but lets you maybe display it later with IPython.core.display.display, which is imported into notebooks by default. The dropdown facility is enough to select among different output facets; this very minimal prototype works, for example:

```
from ipywidgets import interactive

class DataView:
    def __init__(self, data):
        self.data = data

    def widget(self):
        def show(format='text', size=64):
            if format == 'text':
                print(self.data[:size])
            elif format == 'hex':
                print(' '.join("%02x" % ord(c) for c in self.data[:size]))


        return interactive(show, format=['text', 'hex'], size=(0, len(self.data))○
○)

DataView(open('README.md').read()).widget()
```

There's a submit-button version of `interact` called `interact_manual`, and there's widgets.Button which has an on_click method. Setting .value on an existing widgets.Text from another cell causes it to update its value on the screen. There are HBox and VBox widgets for layout (not sure if their .children is mutable), and various kinds of output widgets: widgets.Label, widgets.HTML, widgets.HTMLMath, widgets.Image (whose .value is the binary data of the image file, and also takes format, width, and height parameters). And there's widgets.Tab and widgets.Accordion for selectively hiding things, but

not, I think, for computing them lazily.

I'm able to spawn widgets with a button `on_click`.

Also, suitable for nesting inspectors, there's a widgets.Output to display anything IPython can display, which can be used as a context manager:

```
out = widgets.Output(layout={'border': '1px solid black'})
with out:
    display(YouTubeVideo('eWzY2nGfkXk'))
```

widgets.Output also has a `.clear_output()` method and (maybe in newer versions?) an `.append_display_data()` method which avoids problems with multithreading. However, `display()` with a widget doesn't get nested into the output as it should, so I guess my best bet is mutating a VBox; the following code is a bit awkward but does work:

```
def nestable():
    i = 0
    out = widgets.Output()
    inc = widgets.Button(description='inc')
    def inc_click(ev):
        nonlocal i
        i += 1
        with out:
            display(i)

    inc.on_click(inc_click)

    spawn = widgets.Button(description='spawn')
    vbox = widgets.VBox([widgets.HBox([inc, spawn]), out])
    def spawn_click(ev):
        nonlocal out
        out = widgets.Output()
        vbox.children += nestable(), out

    spawn.on_click(spawn_click)

    return widgets.HBox([widgets.Label('-'), vbox])

nestable()
```

Finally, non-button widgets have an `.observe()` method.

It's possible to insert things with a `_repr_svg_` for instance into these Output widgets, using `display()`.

Topics

- Human-computer interaction (p. 1156) (22 notes)
- Python (p. 1166) (12 notes)
- The Portable Document Format (PDF) (p. 1227) (5 notes)
- Parsing (p. 1228) (5 notes)

- Domain-specific languages (DSLs) (p. 1260) (4 notes)
- Jupyter

Spreadtool

Kragen Javier Sitaker, 02021-07-10 (updated 02021-12-30)
(30 minutes)

Watching an Abom79 video I saw an interesting way of fastening parts together with friction: one part has a slot in it that's too small for the other part, but round holes on both sides of the slot. By using a simple hand tool with two round pins that fit in the holes, one of which is mounted on an eccentric with a handle on it, you can exert a very large force to elastically bend the slot open, allowing the other part (in this case, a ceramic cutting insert for a lathe parting blade) to freely move in and out of the slot.

This is a fascinating idea to me. Bolts and other screws also fasten parts together with friction, but they add weight and tend to vibrate free if they aren't secured with lockwire. (Loctite isn't considered adequate in aviation.) Screws are inherently very three-dimensional, while this bending-open-jaws approach can I think be arbitrarily close to planar; you can fabricate it with 2-D cutting approaches. And screws take a long time to insert and remove, while this eccentric approach is a quarter-turn of a handle, like a camlock.

Screws get a pretty good mechanical advantage, but eccentrics can in theory have arbitrarily high mechanical advantage, because the relevant output lever arm is at most the eccentric axis displacement distance and diminishes toward zero like a toggle mechanism when the eccentric approaches top dead center; additionally, you could drive the eccentric rotation through a separate toggle mechanism if need be.

I'd previously seen the MTM Snap mill by Jonathan Ward and Nadya Peek, which uses snaplock connectors milled out of HDPE instead of screws, because Neal Gershenfeld loves snaplock connectors. And they're sort of the same thing as the flexing-jaw connector in the video, or anyway there's a continuum between them.

Exploring the design possibilities, I think this is a potentially very exciting mechanism offering many possibilities:

- Snap-jaw joints potentially offer a repeatedly-assemblable alternative to screw fasteners with several advantages:

- It could be integrated into the parts to be assembled at the cost of a few cuts and holes, rather than being manufactured separately.
- It could require dramatically less effort, whether measured in maximum force, in energy, or in time, to assemble and disassemble than screws do.
- It is usually enormously more vibration-resistant than screws are.
- On one of the two parts being connected, it's possible to make the connection anywhere in a continuous region, as with a screw slot, but without the reduction in holding force and part strength that a screw slot implies.
- The cost may be comparable, but is probably lower, by at least a factor of 2 or 3, in part because snap-jaw joints can easily be made

with XY cutting processes.
• You could build a reusable, very inexpensive, very rapid construction kit using high-strength snap-jaw fasteners, which would allow you to rapidly assemble and disassemble a variety of shapes.

A basic snap-jaw connection

Consider a mating pair of such parts, made of mild steel with 250 MPa yield strength and a 200 GPa Young's modulus, giving 0.125% yield strain. They're made from 3 mm steel plate (23.7 kg/m²); the first is 2-D cut with a 20-mm-long slot in its edge that tapers from 2.5 mm wide at the edge up to 3 mm at the base of the slot. The second has a corresponding backwards taper ground or rolled into its edge, so that its edge is the full 3 mm thick, while 20 mm from the edge, it's been thinned down to 2.6 mm. The slot must thus be elastically opened by 0.5 mm or more, 0.25 mm per jaw, to assemble the parts. Then these jaws can be clamped onto the edge anywhere along its length, not just at predefined locations.

Although there is a stop at the base of the slot, it consists of a tongue; the jaws do not join onto the main body of the second piece for another 20 mm, so the total bend in each jaw is 0.25 mm over 40 mm.

The Von Mises yield criterion says the shear yield stress should be about 0.58 of the tensile yield stress ($3^{-½}$), or 145 MPa, and the shear modulus G = ½E/(1 + nu), where nu is the Poisson ratio (about 0.26 for mild steels) and E is Young's modulus, which works out to about 79 GPa, so the shear yield strain is about 0.18%. 0.18% of 40 mm is 0.07 mm, which is a lot less than 0.25 mm. This means that if the jaws are so thick and solid that they are deforming in shear instead of flexure when we open them, we have lost the game; we need them to deform almost entirely in flexure. (I need to learn how to calculate that.)

0.25 mm over 40 mm amounts to a radius of curvature of 3200 mm and a diameter of curvature of $40^2/0.25$ = 6400 mm.

So suppose we make them 5 mm thick. Assuming pure bending, the jaw's inner edge is 40/3200 radians of a circle of radius 3200+2.5 mm; its outer edge is the same number of radians of a circle of radius 3200−2.5 mm. This amounts to 0.031 mm of compressive or tensile strain, 0.08% strain, which is comfortably less than the 0.125% yield strain limit of the mild-steel material.

But how much force is this biting down on the other plate with if it tries to pull out? Well, that 0.08% strain gives us 160 MPa stress in the material at the extremes, where it's applying a 2.5 mm lever arm. But as the strain diminishes toward the neutral axis of the beam, so does the lever arm. We need the integral from −1 to 1 of x^2 (which is 2/3), times 160 MPa, times (2.5 mm)², times the 3 mm thickness of the material, and we get a torque of 2 N m, which is like 800 N applied at a 2.5 mm lever arm. Dividing this by 40 mm we get 50 N, which is a pretty modest force. You wouldn't even need a special tool.

But that's when it's almost all the way pulled out; it increases from 20% of that (10 N) up to that value as the taper pulls the blades apart, for an average bite force of 30 N. The interesting thing, though, is

the size of the energy barrier: steel on steel has a kinetic friction coefficient of about 0.6, so it's an average of 18 N of friction resisting the pull-out force (plus 30 N × 0.5 mm / 20 mm = 0.75 N). But because that's over 20 mm, the energy barrier is a very significant 0.36 J. If you don't manage to pull it all the way out, it will tend to move back into place when there is vibration.

Shape variations

A purely-2D-cuttable version is to use a sort of triangle-wave pattern of teeth on the jaws of one plate and a series of slots in the other plate to accommodate the crests of the triangles.

This conformation of the connector type is not very proof against being twisted, without further support; if we turn it inside out so we have two tapered prongs that stick through a slot against the inside of which they are pressing, we obtain a much better purely-2D-cuttable connector which resists more degrees of freedom. The slot can be, for example, 3.1 mm × 40 mm to locate the connector in all dimensions to a reasonable degree of precision (and if it's not perfectly rectangular, thinning out to 3.0 mm at the edges, that will improve the precision]; or it could be 40 mm × 40 mm to permit positioning the connector anywhere within it at either of two angles; or it could be longer, say 150 mm × 40 mm, to permit further positioning freedom.

To increase the force without vastly increasing the required cantilevered jaw length, we can use rigid jaws mounted on many parallel flexible thin strips (a section of metal with many parallel cuts through it, perpendicular to the edge but not reaching it), which bend into parallel S-curves to allow the jaw section to move. Since the strips would be bending twice, they'd need to be half as thick to reach the same strain, so 2.5 mm, which also halves the torque (and thus, I think, the force) applied by each one. So to reach a 20 J energy barrier for sliding out of place, we would need about 60 strips, occupying about 150 mm of lateral space and providing about 1500 N of peak bite force, for which you would definitely want a special tool. The energy for opening the jaws would be 0.75 J.

What people normally do in practice in such cases is to use a much steeper slope than the 1:40 slope I'm talking about above, using something closer to 1:1 or even 2:1 or 10:1, creating something more like a hook than an inverted wedge, and a much larger displacement than 0.5 mm. This makes the snap connector self-locking (like a screw or the most common kinds of worm drive) without requiring excessive forces to open it, when the opening forces are applied directly to the snap jaws rather than to the joint they secure. Because rounding over or breaking off the snap hook is an alternate means of failure, typically only one side of the snap fastener moves.

(Providing multiple hook tips, like interlocking sawtooths, is another possibility for reducing the failure risk from the snap hook.)

If the snap jaws are grabbing onto the edge of something or into a slot, rather than poking through a hole or grabbing a small object, having *two* connected sets of jaws in parallel planes positioned some distance apart along the edge would strongly resist two axes of twisting that a single edge-jaw connection is vulnerable to. And if

you want an edge that can be reliably grasped with such an edge-jaw connector, a 2-D-cutting way to get one is, rather than trying to roll a reverse taper into the edge, to assemble something like one side of an I-beam, where a narrow strip of material runs along the edge of a plate of material, perhaps held there by snap connectors protruding through it. If the edge is perfectly straight and there are no jaws clamped onto it, the strip can be twisted around its intersection line, but if the strip is curved or there are jaws grabbing it, it should be held firmly in place.

Compound mechanical advantage and smooth jaws

An exciting thing about hook tips with steep slopes is that they provide further mechanical advantage that's available to press multiple things together. Suppose your hook tips have a 10:1 slope, so 0.4 mm of bite-down movement is permitted by 0.04 mm of snugging up, and you have 1500 N of bite-down force. That multiplies whatever part of the bite-down force isn't stymied by static friction, potentially all of it if you vibrate the joint enough, so you might be able to get 15 kN of snugging force. (But, with a 250 MPa yield strength, you need 60 mm² of cross-sectional area in tension to withstand 15 kN, so you'd probably want to make such a thing out of 6mm mild steel sheet instead of 3mm.)

An interesting thing about this is that the numbers for the forces needed to fasten and unfasten, or the costs of fabrication, don't really depend at all on how wide the actual jaw is (and thus its tensile strength), just the flexure that supports it. This might seem like a distinction without a difference, though: if the flexure breaks under tension, what does it matter that the jaw didn't?

But I think there is a very clever way around this. The movable jaw always has to work against some kind of fixed jaw (whether pressing toward it or pulling away), and the fixed jaw is rigid and not supported on the flexure. Therefore, why not put the hook on the fixed jaw, which can be arbitrarily robust, and make the movable jaw totally smooth so it isn't subject to tensile forces? This is backwards from the usual snap-connector design, but I think it has a really killer advantage in terms of strength.

In the case where we're joining the edge of one plate to the face of another, we can't do the sawtooth thing to prevent the hook from rounding over or breaking off. Instead, we can use *multiple rigid hooks* that poke through multiple slots in the mating plate, all preloaded with the same flexure-mounted smooth movable jaw.

There are some disadvantages to the hooked-fixed-jaw approach. It means that the parts being joined have some play in the join, depending on exactly how far down the hook ramp the flexure has managed to force the mating parts, so they aren't precisely locate relative to one another. And an impact could apply the mass of either of the two parts to the purpose of disconnecting the smooth movable jaw. I think these can be mostly eliminated by using a chain of wedges, like those in a tusk-tenon joint, those used to assemble reusable concrete forms, or those in a cotter pin, with only the last "keystone" wedge in the chain using a compliant snap joint to wedge

the whole chain tighter and tighter when there's vibration. The small masses of the keystone wedge and its immediate wedgee prevent impacts from breaching the energy barrier to disassembly.

Being able to make each connection extremely strong means that you can use fewer of them, which lowers both costs and time for assembly and disassembly.

Of course, if you *were* to apply 1500 N, you need a fastening and unfastening tool.

Design of a 91%-efficient snap-jaw-loading tool

At 0.5 mm displacement, a simple eccentric lever tool like the one in the Abom79 video would have an 0.25 mm eccentric offset and thus maximum lever arm. A comfortable handle length of 250 mm would thus have a M.A. at worst of 1000:1, halfway through the movement, when the bite force is only 750 N, and the handle force thus 0.75 N plus friction. The peak handle force is just a little past that, and then the M.A. starts to increase togglewise. With a force budget of 10 N, you'd only need a peak M.A. of 75:1, and thus a handle length of 19 mm. (A pullstring might still be a better way to apply the force, though.)

With the snap-fit fastener, though, there's almost no friction inside the workpiece, just elastic hysteresis, which is insignificant for most common materials at these speeds. The friction is all inside the tool used to force the snap open, where it's practical to reduce it with material choice and rolling-element bearings, instead of within the part as with a screw. Consider these scenarios:

- The force is applied to a movable snap jaw by inserting a round steel pin from the tool into a dry round steel hole in the snap jaw, and the pin rotates as an integral part of the eccentric as the handle is turned, thus rubbing against the jaw hole while the normal force increases from 300 N to 1500 N. Suppose the pin is 3 mm in diameter; then in a half turn it moves 4.7 mm with an average of 900 N normal force, which with a friction coefficient of 0.7, gives us 630 N of average friction force, consuming an extra 2.96 J every time you opened *or closed* the jaws, in addition to the 0.75 J that gets elastically stored. This is most similar to the situation with ordinary screws.
- Same, but now the pin is bronze or zinc, so the coefficient is reduced to 0.22, and instead of 2.96 J it's 0.93 J. A big improvement already, and one you can't get with screws unless you either put bronze sleeves in all your screw holes or make all your screws out of bronze.
- Same, but now the pin has a teflon sleeve. This drops your coefficient of friction to about 0.04 to 0.10, but teflon has a compressive strength of only around 10–15 MPa so withstanding 1500 N requires a 150 mm² contact area, so now you might need to make the "pin" and its hole 50 mm wide to not wear out, which would make this less efficient instead of more, not to mention less convenient.
- Same, but now instead of rotating as an integral part of the eccentric, the pin runs through a pair of ball bearings pressed into the eccentric, giving an effective frictional coefficient around 0.0008 to

0.0015. At this point, though, we need to worry about the bearings on which the eccentric shaft as a whole rotates, because they become a more significant source of friction; suppose the pin is on 608 roller-skate bearings with an 8mm bore and 22mm outer diameter. Then the eccentric needs to be at least 22.5 mm in diameter and probably more like 24mm, so at the business end its 24-mm-bore bearing is resisting that same 750 N average force with the same 0.0015 effective frictional coefficient, but over 38 mm of rotation instead of 4.7 mm. So that bearing consumes 50 mJ, while the pin's 608 bearing consumes 17 mJ, for a total of about 67 mJ. SKF rates their 608 deep-groove bearings for 3450 N dynamic, 1370 N static, so this is sort of marginal, but that's for a million revolutions before fatigue.
• Same, but now the eccentric isn't a constant diameter, but has a sort of dogbone shape; its two ends are 25 mm in diameter with an eccentric 22mm bored out of them (0.25 mm off center) to press the 3 mm pin's bearings into, but those two ends are connected by an 8-mm OD pipe which is itself pressed into two more 608 bearings. The 3mm pin runs through the ID of the pipe, which is, say, 4mm. Now the eccentric's bearing also consumes only 17 mJ for a total of 34 mJ, and as a bonus, the bearings cost 50¢ each instead of 600¢ each, like a 24-mm-bore 6802 would.

These 34 mJ are 4.5% of the 750 mJ needed to activate the mechanism, but we pay them twice: once to load the snap jaw, once to unload it. So it's only 91% efficient.

Consider the mechanical advantage of the screw. A coarse M6 screw has 1mm thread pitch and might be driven by a screwdriver with a 25mm-diameter handle, so each 79 mm of screwdriver motion produces 1 mm of axial screw motion. This is a 79:1 mechanical advantage. To get 15 kN of snugging force you would need to apply 200 N of force at the surface of the screwdriver; you might need a T-handled screwdriver or something. But with an appropriate safety factor an M6 screw can handle a snugging load of only about 2090 N in strength class 12.9, because its effective cross-sectional area is only 20.1 mm². To handle 15 kN you would need a larger screw --- the 60 mm² cross-sectional area I suggested above would be a little bigger than an M10, but Misumi recommends a safety factor of 3–15 depending on the situation, and so a at least an M24. And these larger screws have coarser thread pitch, further decreasing the mechanical advantage.

How inexpensive?

McHone says laser cutting steel costs US$13–$20 per hour, and 70" per minute is a common cutting speed, which works out to 12¢–19¢ per meter of cut, but sometimes as slow as 20" per minute, which would bring the cost to 66¢ per meter. (Also they suggest 0.005" precision, which is 127 microns in modern units.) AST Manufacturing says plasma is 0.020" precision (500 microns) on materials thicker than 1", but laser can hit 0.003" (76 microns) and up to 1575 (!) inches per minute.

OSHCut says the nominal 1200-ipm speeds for the laser they bought are misleading because *acceleration* is the limiting factor, not cutting speed. So curvy contours, angles, traversing between

contours, and especially piercing are slower than cutting the contours themselves, but also they say that on their 3kW Trumpf 1030 fiber laser ¾”-thick steel (19mm) cuts at a maximum of 47 inches per minute (20 mm/s), while 0.04”-thick steel (1mm) cuts nominally at 1160 inches per minute (490 mm/s). They measure curviness in radians per inch, and their plot seems to show that at 5 radians per inch (200 radians/m) the 1160 drops by a factor of 8, I guess to about 150 inches/minute (61 mm/s). Even at 1 radian/inch (39 radians/m) it was down to ¼ of max (I guess 290 inches/minute, 120 mm/s). Still higher curvatures added less penalty; even at 30 radians/in. (1200 radians/m) the penalty for 1-mm steel was only about 17×, so I guess 68 in/min or 29 mm/s. For thicker metal the curviness penalty was less, basically no measurable penalty for the 19-mm stuff. Their plot isn’t super well labeled but one of the thicknesses is supposed to cut at 335 ipm (141 mm/s) and I’m guessing that might be 3mm; at 5 radians/in. (200 radians/m) it was only 2× as slow (70 mm/s?) and even at 30 radians/in. (1200 radians/m) the penalty was only 4× (35 mm/s?).

Another way to look at these curvature numbers is as the characteristic feature size, although especially good CAM software can avoid this sometimes for some designs; a closed convex polygon of any shape is 2 pi radians, so a curvature of 200 radians/m means that your closed polygons are about 31.4 mm in perimeter, while 39 radians/m means they’re about 160 mm in perimeter, and 1200 radians/m means they’re 5.2 millimeters in perimeter, which is pretty impressive if your kerf is close to the typical 0.2 mm. Except that it doesn’t include pierce times, so maybe they weren’t really cutting 6 polygons per second, maybe more like 24 radius-0.8-mm 90° corners per second.

Higher-powered 12-kW lasers can cut 12mm steel at 150 ipm, 64 mm/s, but even for straight cuts, laser cutters from 6 kW to 12 kW on 0.5-mm steel all topped out at 3150 ipm (1.3 m/s). JMT has a cutting speed chart cutting 1-mm steel with 4kW from 280 to 2362 ipm (120-1000 mm/s) and 3-mm steel at 120-380 ipm. Interestingly they needed to use either O2 or N2 gas for such thick steel; air was insufficient. Air could cut thin steel faster, and nitrogen faster still, but oxygen was needed for steel over 6.4 mm.

Plasma is cheaper and faster than laser as of 02018 anyway. This article also gives me the term “XY cutting processes” to cover plasma, laser, and waterjet, and mentions the acceleration limitation. Laser vendor Trumpf disagrees, saying plasma is more expensive per hour, while laser is only US$3/hour in operating costs, and also cuts faster (114 ipm (48 mm/s) in ¼” (6.4mm) mild steel rather than “typical plasma”’s 70 ipm, which commenters say is much slower than typical).

A company in Monte Castro called ZYX Mecanizados (4674-6826, zyxmec...@gmail.com) is one possible vendor of the service; they offer plasma, CNC router, laser, and hot-knife cutting, for up to 1000 × 1300 mm. Their website is down but dates from 02014 and it says they cut lots of things but not ferrous metals. There seem to be some other vaguely relevant companies listed in its category on MercadoLibre, but that is probably not really true. Near here Google Maps lists Plasmacenter (Perón 1898, Lomas de Zamora,

4282-3855 or maybe now 6233-5443), but they just sell CNC plasma, oxy, and laser machines. And in Palermo there's S.A.D.I. Metales, whose web site says they do CNC plasma (1.6 mm to 12.7 mm) and oxy-fuel (4.7 mm to 300 mm) cutting, but who said they just do laser last time I asked. They sell sheet metal from 1 mm to 6.4 mm in thickness.

Sheet steel is sold here as construction material in a weird mix of medieval and modern units: 1500 mm × 3000 mm × 3.2 mm (0.125") for AR$28650, which at today's rate of AR$163/US$ is US$176, or US$39/m²; 1220 mm (4 feet) × 2440 mm (8 feet) × 1.25 mm (18-gauge), 20 kg, for AR$9100, US$56, US$19/m², but it also can't possibly be those dimensions or it would be 29 kg; 1 m × 2 m × 1.25 mm (18-gauge), 20.5 kg, SAE 1010, for AR$7458, US$46, US$23/m², which ought to be 19.75 kg, which is close enough; 1.22 m × 2.44 nm × 2.0 mm (14-gauge) for AR$15043, 49 kg, SAE 1010 from Hierros Torrent, US$92, US$31/m². These are all in the range US$1.43-US$2.80/kg, with the thinnest gauges costing nearly as much per square meter and thus far more per kg.

So, to get a cost estimate, let's try scaling down the hook thing a little. Say we're satisfied with the 2kN snugging load a strength-class-12.9 M6 screw can handle, and we get a 10:1 M.A. from the angle of the hook (so we only need 200 N of bite), and 500 microns *is* enough. And to get 200 N over 500 microns, let's say my calculations above are okay and each additional 2.5-mm-wide 40-mm-long strip in 3-mm mild steel would give you 25 N of bite, so you need 8 of them; a 20-mm-wide strip cut lengthwise into 8 straight 40-mm strips with 7-9 cuts (although where do we pierce?). And let's say the US$15/hour number is about right, and 70 mm/s (one of what I think are OSHCut's numbers) is about right for these cuts. That works out to 6¢ per meter of cut, or 1.9¢ to cut the springs for each such snap connector. The connector itself might occupy more like 1000 mm² (0.001 m²), so the steel thus used costs about 3.9¢, but it can also be serving other roles as structural support.

(At 200 N we don't need a special tool with eccentrics and bearings and stuff. We can comfortably use snap-ring pliers with an M.A. of 3:1.)

A box of 50 15mm M6 screws costs AR$540 or US$3.31, or 6.6¢ per screw, but those aren't strength class 12.9 or actually any declared strength class. A box of 25 20 mm class-12.9 M6 screws costs AR$656 , 8¢ per screw. Drilling and tapping the holes for a screw costs extra, probably a comparable amount.

Topics

- Contrivances (p. 1143) (45 notes)
- Pricing (p. 1147) (35 notes)
- Digital fabrication (p. 1149) (31 notes)
- Manufacturing (p. 1151) (29 notes)
- Physics (p. 1157) (18 notes)
- Mechanical (p. 1159) (17 notes)
- Strength of materials (p. 1164) (13 notes)

- Machining (p. 1165) (13 notes)
- Hand tools (p. 1197) (7 notes)
- Argentina (p. 1200) (7 notes)
- 2-D cutting (p. 1201) (7 notes)
- Steel (p. 1222) (5 notes)
- Flexures (p. 1232) (5 notes)
- Spreadtools (p. 1321) (2 notes)
- Snaps (p. 1325) (2 notes)

Smolsay: the Ur-Lisp, but with dicts instead of conses

Kragen Javier Sitaker, 02021-07-12 (updated 02021-07-27) (22 minutes)

What if we wanted a small dynamically-typed imperative language with the flexibility of Lisp that admitted reasonably efficient implementations, but not based on conses? Like, something with an implementation about the size of the ur-Lisp? The power of first-class hash tables ("dictionaries" or "tables" or "associative arrays") has been convincingly shown by JS, Perl5, PHP, Python, and Lua, and they certainly produce much more readable code than Lisp.

Lua has shown that freeform syntax without statement terminators can be reasonably usable (though it mostly prohibits you from using juxtaposition as an operator, as ML does for function composition, and it limits your flexibility in what expressions can start with), and Python has shown that indentation-based syntax can be too, at least if you don't demand too much from anonymous lambdas.

Statement-level structure

We can start with variables. Imperative languages probably need variable updates, but most variables should be declared and initialized and then not mutated, so it makes sense to privilege the declaration form over mutation with brevity; and ideally the thing being declared should be in the left margin, permitting easy scanning, rather than preceded by a keyword or punctuation:

```
decl ::= name "=" expr
assi ::= "set" lvalue "=" expr
```

Multiple assignment (and multiple declaration) is convenient syntactic sugar at times, especially with Lua-style or Perl-style multiple return values, but it doesn't add any fundamental power if we have dicts. And I think non-multiple assignment tends to push code toward being "boring" rather than "clever", which is worthwhile here.

So an lvalue is just a dictionary lookup chain from a name. A pathname, you might say. I think it's really important to be able to say literally `p.x = 3` rather than the much noisier options like, say, `p{x} = 3` or `p:x = 3` or `p.:x = 3` or `p['x'] = 3` or `x(p) = 3`, or a more implicit option like `p x = 3`. But it's also important to be able to index with expressions rather than literal symbols like `:x`. One approach to this would be to distinguish literal symbols by case: perhaps `X` would be a literal symbol and `x` a variable, so `p.X = 3` would index by the literal symbol, but `p.x = 3` would use whatever the current value of `x` was. But I think maybe a better approach is to use a parenthesized expression, so `p.x = 3` assigns to the property `x`, while `p.(x) = 3` reads the variable `x` and assigns to the property it names. Thus `.()` takes the place of `[]` in most conventional languages.

```
lvalue ::= name ("." arc)*
arc ::= name | "(" expr ")"
```

Now we need conditionals, functions, and (if this is really imperative) iteration. My Lisp sense tells me that conditionals and iteration should be expressions rather than statements; my Lua and Python sense tells me that this might make parsing errors unusable (Lua's parsing can reliably distinguish a function call from a trailing expression followed by a new statement beginning wih `(` or `[` because Lua statements can't begin with those because Lua doesn't have expression statements) but in any case they should use conventional words; my C sense tells me that I should use curly braces. So, conditionals:

```
if ::= "if" expr "{" expr+ "}"
       ("elif" expr "{" expr+ "}")*
       ("else" expr "{" expr+ "}")?
```

Since we don't have multiple value returns, the value returned by the conditional is the value of the last expression in the chosen branch.

Function calls are similarly syntactically simple, although they involve the expression-juxtaposition danger that shows up in JS, since presumably we will allow expression parenthesization:

```
call ::= expr "(" (expr ("," expr)*)? ","? ")"
```

The arrow syntax from current JS is the lowest-hassle way to define an anonymous function. Semantics are that it is a closure with arguments are passed by value.

```
lambda ::= "(" (name ("," name)*)? ","? ")" "=>" "{" expr* "}"
```

Tentatively I'm using Sam Atman's `->` syntax from Lun for function return values:

```
return ::= "->" expr
```

This allows us to write `add1 = (x) => { -> x + 1 }`, which is lightweight enough to not wish for a sugared `function add1(x) { -> x + 1 }` form.

Minimally iteration needs a while loop, but most loops are better expressed as iteration over a sequence:

```
while ::= "while" expr "{" expr* "}"
for ::= "for" name "in" expr "{" expr* "}"
```

List comprehensions in Python are extremely useful, so these ought to return sequences, but what should they contain? The conventional answer would be the last expression invoked in the loop body, and that seems adequate.

The Lisp approach to sequences is to use cons lists, in which case the "for" structure would desugar somewhat as follows:

```
for x in y { z }

yi = y
while !yi.null {
    x = yi.car
    z
    set yi = yi.cdr
}
```

A different approach would use arrays and perhaps slices:

```
yi = 0
while yi < y.len {
    x = y.at(yi)
    z
    set yi = yi + 1
}
```

The remaining fundamental means of combination is explicit aggregate variable construction. Unlike in JS, it's syntactically unambiguous to use {} here because all our previous uses of { were semantically obligatory and thus cannot occur where an expression is expected.

```
dict ::= "{" (arc ":" expr ("," arc ":" expr)* ","?)? "}"
list ::= "[" (expr ("," expr)* ","?)? "]"
```

So, the basic expression language then is

```
expr ::= decl | assi | if | call | lambda | return | while | for
       | dict | list | infix
```

Infix expressions

Pop infix syntax is fairly straightforward; basic arithmetic is:

```
atom ::= "(" expr ")" | string | int | real | symb | lvalue
unary ::= atom | "-" atom | "!" atom
expo ::= atom ("**" exp)*
term ::= expo (("/" | "//" | "*" | "%") expo)*
terms ::= term (("+" | "-" | "..") term)*
```

Here `..` is Lua's string concatenation operator. I think it's safe to relegate bitwise arithmetic to named functions.

There's an unfortunate precedence ordering thing in C where booleans have precedence close to the bitwise operators, tighter than comparisons, rather than the more desirable looser-than-comparisons thing. Another problem is that chained comparisons (`x == y == z`) have unintuitive results in most languages. A simple solution is to restrict the syntactic composability of these elements, requiring the use of parentheses to disambiguate:

```
infix ::= terms (( "==" | "!=" | "<"  | ">"
                 | "<=" | ">=" | "&&" | "||") terms)?
```

A slightly better solution is to provide a separate case for the associative short-circuiting Boolean operators that does allow them to be individually chained:

```
infix ::= terms (( "==" | "!=" | "<"  | ">" | "<=" | ">=") terms)?
        | terms ("&&" terms)+
        | terms ("||" terms)+
```

Tokens

Strings, numbers, names, and symbols are simple enough. The {} around these productions indicate that whitespace should not be skipped within them.

```
string ::= {"\"" ([^\\"] | "\" byte)* "\""}
int ::= {"-"? [0-9]+}
real ::= {"-"? ("." [0-9]+ | [0-9]+ "." [0-9]*)}
name ::= {[A-Za-z_$] [^] \t\r\n(){}[.=!<>+-*/%&|#]*}
symb ::= {":" name}
```

Whitespace itself, implicitly ignored elsewhere, includes comments to end of line, which are marked with Unix # rather than Ada/Lua --.

```
whitespace ::= ([ \t\r\n] | "#" [^\r\n]* (\r\n | \n | \r))*
```

Modules and scoping

There's no need to put an import statement into the program grammar; the "." syntax will work perfectly well for reaching into modules if there's an ordinary function that imports a module and returns it, like JS's `require`. We can simply declare, Python-like, that global variables in a file are the properties of the corresponding module.

So then we simply have

```
module ::= expr*
```

I'm convinced that lexical scoping is adequate and the right default.

The whole Smolsay grammar (Smolbutswol?)

```
module ::= expr*
expr ::= decl | assi | if | call | lambda | return | while | for
       | dict | list | infix
decl ::= name "=" expr
assi ::= "set" lvalue "=" expr
lvalue ::= name ("." arc)*
arc ::= name | "(" expr ")"
```

```
if ::= "if" expr "{" expr+ "}"
       ("elif" expr "{" expr+ "}")*
       ("else" expr "{" expr+ "}")?
call ::= expr "(" (expr ("," expr)*)? ","? ")"
lambda ::= "(" (name ("," name)*)? ","? ")" "=>" "{" expr* "}"
return ::= "->" expr
while ::= "while" expr "{" expr* "}"
for ::= "for" name "in" expr "{" expr* "}"
dict ::= "{" (arc ":" expr ("," arc ":" expr)* ","?)? "}"
list ::= "[" (expr ("," expr)* ","?)? "]"
atom ::= "(" expr ")" | string | int | real | symb | lvalue
unary ::= atom | "-" atom | "!" atom
expo ::= atom ("**" exp)*
term ::= expo (("/" | "//" | "*" | "%") expo)*
terms ::= term (("+" | "-" | "..") term)*
infix ::= terms (( "==" | "!=" | "<"  | ">" | "<=" | ">=") terms)?
        | terms ("&&" terms)+
        | terms ("||" terms)+
string ::= {"\"" ([^\\"] | "\" byte)* "\""}
int ::= {"-"? [0-9]+}
real ::= {"-"? ("." [0-9]+ | [0-9]+ "." [0-9]*)}
name ::= {[A-Za-z_$] [^] \t\r\n(){}[.=!<>+-*/%&|#]*}
symb ::= {":" name}
whitespace ::= {([ \t\r\n] | "#" [^\r\n]* (\r\n | \n | \r))*}
```

This still contains the frustrating ambiguity where an expression immediately followed by `(` is a call, but expressions can also begin with `(`. The alternative of replacing `sin(x)` with something like `sin:(x)` or `sin[x]` is unappealing. It's possible to disambiguate in these cases by assigning to a dummy variable: `_ = (foo)`. Requiring no whitespace or at least no line breaks before the paren would pretty much solve the problem, but it sort of requires that the parsing of `expr` not consume following whitespace.

Because conditional blocks are always wrapped in braces, the if-else ambiguity doesn't occur.

Resulting design at other levels of abstraction

This suggests the following set of 23 basic "bytecode" operations: getlocal, setlocal, get, put; jumpfalse, jump, call, return; makedict, makelist; constant; negative, not, exponent, truediv, floordiv, mul, mod, add, sub, cat, eq, lt.

Of course, the proper order of implementation would be something like:

- Implement bytecode interpreter with textual bytecode syntax and program in it a bit.
- Implement S-expression syntax, compiling to the bytecode, and program in it a bit.
- Implement debugger stuff, maybe native-code compiler, etc.
- Implement pop infix syntax above.

The S-expression syntax can get by with six special forms:

- `(lambda args body...)`
- `(return x)`
- `(cond xy...)`
- `(while p body...)`
- `(for x y body...)`
- `(setlocal n v)`

The other bytecode operations are either implicit (getlocal, call, constant), folded into the three control structures, or ordinary functions (get, put, makedict, makelist, negative, not, exponent, truediv, floordiv, mul, mod, add, sub, cat, eq, lt), along with the other ordinary functions (require, print, length, etc.).

From the point of view of the pure ur-Lisp, if we strip away the imperative and arithmetic frippery, we're replacing ATOM, CONS, CAR, and CDR with get, put, makedict, and makelist, and makelist is unnecessary.

Of course, other alternatives to S-expressions exist. RPN, for example (which is on my mind lately because I've been hacking on PDF and PostScript) or Prolog notation, or REBOL-style non-R PN, or APL-style precedence-free infix.

Alternative debugging-first semantics

What if functions returned their internal namespaces instead of an explicit return value? You wouldn't need a return statement or a dict type, and in most cases debugging would get a lot easier.

A hairier approach to this would be to associate the internal namespace with the value thus produced, and provide a function `why(x)` that returns the activation record of the function call that returned `x`. This implies that returning a value sort of makes a copy of it to potentially associate it with a different activation record, but accessing it as a property does not; we want `why(p.x)` to work. Other internal operations can be treated as functions.

That is, a variable/property has a *value* aspect, the usual reference; but it also has a *why* aspect, which is the activation record of some function, accessible via the `why` built-in function. Both of these aspects are returned when a function returns and stored in the variable/property. It might as well also have other aspects useful for debugging, like `prev`, which gives you the variable as it was before being overwritten, and `where`, which tells you what statement did the overwriting (or initializing), and in what activation record. An activation record might also have a `caller`; often `caller(why(v))` will be `where(v).call`. You might also want to know the control-flow context of the `where`, with something like `where(v).context.condition` to find out why the if or while statement was continuing.

If all this data is always available, no activation record and indeed no value ever becomes garbage, so only very short programs can run fast. (We're in an imperative world here, so we can't just recompute the activation record on demand like Bicicleta.) We could maybe store it in some kind of a ring buffer, like Cheney on the MTA.

The no-return-statement variation doesn't have this problem; `sin(x)` is a whole activation record, but `sin(x).val` is just the return

value, so you only retain activation records when you want them. In Bicicleta I had syntactic sugar for this: `sin{x}` and `sin(x)` respectively, the second of which extracted a variable annoyingly called `()`.

Efficient implementation

A standard pointer-bumping generational GC should work pretty well here, though maybe not quite as well as in a strictly immutable language. If we use a standard sort of stack structure rather than heap-allocating activation records, we need to scan the stack for roots on nursery collections anyway, so we don't need a write barrier for writes to the stack, which are statically apparent, nor for property initializations. Only property mutations and module-variable mutations need a write barrier.

To statically resolve callees, we'd like to be able to treat the module variables in which "global" functions live as constant. Static analysis can't show that module variables are never mutated, but we could maybe trigger a recompile after mutating such a cell. This makes the property-mutation write barrier more expensive, and it might require on-stack replacement (with a recompilation of the same code, not a compilation of the new function), which is easier if our stack-frame structures aren't too optimized. Debugging is also easier if those structures are ordinary dictionaries.

Statically compiling local-variable accesses into indexing off a stack frame that is also an ordinary dictionary would be easier if we could ensure that dictionaries never changed shape, as local-variable vectors normally don't. If you can delete values from a dictionary, which is probably important, there's the question of what happens when you try to read the deleted value. If getting nil as in Lua is an acceptable answer, then you can ensure that dictionaries never change shape. But, if you want that to raise an exception, you either need to check for existence every time you read a local variable, as Python does, or you need to recompile the function to crash (and potentially do on-stack replacement) when you delete the variable from its stack frame.

Or you could just raise an error when you try to delete a variable from a stack frame, which is probably the most sensible thing to do --- changing their values is reasonable but deleting them is not. That would require that the deletion operation do a special check for such protection --- which is fine, because key deletion is so rare that it doesn't need to be especially fast.

The space-efficient way to make such early-bound accesses fast would be to put dictionaries' keys in one vector (or set of vectors, but probably just one in this case, though maybe it would be worthwhile to use the set-of-vectors approach to handle lexical scopes) and the corresponding values in another one, which would be in the stack frame. For large key sets, a canary hash table for the key set would enable fast access by key. Newly added keys would go into a new key vector.

This kind of design, where the stack frame structure is a regular first-class data value that happens to be efficient enough to use for local variables, seems like it might be an appealing alternative to dynamic deoptimization.

(It might be useful to expose the inheriting-dictionary approach used to handle nested scopes as first-class in the language itself, so that you could, for example, put a class's methods in one dictionary and then create objects that inherit from it, or efficiently override a few things in a large dictionary.)

This structure would also make the common makedict case, where the set of keys is constant, fast and space-efficient. And it might ease the task of "hidden class" compilers.

Initialization can be statically guaranteed for all local variables if variable declarations are block-local rather than function-local.

What about dynamically dispatched methods?

We can use closures as methods, of course, or we could supply syntactic sugar like Lua's `obj:method(args)`, which translates to `obj.method(obj, args)`, although both `:` and `->` are already taken in the syntax above. `obj.method[args]`, `obj << method(args)`, `obj <- method(args)`, `^method(obj, args)`, `@method(obj, args)`, and `method = generic(:method) ... method(obj, args)` come to mind. But we'd like to be able to do this in a way that can be compiled efficiently in a dynamically-typed system, which means we need some way to quickly verify that the method to invoke is the same as last time. So we'd like the memory representation of `obj` to be possessed of some easily readable register-sized value that can be efficiently compared against the one for the method we think we ought to call.

One approach is to attack the problem entirely at the desugared level: if we're first fetching a property from the object, then invoking the resulting closure, we could try to speed up the property fetch, and/or we could compare the resulting property against the last closure we invoked at this callsite (or a PIC of them). This would be fine for a tracing JIT but it seems unlikely to provide much benefit for a simpler compiler. If your objects are made out of a delegation chain of dictionaries as described earlier, you can store the chain in a counted vector in the dictionary object, and you can copy the chain item in which you found the method to your inline cache, along with a number that says what index it was at. Then you can compare the cache-key dictionary pointer to the dictionary pointer in the delegation chain at the appropriate position, if it's within bounds. But then you need to worry that a later (more superficial) dictionary in the chain might be hiding the method, so you can also maybe store an index in the object that tells which is the last dictionary in the chain none of whose keys are hidden, and use *that* instead of the dictionary where you actually found the keys. Hopefully this will allow you to have cache hits for "objects" in the same "class", since instance data doesn't normally hide methods.

Another approach is the Perl/Python `bless` approach, where the instance data of an object is in one dictionary, and its methods are in another, providing a totally separate namespace and enabling you to use dicts with much greater security than in JS. Lua does this with `setmetatable`: the things that tell Lua how to index a table or convert to strings are stored in a separate table, the metatable, which may be shared among many objects. So maybe `obj << method(args)` should

desugar to what we'd write in Lua as `getmetatable(obj).method(obj, args)`.

This is somewhat less conceptually simple (a dict isn't just a set of key-value pairs; instead it has become merely the puppet of a shadowy class) but much easier to compile efficiently and much safer. You don't need to grovel over a delegation chain to figure out what the best cache key is; it's just the class, which doesn't need to support inheritance. It would be better to put regular user methods in the class too, unlike Lua, which is at this intermediate point where methods like **index and** tostring are in the metatable as in Perl and Ruby, but you're expected to put ordinary user methods in the same namespace as your instance variables or dictionary items, as in JS.

I think this is slightly less simple than the current Lua approach, since in Lua, you don't need to add a metatable to a table in order to put methods on it. but the metatables are already in the picture and it's already common to `return setmetatable(...)` in object constructors, and in those cases it seems like it would just simplify user code, as well as having the above advantages.

One of the things I really enjoyed about LuaJIT's FFI (by contrast to Python's cffi) is that I can just add Lua methods to C structs with `ffi.metatype(ctype, { __index = { methods } })` instead of having to do the whole song and dance of the membrane pattern where I wrap each C type in a separate porcelain object, everywhere it's returned through the looking glass (and possibly going to some effort to keep that relationship 1:1). But the standard Lua way of defining ordinary methods means that if I want to do the same thing for an ordinary table that's used as a dictionary, I do have to create a separate porcelain façade to hold its methods.

Some notes on COLA-like object models are in Faygoo: a yantra-smashing ersatz version of Piumarta and Warth's COLA (p. 570).

Topics

• Programming (p. 1141) (49 notes)
• Lisp (p. 1174) (11 notes)
• Virtual machines (p. 1182) (9 notes)
• Small is beautiful (p. 1190) (8 notes)
• Programming languages (p. 1192) (8 notes)
• Higher order programming (p. 1196) (7 notes)
• Systems architecture (p. 1205) (6 notes)
• Syntax (p. 1221) (5 notes)
• Bytecode (p. 1236) (5 notes)
• Garbage collection (p. 1255) (4 notes)
• Dynamic dispatch (p. 1259) (4 notes)
• LuaJIT (p. 1353) (2 notes)
• Lua (p. 1354) (2 notes)
• The JS programming language (p. 1359) (2 notes)
• Debugging (p. 1375) (2 notes)
• Lun

Electrolytic berlinite

Kragen Javier Sitaker, 02021-07-12 (updated 02021-12-30) (7 minutes)

Thinking more about 3-D printing phosphate rocks. The prince of phosphates is the quartz-like berlinite, an aluminum phosphate. Obvious solution: squirt soluble aluminum salts like aluminum chloride into soluble phosphate salts like diammonium phosphate, or vice versa, to precipitate it, maybe in the interstices of some dirt.

(The other soluble aluminum salts are the sulfate, the nitrate, the lactate, and the perchlorate, as well as the alums, and any of them could be used; sodium aluminate might also be worth a mention, since it precipitates out the insoluble aluminum hydroxide gibbsite in anything but a very alkaline environment. Potassium aluminate, which has very similar properties, is sold for phosphorus-precipitation water treatment.)

Next step: what if we produce the aluminum salt electrolytically on demand from an aluminum electrode? Maybe run NaCl over the electrode to produce aluminum chloride or sodium aluminate. Probably you still need to do the electrolysis in a clean chamber and squirt the result into some kind of powder or solution instead of just sticking the electrode into mud and applying a voltage, unless buildup on the electrode itself is what you're going for.

But producing aluminum itself is costly and demanding in a variety of ways. What if we don't have aluminum metal, just an aluminum ore like bauxite? Maybe we could electrolytically produce soluble aluminum salts from it and squirt them out; for example, with a chloride electrolyte like NaCl and a silver-plated, gold-plated, or graphite anode, or a sulfate electrolyte and a lead or graphite anode, or a sodium-salt electrode and a steel cathode. The alums occur naturally on Earth in water-soluble form, so in their cases you wouldn't even need electrolysis.

What if you instead want to extract the phosphate electrolytically? Maybe a slurry of apatite with a sulfate salt (sodium, say) and a lead or graphite anode could solubilize phosphate while immobilizing most of the calcium, and the result could be directly squirted onto a power containing aluminum cations, maybe even in an insoluble form like the trihydroxide, to form the phosphate of aluminum.

You might even be able to carry out both processes simultaneously, thus eliminating the need to neutralize unwanted ions from the other half of the electrolysis.

In both these aluminum-metal-free processes, you could probably skip most or all of the usual refining of the mineral feedstocks, because the usual contaminants either mostly won't dissolve or won't harm the desired cement-formation reactions. For example, bauxite often contains troublesome impurities of reactive silica, hematite, rutile/perovskite, and lime, all of whose cations would react with phosphate to form hard, insoluble rocks in much the same way as aluminum. Such a mix might have more favorable properties than pure aluminum phosphate; for example, species like hydroxyapatite

that nucleate and grow rapidly could provide early strength, especially if they form acicular or platy crystals, allowing time for slower-growing crystals of other minerals to form over a longer period of time.

Aside from the potential for 3-D printing, this family of processes could also be used for grouting soil to stabilize it as a foundation for buildings or civil-engineering projects like highways.

Massive crystalline berlinite is fairly hard and inert, so if that can be persuaded to form, it should be possible to reveal the resulting shape by abrading or dissolving away the softer or more reactive ingredients.

Another interesting possibility is the mineralization of wood (or other porous substances that are easy to shape, such as carbon foam or plaster of Paris) by soaking such materials into it one after the other (perhaps forcing them in under high pressure), hopefully forming a surface layer of wood enhanced with the incombustible, hard, stiff, insoluble phosphate mineral. This obviously has potential disadvantages: the phosphate is not the only product of the reaction --- aluminum chloride and diammonium phosphate, for example, will also produce ammonium chloride; the solvent remains, and may have difficulty escaping the low-porosity material; and the ingredients may not be completely consumed, and in some cases may attack the wood over time, for example hydrolyzing it.

This sort of thing seems like a potential appealing low-cost hydrothermal route to "ceramic-matrix composites", somewhat similar to the bargain-basement gypsum remineralization described in Making mirabilite and calcite from drywall (p. 564). First, layup and fixation of the reinforcing fibers (perhaps zirconia, alumina, mullite, carborundum, graphite, basalt, metals like steel, or ordinary glass fiber), using some sort of very porous adhesive (perhaps a minimal amount of plaster of Paris), and sizing of the fibers with a lower-strength material to enable fiber pull-out. (Traditional CMC manufacturing uses pyrolytic carbon or boron nitride for this sizing.) Second, saturation of the fiber reinforcement with one of the salts that will produce the matrix, followed by drying it to remove the unnecessary water. Third, application of the other salt to form the desired mineral matrix, such as berlinite.

To form large crystals, capable of bridging larger gaps, it's desirable for reaction conditions to be just barely favorable for crystal nucleation. For some systems, this can be achieved by temperature control, but that seems unlikely to help much in this case, given the very low solubility of the aluminum phosphate in water at any reasonable temperature. If the amount of aluminum phosphate dissolved in the water never rises very high, this could help, although at the cost of making the reaction take a long time. This requires limiting the rate of the reaction that produces it, either (unlikely) by very low temperatures, or by having a very low concentration of one of the ingredients.

For example, if solid gibbsite is the source of the aluminum, you could flow an abundant amount of very dilute diammonium phosphate through it, perhaps forming aluminum phosphate at a low enough rate that nearly all of it would deposit on existing phosphate

crystals rather than forming their own. But this might simply encase the gibbsite crystals in an inert layer of berlinite, passivating them. The difficulty with doing this with two water-soluble ingredients is that flowing an abundant amount of a very dilute ingredient through the porous object would wash away the other ingredient.

If it's possible to supply one of the salts as a vapor instead of a liquid solution, that could perhaps solve this problem, because a low concentration could be applied for a period of time. Aluminum chloride sublimes at 180°, for example, while the soluble phosphate remains liquid even after losing all its water at 212°, so perhaps hot aluminum chloride vapor could form berlinite on phosphate-bearing surfaces in the 100°–200° range.

If using the phosphate as a cement to selectively join an aggregate (a functional filler), obvious candidates for the aggregate include clay grains, quartz, and sapphire, aside from the fibers and whiskers mentioned previously.

Topics

- Materials (p. 1138) (59 notes)
- Digital fabrication (p. 1149) (31 notes)
- Electrolysis (p. 1158) (18 notes)
- 3-D printing (p. 1160) (17 notes)
- Filled systems (p. 1161) (16 notes)
- Phosphates (p. 1184) (9 notes)
- Minerals (p. 1210) (6 notes)
- Cements (p. 1235) (5 notes)
- Ceramic-matrix composites (CMCs) (p. 1265) (4 notes)

Making mirabilite and calcite from drywall

Kragen Javier Sitaker, 02021-07-12 (updated 02021-12-30)
(4 minutes)

Glauber's salt is an appealing material for both phase-change thermal energy storage and thermochemical energy storage, but the usual ways to make it might be inconvenient at small scales, particularly in places where the police consider such activities suspicious. So how can you make it from non-suspicious materials?

It occurs to me that gypsum is much more water-soluble than calcite, particularly at pH above 7, so in an equilibrium between solid gypsum, solid calcite, and some aqueous solution with some other cation that is highly soluble with both anions, there will be very little gypsum left. So perhaps you can boil drywall (plaster of Paris) in baking soda for a while (months?) to get Glauber's salt, or just boil the baking soda to liberate the carbonate ion and then soak the plaster in the resulting liquor (for months?) after cooling.

Aside from its potential use as a source of mirabilite, this remineralization process might be a useful way to strengthen objects initially shaped from plaster of Paris (or composites cemented with it), at least if they survive it intact rather than falling to pieces or swelling out of shape. Plaster of Paris is easy to mold and sets up in a few minutes; it's easy to abrade, cut, and burnish to precise dimensions once it's set; it's a fairly non-corrosive environment; and it adheres well to a wide variety of functional fillers, including traditional fillers like quartz sand, horsehair, and various forms of cellulose fibers, but also I think higher-performance fillers like fiberglass which are never used with it in practice. It would be very valuable to convert the resulting weak plaster or plaster-cemented composite, which will simply crumble with enough exposure to water, to the much stronger calcite or a calcite-cemented composite like lime mortar. (Matweb gives limestone as 5–25 MPa tensile, 14–255 MPa compressive, but that's presumably full of cracks; I'm having a hard time finding properties of the pure minerals.)

Lime-mortar buildings have stood for thousands of years in rainy areas, and calcite can also withstand much higher temperatures than plaster of Paris for many cycles. (Plaster of Paris won't melt, but it dehydrates below 200°, shrinking and losing strength, while calcining calcite back to quicklime requires temperatures over 600°, normally over 800°.) Moreover, the use of lime cement in the traditional form imposes very significant restrictions: the pH of the slaked-lime cement mix before it sets is high enough to degrade most glass fibers, limiting the strength of the final result. It also attacks most candidate thickeners and thixotropic additives, and so without a large amount of filler (normally quartz sand), lime cement tends to just turn into a puddle. If calcite objects could instead be created by remineralization of plaster of Paris, it would avoid these problems.

Even if the remineralization process required a large amount of other feedstocks relative to the gypsum, it could be very valuable:

small additions of plaster of Paris with ordinary portland cement have of course been tried, and while the result does set up quickly, it never achieves much strength. If this is due to gypsum in the structure that could be converted to the much stronger calcite in such an easy way, many new possibilities appear.

Topics

- Materials (p. 1138) (59 notes)
- Minerals (p. 1210) (6 notes)

Potential local sources and prices of refractory materials

Kragen Javier Sitaker, 02021-07-14 (updated 02021-09-11)
(9 minutes)

Geese Quimica SRL

Aqueous sodium silicate, unspecified concentration and modulus, goes for US$2/kg.

Bentonite (“bentonita sodica”) goes for US$0.30/kg (AR$1260/25kg). It’s the binder in greensand. Other vendors sell it in adulterated form as clumping cat litter for US$0.55/kg.

Ground coal (“carbon mineral molido para fundicion”) goes for US$0.70/kg (AR$2941/25kg).

They sell a product called “tierra de Junín” for aluminum casting for US$0.20/kg (AR$1346/40kg). I’m guessing this is a substitute for quartz sand; people say you need to add bentonite.

HIDRO-SAN

Low-temperature (1100°) kaowool-type ceramic fiber blanket, density unspecified, goes for US$0.60/ℓ (AR$1565/2.5cm/61cm/1m, which is 15 liters and US$9.60).

Fiberglass

Fiberglass is good to much lower temperatures but far cheaper at US$0.026/ℓ (AR$13583 for 3 × 1.2 mm × 18 m × 50 mm = 3240 ℓ for US$83). Typically it has trouble above 230°. “High density” is evidently 14kg/m³ and normal values are 7.7–12.1 kg/m³ so this is probably in that ballpark; if it were 10 kg/m³ that would be almost US$3/kg.

SAEMSA

Fiberfrax 1260° 96 kg/m³ ceramic fiber blanket costs US$0.40/ℓ (AR$7000 for 7200 mm × 610 mm × 25mm).

Solid firebrick (advertised as 1300°, 38% alumina) costs US$1.10/brick (23 cm × 11.5 cm × 6 cm, 3.6 kg).

Insulating 1480° firebricks (“K26”) cost US$3 (23 cm × 11.5 cm × 6 cm).

1000° refractory mortar costs US$0.80/kg, premixed.

1750° refractory mortar (SURECAST 80, 80% alumina) costs US$0.90/kg, premixed, but comes in a 30kg bucket.

AUKAN MINERA

Alumina in 100-μm grains (325 mesh) is US$3/kg.

Red clay is US$0.30/kg (“temp max 1180°”, which I doubt it reaches.)

Ball clay (?) is US$0.30/kg (AR$1240/25kg). 200 mesh, 1150° firing.

200-mesh quartz is US$0.30/kg (AR$1120/25kg). Quartz sand ("molido tipo azúcar o sal fina", so probably about 150 μm) is US$0.35/kg, while quartz flour (200-mesh, but "impalpabale"?) is US$0.27/kg.

200-mesh talc is US$0.40/kg (AR$1550/25kg). Talc is very interesting; see Firing talc (p. 576).

200-mesh calcite is US$0.20/kg (AR$790/25kg).

200-mesh potassium feldspar is US$0.30/kg (AR$1240/25kg).

325-mesh infusorial earth is US$0.40/kg (AR$1390/20kg).

Calcined metakaolin is US$0.35/kg (AR$1160/20kg). Uncalcined "super white" kaolin is US$1/kg.

Sparkly 30/80-mesh wet-milled mica is US$2.30/kg.

Workcrat

Alumina sold as abrasive (polvo abrasivo para pulir) is US$7/kg, regardless of whether it's 80-grit, 120-grit, or 220-grit. I'm *assuming* it's alumina; it's gray, so it's hard to tell.

JF Grafito

There's this guy in San Juan who sells graphite crucibles at prices from US$52 to US$58 to US$100, depending on the size. A different vendor they're good to 1600°, but I don't know if that's because the graphite is bonded with clay or something or because graphite burns too fast in air above that temperature.

Eiffel Química

These guys sell natural zeolite clay as an insecticide for US$7/kg; otherwise they mostly sell cosmetics supplies (ti tree oil, cetyl alcohol, castor oil, cocoa butter, bentonite, etc.)

Etc.

Zirconia abrasive wheels are widely available but I haven't been able to find zirconia in bulk.

Astonishingly, high-pressure sodium lucalox bulbs are also widely available; this 250W bulb costs US$12.

You can get construction sand for US$0.03/kg, which is pretty much pure quartz, alabaster for US$0.30/kg, and slaked lime for US$0.11/kg.

Apparently pumice is sold for US$0.14/ℓ under the brand name "Pometina" as an aggregate for lightweight concrete ("la roca natural mas liviana con un peso de 0,4 Kg / dm3 (menos que el agua)"), or at US$0.17/ℓ, which I guess would be US$0.42/kg, in 1-4 cm stones; they say it has no compressive strength, as a substitute for LECA. (At a slightly later date, it turns out the garden store down the street sells it for AR$370/10ℓ when US$1 = AR$180, thus US$2.10 or US$0.21/ℓ.) LECA itself (1200°?) is mostly sold for hydroponics these days, for US$0.29/ℓ or US$0.12/ℓ for construction, and is about three

times as dense; perlite (1150°) goes for US$0.38/ℓ at 0.128 kg/ℓ, so US$3/kg; rock wool (maybe 700–850°?) growth medium for US$0.31/ℓ at 0.1 kg/ℓ, so also US$3/kg; and vermiculite (1100°) for US$0.23/ℓ, at 0.06-0.16 kg/ℓ.

Unspecified broken-rock aggregate for concrete is more like US$0.08/kg, same price as ornamental 1–4 cm landscaping rocks. Even better, round 1–3 cm river rocks are only US$0.09/kg. However, these rocks probably don't have reliable refractory properties.

Gravel for fishtanks goes for US$0.19/kg (2–4 mm I guess) while gravel ("cascote") for concrete is US$0.016/ℓ, so probably US$0.008/kg, and construction gravel "6/20" (I think that means "6-20 mm") is US$0.03/ℓ, which is probably close to US$0.01/kg. Pure white marble pebbles (700°, say) is pricier at US$0.14/ℓ, which is at 1.25 kg/ℓ US$0.12/kg. Quartz gravel aggregate for concrete is sold as one type of "nonmetallic hardener"; for example, US$0.15/kg for an 8/20 grade (which I think is 8–20 mm); as ornamental stone another vendor sells it for US$0.021/kg, 6 tonnes minimum order.

Commercial castable insulating refractories sell for prices around US$1/kg.

[Other solid firebricks] go for prices depending on their alumina content. 23 cm × 8 cm × 11 cm bricks go for US$1.50 used, US$2 new if 45% alumina; US$2.50 new if 60% alumina; US$3.50 new or US$4.30 used (?) if 90% alumina.

This guy Ariel Weston is evidently marketing carborundum ("carburo de silicio") as "esmeril" at US$1.80/kg up to 220 mesh.

Phosphate is a crucial ingredient in many refractory cements. Previously I found that the cheapest way to buy it is as diammonium phosphate fertilizer, which is now US$1.00/kg and not very pure. By comparison, food-grade 85% phosphoric acid is US$3.40/kg; the pure substance is 98 g/mol of which 30.97 is phosphorus, so the liquid is thus 26.9% phosphorus by weight and US$13 per kg of phosphorus, while the fertilizer is nominally 132.06 g/mol and thus 23.4% phosphorus by weight and US$4.30 per kg of phosphorus. Either of the two might be more convenient, depending on circumstances. Trisodium phosphate goes for about US$2/kg.

Químico Cotton Fields sells boric acid for US$2.90/kg, but Planeta Verde sells it for US$1.70/kg; borax is sold by others for sealing ceramic crucibles for US$10/kg. Planeta Verde also sells aluminum sulfate for US$1.50/kg as a swimming pool clarifier; this is a potentially useful soluble aluminum salt for making either aluminum phosphate or foam of hydrated alumina. Many vendors also sell potassium alum, a soluble double sulfate of aluminum, for about US$3.50/kg, and MG Química also sells ammonium alum for US$8/kg, along with a bunch of food and cosmetics ingredients like borax, menthol, glycerin, turpentine, sodium lauryl sulfate, triethanolamine, sodium hyposulfite, silicone lubricant, lye, colophony, polyethylene glycol, carboxymethylcellulose, etc.

(Unrelatedly, Cotton Fields sells the pharmaceutical grade of the nontoxic wide-temperature-range coolant propylene glycol at US$6/kg, as well as triethanolamine, oxalic acid, naphthalene, isopropanol, sorbitol, povidone (polyvinylpyrrolidine), etc.)

Silica gel in particular goes for about US$5/kg. Other vendors sell it as silica cat litter for US$1.54/ℓ, which seems cheaper.

Someone is selling a lot of 5 carborundum globars for US$12 (brand: "Delta"). Someone else is selling what claims to be Kanthal A-1 for a variety of prices, such as US$0.72/m for 1-mm diameter, as well as 1-mm-diameter 80/20 nichrome for US$1.20/m. Thinner nichrome is naturally cheaper per meter; it's largely marketed for making and fixing *segelines*.

My efforts to find abrasives

welding blankets

zirconia

carborundum

alumina

Topics

- Materials (p. 1138) (59 notes)
- Pricing (p. 1147) (35 notes)
- Waterglass (p. 1189) (8 notes)
- Argentina (p. 1200) (7 notes)

Faygoo: a yantra-smashing ersatz version of Piumarta and Warth's COLA

Kragen Javier Sitaker, 02021-07-14 (updated 02021-12-30) (17 minutes)

I was rereading Piumarta and Warth's paper on open and extensible object models today (implementation), and I was thinking about how to extend it, which sadly seems not to have been done much, though Piumarta did build his COLA/Idst on it.

Piumarta & Warth

The paper describes a relentlessly simple metaobject protocol, with three core object types (object, symbol, and vtable) and five core methods (symbol.intern, vtable.lookup, vtable.addMethod, vtable.allocate, and vtable.delegated). The system starts with a symbol table, five symbols, and two vtables, one for vtables themselves and one for non-vtable objects. The fundamental operation it provides is send(anObject, aSymbol, args...). This is implemented by invoking bind(anObject, aSymbol), which in the usual case fetches the vtable pointer v from the word preceding anObject and then recursively invokes m = send(v, :lookup) on it to get the pointer to the method code. This having been done, the method code pointer m is then socked away in various caches for performances, and then invoked m(anObject, args...). To bottom out the recursion, bind() has a special case such that bind(vtablevtable, :lookup) returns a hardcoded constant function pointer.

If I'm reading the numbers right, they report that the usual C function call mechanism on their 2.16GHz Core 2 Duo takes about 8 ns, while this dynamic send mechanism takes about 15 ns in the fast path for monomorphic sends found on the fast path. This is a pretty reasonable cost; I think it replaces the single-instruction call found on most modern CPUs with 5-7 instructions. Deutsch & Schiffman report that in their measurements such a single-item inline cache was effective 95% of the time, though it is well known that the Self researchers got significant system speedups out of a polymorphic inline cache, since many of the remaining 5% of calls are not "megamorphic".

The other four core methods provide a runtime mechanism for constructing other object classes. intern creates new symbols that can be used as method selectors; x.addMethod(selector, code) mutates vtable x by overwriting or adding a method; x.allocate(size) instantiates a new object with vtable x; and x.delegated() creates a new vtable whose lookup method delegates to x when it doesn't find a method internally.

In more conventional terminology, vtables are called classes, allocate is called new, and delegates is called subclass.

The objective is that you should be able to extend the system with

new kinds of vtables that use a different lookup algorithm than the built-in version. They are only constrained in that whatever method pointer they return will be cached, both in an inline cache and in a system-wide cache indexed by (class, selector) tuples, forever.

The particular mechanism chosen for the inline cache allocates two words of writable memory for each callsite, one for the class and one for the method. Upon visiting the callsite a second time, if the receiver's class is equal to the class last time around, the chosen method is invoked without invoking bind(); otherwise, bind() is invoked for the new class, and the result is duly cached before invoking it. In pseudo-assembly:

```
    ;; callsite 4310
    ld.8 r1(-8), r2         ; r1 is the receiver, word size is 8
    ld.8 icclass.4310, r3   ; icclass.4310 is a statically allocated word
    bne r2, r3, 1f          ; if this is the wrong class, jump to slow path
    ld.8 icmeth.4310, r4    ; load method pointer
    b 2f
1:  st.8 r2, icclass.4310   ; update the IC key so we'll have a hit next time
    call bind               ; bind expects class in r2, method selector in r5
    st.8 r4, icmeth.4310    ; bind's return is in r4; r1 and r2 are preserved
2:  call *r4                ; method expects receiver in r1, class in r2
```

The fast path here is 6 of these 9 instructions, but it's common for absolute loads to require multiple machine instructions (auipc, loads from PC-relative constant pools, that kind of thing). Also, though, note that 9 instructions per callsite occupies quite a bit of code space. The slow path is mostly inside of bind, which may recursively invoke send() with lookup, and maintains the system-wide cache.

But what if we want to invalidate these caches to accommodate new methods, changes in the inheritance hierarchy, recompiled methods with new code, and so on? It's okay for the invalidation to be relatively expensive, since code changes happen much less often than message sends; but, with the above pseudocode, unless we want to add additional memory accesses and comparisons on the fast path, there are only two paths open to us:

- To *change the class of every affected object*, which would seem to involve scanning the entire heap and mutating potentially every live object, and probably also tripping the GC's write barrier on all those objects, possibly resulting in a second scan of the entire heap. This might actually be slower than just saving the whole system image and restoring it.
- To *scan through the entire IC space*, clearing the pointer in icclass.N for every N whose callsite might need to be relinked. This is surely less expensive; it might amount to a few million instructions for a large program.

Trampolines or yantras

In the original Deutsch & Schiffman JIT for Smalltalk-80, each method body was preceded by a header that validated that the receiver was of the correct class, failing over to a generic dispatch procedure if not:

The entry code [prologue] of an n-code [compiled] method checks the stored receiver class from the point of call against the actual receiver class. If they do not match, relinking must occur, just as if the call had not yet been linked.

This suggests the following alternative compilation of send():

```
    ld.8 icmeth.4310, r4
    call *r4
callsite.4310:
```

This transfers control to a trampoline, which has access to the receiver in r1 and the callsite in the link register. We'll call these trampolines "yantras", because they are a sort of apparatus the object system sets up to efficiently achieve the desired result. In the happy path, this yantra verifies that it is being invoked for the correct class, then transfers control to the method body:

```
    ld.8 r1(-8), r2
    li r3, $foobarclass            ; load immediate
    beq r2, r3, foobar::ogonkify
    b bind
```

There might be several such yantras for the same method, one for each class that inherits it, but the total number of yantras in the system is relatively small, perhaps tens of thousands. In the case that the inheritance hierarchy changes, or the method is replaced with a new version at a different address, or a subclass starts to override the method with its own version, we can simply smash the yantra by overwriting the first instruction or two with a `b bind` instruction.

The world is a little less rosy for `bind`. It has the receiver in r1 and the receiver's class in r2, but to finish its job it needs two more pieces of information: the selector of the method that was being invoked, and the address of the inline-cache variable `icmeth.4310` into which it must store the address of the correct yantra, which it may additionally need to synthesize. For these, it must look at the link register to find the callsite `callsite.4310`; then it can look up the callsite in a table of all callsites like the following:

```
callsites:
    .8byte callsite.0
    ; ...
    .8byte callsite.4308
    .8byte callsite.4309
    .8byte callsite.4310
    .8byte callsite.4311
    ; ...
```

Binary search on this table provides the index 4310 in about 12 memory accesses and about 60 instructions. (However, if code is not emitted in strictly increasing memory-address order, it might be necessary to use a segmented table, so maybe we should be using a 256-way trie instead.)

This index allows `bind` to look up the selector at index 4310 in a table of callsite selectors, so it can perform the correct lookup, and

also to store the yantra pointer into `icmeth.4310` for the next time around.

At system startup, the `icmeth.N` values for every callsite simply point directly at `bind`, which gradually materializes yantras for frequently-called methods and populates the global cache and the IC with pointers to them.

Ideally the yantras are allocated *after* the method bodies that they jump to so that their conditional branch will be correctly predicted the first time (and most of the times they fall out of the branch prediction buffer).

Let's call this inline-cache mechanism "faygoo", for "fast abstract yantra generic object orientation". Its primary benefit over Piumarta & Warth's implementation is that it provides a reliable, efficient, and fine-grained way of invalidating elements in the inline caches. But it might also be more efficient; on the fast path, faygoo runs 5 of these pseudo-assembly instructions instead of the 6 used by the Piumarta & Warth mechanism, and performs one data memory read instead of two. It uses much less space per callsite (two pseudo-assembly instructions, a read-only word, and a read-write word, rather than nine pseudo-assembly instructions and two read-write words) at the expense of allocating a megabyte or so of yantras, and perhaps having somewhat worse locality of reference.

Yantra size

Why a megabyte or so? In "bytecode interpreters for tiny computers" I dissected the Squeak 3.8-6665 image a little and found 49775 methods; presumably the majority of these are in leaf classes, while a few are inherited by thousands of classes, so perhaps in a system as large as Squeak, there would eventually be 65536-131072 yantras. Each of these consists of the four pseudo-instructions listed above; ld.8/li/beq/b.

I think that in the RISC-V C extension, the ld.8 would be C.LD if we store the class pointer at index 0 instead of index -1, the li is probably an uncompressed LUI/SLLI/LUI/ADDI sequence (14 bytes), the beq is probably an uncompressed BEQ (to avoid a separate subtract instruction and to be able to put the yantra within 4 KiB of the method start rather than 256 bytes like C.BEQZ), and the b is probably a compressed C.J to an uncompressed JAL instruction that's shared between many yantras. That's 20 bytes in all.

The amd64 realization might be something like this, 25 bytes per yantra if we don't do any alignment and don't use a shared fast-path second-level trampoline to allow the use of the shorter jump format, or 21 bytes if we do:

```
0000000000000000 <bind_shared_trampoline>:
   0:   e9 00 00 00 00          jmpq   5 <yantra1>

0000000000000005 <yantra1>:
   5:   48 8b 43 f8             mov    -0x8(%rbx),%rax
   9:   48 b9 ef cd ab 89 67    movabs $0x123456789abcdef,%rcx
  10:   45 23 01
  13:   48 39 c1                cmp    %rax,%rcx
```

```
  16:   0f 84 00 00 00 00       je     1c <foobar::ogonkify>
  1c:   eb e2                   jmp    0 <bind_shared_trampoline>
```

This also gets a byte shorter if we put the class pointer at offset 0 instead of offset -1:

```
000000000000001e <yantra2>:
  1e:   48 8b 03                mov    (%rbx),%rax
  21:   48 b9 67 45 23 01 ef    movabs $0x89abcdef01234567,%rcx
  28:   cd ab 89
  2b:   48 39 c1                cmp    %rax,%rcx
  2e:   0f 84 00 00 00 00       je     34 <foobar::ogonkify>
  34:   eb ca                   jmp    0 <bind_shared_trampoline>
```

In the i386 instruction encoding, conditional jumps could only use the short format, which would have required the extra shared unconditional jump.

If class identifiers were HotSpot-style 32-bit "compressed oops", or indices into some kind of object table or class table, instead of 64-bit memory addresses, these numbers would probably shrink a little. But probably each yantra costs about 20 bytes, so 65536–131072 of them will cost 1310720–2621440 bytes.

When to smash yantras?

One of the interesting items in Deutsch & Schiffman, carried over to most modern JITs, is the notion of a restricted-size cache for JIT-compiled code. Because they were using machines with small memory by current standards (typically using 16-bit address spaces, and usually less than a megabyte) and slow disks, and the Smalltalk bytecode ("v-code") was something like 5× more compact than the native machine code ("n-code"), they considered it more practical to discard the least-recently-used native code rather than swapping it out to disk. Nowadays, rather than main memory, the objective might be to fit the n-code into L1 or at least L2 cache (4096 KiB on this AMD A10-5745M). If a compiled method is to be thus discarded, smashing its yantras is a cheap way to ensure that control will not flow into it again.

But that's just an optimization. The main reason to smash yantras is because the method they jump to is no longer the correct method for the class; in this case the yantra must be smashed and its corresponding global lookup cache entry must be removed. If your only metaclass (vtable vtable) is the primitive metaclass, and you only modify classes through Piumarta & Warth's methods, this happens only when you call `addMethod`, at which point we must smash that method's yantras in that class and all its subclasses. This should be straightforward.

However, the great appeal of the OEOM metaobject protocol is that it makes it straightforward to extend the system with other metaclasses. Piumarta & Warth give the example of implementing Traits, but other possibilities include RPC remote stub objects, the proxy-membrane pattern more generally (which is also useful for auditing or debugging), record-reply-style creation of mock objects

for unit tests, and the creation of data classes from, for example, Proto3 files.

Some of these cases, like dynamic data-class creation, don't actually need custom metaclasses; being able to subclass existing classes and add methods to them under program control is quite sufficient. Other cases require the creation of new methods on demand, but never subclass the classes they create or change those methods once they exist, so never need to smash any yantras. However, in theory, the ability for the lookup method to examine arbitrary mutable data means that any change in the system state, or even outside of it (such as on a network server with which the lookup method communicates) might necessitate smashing yantras.

The most significant missing facility from the methods in the primitive metaclass is the ability to mutate the inheritance graph by changing a class's parent, which would necessitate smashing some or all of the yantras of that class and all its transitive subclasses, depending on whether the methods they inherited had changed.

For *interactive* changes to the system, such efficient and fine-grained cache invalidation is probably not necessary. If there are 65536 methods with 131072 yantras and 1048576 callsites, flushing the entire cache can be done by brute force by overwriting only 8 mebibytes of RAM, which takes on the order of a millisecond; repopulating the cache as callsites get rebound will slow down the system over a longer period but still possibly be a tolerable slowdown. But you wouldn't want the system to hang for a millisecond every time a new method got invoked on a debugging membrane proxy object.

Eliminating inheritance

I think the OEOM system becomes simpler if you eliminate inheritance entirely. There's no need to propagate invalidation through subclasses and no need for class.subclass() (called vtable.delegated() by Piumarta & Warth). There would still be the possibility that a changed lookup() method might return different results, and the only way to be sure to avoid this is to flush the cache completely.

Topics

- Performance (p. 1155) (22 notes)
- Assembly-language programming (p. 1175) (11 notes)
- Compilers (p. 1178) (10 notes)
- Small is beautiful (p. 1190) (8 notes)
- Higher order programming (p. 1196) (7 notes)
- Systems architecture (p. 1205) (6 notes)
- Dynamic dispatch (p. 1259) (4 notes)
- Jit

Firing talc

Kragen Javier Sitaker, 02021-07-14 (updated 02021-12-30)
(17 minutes)

Talc ($Mg_3Si_4O_{10}(OH)_2$) is very interesting, particularly for cyclic fabrication systems, because it calcines to enstatite and amorphous silica at 800–840°, and the enstatite ($MgSiO_3$) converts to clinoenstatite at 1200° (?), and the amorphous silica to cristobalite at 1300° (or possibly it converts to enstatite above 1050°?); enstatite melts at some 1550° and quartz/cristobalite at 1713°. You could think of enstatite as being a quartz/forsterite eutectic, and indeed 1543° is the lowest-melting point in the quartz/forsterite continuum, but the cristobalite crystals remain solid until 1713°. (I'm not sure how solid the resulting slush is, though...)

Potential for inorganic cyclic fabrication systems

If you have an ample supply of talc, then, as with clay, you can shape it into the geometry you want, and then fire it to harden it. As with clay, you can use tools thus made to shape new objects. However, firing talc to harden it does not (?) cause it to become plastic and change in shape or shrink; the results are nonporous and extremely unreactive; and they are capable of withstanding much higher temperatures than those needed to fire them, so it should be easy to make talc-firing ovens out of fired talc. The required firing temperature is lower than for most clays, and much lower than for high-performance porcelain clays.

Because the unfired talc is not plastic and does not become plastic during firing, very large shapes can be precisely cut and fired in a gravitational field without suffering plastic deformation. Though brittle, the fired objects are highly resistant to thermal shock, in part because of their low thermal coefficient of expansion (enstatite's linear TCE is a steel-like 11 ppm/K), permitting their use for a wide variety of applications with demanding temperature gradient requirements.

Talc is a useful feedstock for synthesizing cordierite, a common refractory with an acicular crystal habit and even lower TCE, which can be as low as 0.8 ppm/K for porous cordierite materials.

For machinery involving sliding contact, unfired talc serves as a dry lubricant.

Obtaining adequate supplies of natural talc may be difficult, but straightforward hydrothermal talc synthesis processes are known, and some of them can be carried out in vessels made of fired talc. The raw feedstocks are abundant: oxygen is 47% of the Earth's crust, silicon 28%, and magnesium 2.1%. Only hydrogen is scarce (0.14%), and it is removed in the firing process.

Fired talc is hard enough to grind or cut softer metals, but I think it can only barely cut steel, if that.

Even fired, talc is not a very good material for withstanding impact, controlling electricity, or spring energy storage.

History of firing talc

An anonymous stone sculptor writes that they have been firing soapstone as a stone-sculpting technique for many years:

> Many years ago I wrote an article about firing soapstone to make the stone harder. It is possible to harden the stone to the point that you cannot scratch it with anything short of steel. This will also close the structure of the stone enough to allow it to be used in outdoor applications. ... Insulators on the electric poles were at one time made of fired soapstone.

A geology.com article says the Americans have been carving soapstone for 8000 years, although I'm not clear on whether the results were fired — but because cooking bowls, cooking slabs, and smoking pipes were among the uses, I suspect they were. The same article mentions the Scandinavian Bronze-Age use of soapstone for carving bronze-casting molds. (Also, it mentions that some "soapstone carvings" are actually alabaster or serpentine.)

Reportedly potters use 50% talc, 50% ball clay for slipcasting.

Material qualities

In "Synthesis of $MgSiO_3$ ceramics using natural desert sand as SiO2 source" Wang et al. mention some pretty cool benefits of enstatite-based ceramics, including protoenstatite's "adjustable coefficient of thermal expansion (CTE), high strength, low dielectric loss, and strong vibration resistance" at high temperatures, but lament its tendency to crack when transforming to clinoenstatite at lower temperatures. Evidently this is not a problem with fired-talc ceramics, though.

Firing synthetic magnesium silicate

Apparently if you react soluble magnesium and silicate salts, at low enough Mg/Si ratios you get crystalline talc but only above 200°. But, according to the same paper, it looks like amorphous synthetic magnesium silicate hydrates also produce silica when fired in the same way, but at a slightly lower temperature:

> Differential thermal analysis (DTA) of the M-S-H samples reveals a distinct exothermic transition around 840 - 860 °C, which is attributed to the decomposition of amorphous M-S-H and the recrystallization to SiO_2 and enstatite ($MgSiO_3$) [11, 12] or SiO_2 and forsterite (Mg_2SiO_4) for high Mg/Si ratios [29]. The dehydroxylation of the phyllosilicate sheets in talc ($Mg_3Si_4O_{10}(OH)_2$) and the transformation to SiO_2 and enstatite occurs at slightly higher temperatures of 850 to 950 °C [11, 12]. The lower dehydroxylation temperature of M-S-H compared to talc is consistent with a less crystalline structure [11, 12].

(It's worth noting that their room-temperature synthesis took a year, but only because they were using low-solubility feedstocks.)

And I've prepared what seems to be this amorphous material in my kitchen by dumping magnesium chloride from the health food store into a can of waterglass (see Glass foam (p. 595) for details). However, it came out in the state of sort of slushy lumps; I'm not sure how to stick it together into a more solid material. Maybe with more waterglass, I don't know.

Links to follow up

https://usenaturalstone.org/soft-spot-soapstone/

https://doradosoapstone.com/blog/6-fun-and-historical-facts-about-soapstone/ notes on firing

https://www.golcha.com/insulators.html talc and clay electrical insulators

https://wmblogs.wm.edu/sgtresearch/the-soapstone-special/ mining in Virginia

https://en.wikipedia.org/wiki/Insulator_%28electricity%29#Insulation_of_antennas steatite mountings

https://www.jpcfrance.eu/technical-informations/electrical/historical-introduction-of-ceramics-used-in-connection-blocks/ notes on historical electrothermal applications

https://isolantite.com/ “Isolantite has been a major producer of precision Steatite Ceramic electrical components for over 100 years”

https://www.r-infinity.com/Companies/ history of insulator companies

https://www.lspceramics.com/steatite-ceramic-insulators/ “We can grind steatite ceramic insulators to your specs, achieving tolerances within 0.0005”.”

https://www.indiamart.com/proddetail/anoop-steatite-and-alumina-ceramic-insulators-1460976162.html “used in all types of communication devices operating at a very high voltage and high frequency”

https://www.sciencedirect.com/science/article/pii/S027288421930906X “Synthesis of $MgSiO_3$ ceramics using natural desert sand as SiO_2 source”

https://www.sciencedirect.com/science/article/pii/S0272884220600227 “Synthesis and characterization of $MgSiO_3$-containing glass-ceramics”

https://www.tandfonline.com/doi/abs/10.1179/174367606X128423 “Microstructure, mechanical and thermal properties of boron oxide added steatite ceramics”

https://www.tandfonline.com/doi/abs/10.1080/0371750X.2019.1657954 “Production and Characterization of Alumina and Steatite Based Ceramic Insulators”

https://www.sciencedirect.com/science/article/pii/S0272884206000861 “Low-temperature fabrication of steatite ceramics with boron oxide addition”

Steatite ceramics have been fabricated by using coarse starting materials such as talc, clay, and barium carbonate with addition of boron oxide (B_2O_3). B_2O_3 has been found to be a useful flux to densify $MgO–Al_2O_3–SiO_2–BaO$ (MASB) powders. The steatite ceramic with a relative density of 97% was obtained at a sintering temperature of 1200 °C.

https://www.sciencedirect.com/science/article/pii/S02728842183290262 “Effects of mechanical-activation and TiO_2 addition on the behavior of two-step sintered steatite ceramics”

Steatite, as ceramic with composition predominantly resting on magnesium silicate, was produced from economic resources – talc, aluminosilicate clays, and either $BaCO_3$ or feldspar as flux.

https://www.degruyter.com/document/doi/10.3139/146.110409/html "The role of talc in preparing steatite slurries suitable for spray-drying"

https://link.springer.com/article/10.1007%2FBF01285830 “Sintering corundum with additives”, 01964, Ukraine Scientific-Research Institute of Refractories, USSR

https://ieeexplore.ieee.org/abstract/document/5781588 “Notice of Retraction: Synthesizing Cordierite from High-Alumina Fly Ash and Talc Powder”

“Continuous synthesis of nanominerals in supercritical water”

https://www.degruyter.com/document/doi/10.1515/chem-2020-01504/html “Synthesis of magnesium carbonate hydrate from natural talc” cc-by. kind of backwards from what I’m looking for tho…

https://www.nature.com/articles/srep22163 “Rapid growth of mineral deposits at artificial seafloor hydrothermal vents” mostly sulfide with a little talc

https://link.springer.com/article/10.1007/s00410-009-0395-4 “Growth and deformation mechanisms of talc along a natural fault: a micro/nanostructural investigation”

https://pubmed.ncbi.nlm.nih.gov/28771845/ “Synthetic Talc and Talc-Like Structures: Preparation, Features and Applications”

This contribution gives a comprehensive review about the progress in preparation methods, properties and applications of the different synthetic talc types: i) crystalline nanotalc synthesized by hydrothermal treatment; ii) amorphous and/or short-range order nanotalc obtained by precipitation, and iii) organic-inorganic hybrid talc-like structures obtained through a sol-gel process or a chemical grafting. Several advantages of nanotalc such as high chemical purity, high surface area, tunable submicronic size, high thermal stability, and hydrophilic character (leading to be the first fluid mineral) are emphasized

https://www.nature.com/articles/ncomms10150 “Talc-dominated seafloor deposits reveal a new class of hydrothermal system”

https://www.chemistryviews.org/details/ezine/9489511/Talc_from_the_Lab.html “In the journal Angewandte Chemie, French scientists introduce a laboratory synthesis of talc, which needs only seconds and produces nanocrystals with unique properties useful in many applications … Two research groups under the direction of Cyril Aymonier and François Martin in Bordeaux and Toulouse, France, have developed a method to produce synthetic talc nanocrystals hydrothermally in supercritical water.”

https://pubs.rsc.org/en/content/articlelanding/2010/jm/c0jm01276a#!divAbstract “Functionalization of synthetic talc-like phyllosilicates

by alkoxyorganosilane grafting"

A range of talc-like phyllosilicates were prepared via a hydrothermal synthesis performed at five different temperatures from 160 to 350 °C. The organization of the lattice and the degree of crystallinity of the new materials were evaluated by different techniques such as XRD, FTIR, solid-state 29Si NMR, TEM, FEG-SEM and TG-DTA. When synthesized at low temperature the material presents high degree of hydration, low crystallinity and flawed structure. This was attributed to stevensite-talc interstratified product present in the samples. The stevensite/talc ratio and the hydration decrease in the talc-like phyllosilicate samples when the hydrothermal synthesis temperature increases and so the crystallinity becomes higher. A thermal treatment at 500 °C allowed a significant flaw reduction in talc-like phyllosilicate structure; the synthesized sample at 350 °C and heat treated presents a structure close to that of talc.

https://www.sciencedirect.com/science/article/abs/pii/S016913171300003001 "Synthetic talc advances: Coming closer to nature, added value, and industrial requirements"

Over the past 2 years, the synthetic process of talc particles has evolved considerably, leading to an inexpensive, convenient, and rapid process that is compatible with industrial requirements. In addition to facilitate the synthetic talc preparation, the evolution of the synthesis process has led to an improved crystallographic arrangement of the talc particles in both the c* direction and (ab) plane. In the present study, the most recent process was investigated with respect to the reaction time, temperature, pressure, pH, and salt concentration to determine the optimal reaction parameters.

https://egyptmanchester.wordpress.com/2018/01/19/the-use-of-steaotite-in-ancient-egypt/

In its raw state the softness of steatite make it extremely easily damaged, and simply wearing or using a carved object would damage the carved detail. Steatite has an interesting property, when it is fired it will convert from steatite into enstatite. Unlike steatite, enstatite has a hardness of Mohs 5.5 which is close to that of granite – making it extremely hard wearing and resistant to damage, whilst still retaining its carved detail. Steatite has also been glazed since the Predynastic era for objects such as beads and amulets. ... Firing at a temperature of ~950°C will cause steatite to dehydrate and crystallise into enstatite. Clay will begin its vitrification process ~800-900°C and firing will generally require temperatures in excess of 1100°C , therefore the steatite to enstatite conversion can be achieved using similar technology as is required for firing clay objects. A wood fuelled open fire can easily reach temperatures exceeding 1100°C, and can be used for firing ceramics and also for converting steatite to enstatite.

Further reading: Connor, S, Tavier, H and De Putter, T. 'Put the Statues in the Oven: Preliminary Results of Research on Steatite Sculpture from the Late Middle Kingdom'. Journal of Egyptian Archaeology 101 (2015).

https://www.hindawi.com/journals/geofluids/2017/3942826/ "An Experimental Study of the Formation of Talc through $CaMg(CO_3)_2$–SiO_2–H_2O Interaction at 100–200°C and Vapor-Saturation Pressures"

In this study, in situ Raman spectroscopy, quenched scanning electron microscopy, micro-X-ray diffraction, and thermodynamic calculations were used to explore the interplay between dolomite and silica-rich fluids at relatively low temperatures in fused silica tubes. Results showed that talc formed at ≤200°C and low CO_2 partial pressures (PCO_2). The reaction rate increased with increasing temperature and decreased with increasing PCO_2. The major contributions of this study are as follows: we confirmed the formation mechanism of Mg-carbonate-hosted talc deposits and proved that talc can form at ≤200°C; the presence of talc in carbonate reservoirs can indicate the activity of silica-rich hydrothermal fluids; and (3) the reactivity and solubility of silica require further consideration, when a fused silica tube is used as the reactor in high P–T experiments.

https://www.sciencedirect.com/science/article/abs/pii/S01691317150001532 “Technological properties of ceramic produced from steatite (soapstone) residues–kaolinite clay ceramic composites”

Ceramic bodies (7.0 cm × 2.0 cm × 1.0 cm) of kaolinite clay and soapstone residuals collected from workshops in Ouro Preto and Mariana, Minas Gerais, Brazil, containing from 2.5 to 97.5 wt% steatite (soapstone) were prepared and firing at 500, 1000 and 1200 °C, for 2 h, in air. The linear shrinkage, compressive strength, water absorption and mass loss by heating were determined on the samples after heat treatment. The fired samples at 1000 and 1200 °C, with steatite percentages of 85, 90 and 95%, presented the best results for technological applications in ceramic industry. For these samples, the values of the compressive strength were higher than 10 MPa and those of water absorption varied between 8 and 22%, which means that the values of these properties are superior and inferior, respectively, to the reference values established by Brazilian Standards. The linear shrinkage was lower than 6%, which is the maximum value established by the Pólo Cerâmico de Santa Gertrudes, in São Paulo State.

https://experimental-prehistory.blogspot.com/2016/02/soapstone-steatite-as-hard-as-flint.html “Making Soapstone/Steatite as hard as flint ? after firing, at ~950°C?”

https://www.mdpi.com/2075-163X/8/5/200/htm “Looking Like Gold: Chlorite and Talc Transformation in the Golden Slip Ware Production (Swat Valley, North-Western Pakistan)”

...In order to constrain the firing production technology, laboratory replicas were produced using a locally collected clay and coating them with ground chlorite-talc schist. On the basis of the mineralogical association observed in both the slip and the ceramic paste and the thermodynamic stability of the pristine mineral phases, the golden slip pottery underwent firing under oxidising conditions in the temperature interval between 800 °C and 850 °C. The golden and shining looks of the slip were here interpreted as the result of the combined light reflectance of the platy structure of the talc-based coating and the uniform, bright red colour of the oxidized ceramic background.

https://digitalfire.com/material/talc

http://webmineral.com/data/Clinoenstatite.shtml#.YO9psLOdJhE

https://www.mindat.org/min-1072.html clinoenstatite

https://www.sciencedirect.com/science/article/pii/S218707641500060052 “Talc-based cementitious products: Effect of talc calcination” cc-by-nc-nd

This study reports the use of calcined talc for cementitious products making. The calcination is used to enhance the availability of magnesium from talc to react with phosphate for cement phase formation. It is shown that previous calcination of talc leads to products having enhanced mechanical performance due to the formation of more cement phase than in products based on raw talc. Talc fired at 900 °C was found to be the one in which magnesium release was maximal. Firing at temperature higher than 900 °C leads to the stabilization of enstatite, which decreased the magnesium availability. The cement phase is struvite, which was better detected on the X-ray patterns of the products involving fired talc. All the products have very rapid setting time and low shrinkage.

https://www3.epa.gov/ttnchie1/ap42/ch11/final/c11s26.pdf talc processing

https://nvlpubs.nist.gov/nistpubs/jres/15/jresv15n5p551_A1b.pdf

https://rruff.info/doclib/hom/clinoenstatite.pdf

http://deanpresnall.org/files/90Enst.melting.pdf

http://www.geo.umass.edu/courses/geo321/Lecture%208%20Binary%20Systems.pdf

https://hal.archives-ouvertes.fr/hal-01686526/file/ACL-2018-009.pdf

https://www.researchgate.net/profile/Adnan-Badwan/publication/223981891_Magnesium_Silicate/links/5a5b07320f7e9b5fb388e3bb/Magnesium-Silicate.pdf

https://www.researchgate.net/profile/Kamal-Tabit/publication/327635588_Crystallization_behavior_and_properties_of_cordierite_synthesized_by_sol-gel_technique_and_hydrothermal_treatment/links/5e525d6492851c7f7f504015/Crystallization-behavior-and-properties-of-cordierite-synthesized-by-sol-gel-technique-and-hydrothermal-treatment.pdf

https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.643.1922&rep=rep1&type=pdf

http://www.doiserbia.nb.rs/img/doi/0354-4656/2018/0354-46561803297R.pdf

http://75.72.27.96/dad/Journal%20of%20the%20European%20Ceramic%20Society/2004%20%28Vol%2024%29/Volume%2024%2C%20Issues%2015-16%2C%20Pages%203693-3848%20%28December%202004%29/3817.pdf

https://dais.sanu.ac.rs/bitstream/handle/123456789/4536/Terzic_Science-of-Sintering_50_2018_299-312.pdf?sequence=1&isAllowed=y

http://przyrbwn.icm.edu.pl/APP/PDF/127/a127z4p077.pdf

https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.537.1305&rep=rep1&type=pdf

https://www.gtcountymi.gov/AgendaCenter/ViewFile/Item/1198?fileID=5819

https://www.repositorio.ufop.br/bitstream/123456789/8834/1/ARTIGO_TechnologicalPropertiesCeramic.pdf

https://www.ceramics-silikaty.cz/2015/pdf/2015_04_331.pdf

Topics

- Materials (p. 1138) (59 notes)
- Experiment report (p. 1162) (14 notes)
- Ceramic (p. 1193) (8 notes)

- Self replication (p. 1204) (6 notes)
- Minerals (p. 1210) (6 notes)
- Talc (p. 1319) (2 notes)
- Forsterite
- Enstatite

Improvements on C for low-level programming such as block arguments

Kragen Javier Sitaker, 02021-07-14 (updated 02021-12-30) (8 minutes)

C is simple and efficient, doesn't require GC, and potentially allows you to handle failures, but it can take a lot of code to get things done, and you tend to bake in a lot of inflexibility. Things like Smalltalk are less efficient and can run out of heap, but require less code to get stuff done, and allow more flexibility. I've been thinking about how to get a Pareto improvement over C: something that allows you to handle failures, but still supports failure-free GC-less computing, and is simpler. Seven key features I've identified are coercions, dynamic dispatch, call-by-name, block arguments, pattern matching on sum types, uninitialized data, and list comprehensions.

C tries to help you by implicitly converting between data types, but the rules for this are extremely complex and bug-prone. I think it's better to require explicit coercions but, as in Golang, make constants sort of typeless. This makes the language safer and much simpler in exchange for, generally, a tiny increment of brevity.

C++ can do dynamic dispatch with virtual functions, but the design is all wrong. The Golang design for virtual functions by indirection through an interface is the correct design; whether a method call is statically dispatched or dynamically dispatched is a property of the callsite, not the function being called. This is achieved by (implicitly, in Golang) creating an "interface object" containing a pointer to the object and the relevant function pointers. In the case where you're creating an interface object for a concrete type, or for a more restricted facet of a broader interface, this can be done statically. Golang also, using dynamic checks, implements a broadening type conversion; I think this is basically just a kludge to fake dynamic typing, and can be dispensed with.

The Golang interface construct doesn't create a subtyping relation. The interface is a separate object from the concrete object that it provides access to; they are separate types, although Golang does unfortunately have implicit coercions. The absence of subtyping relations dramatically simplifies the task of type inference.

Call-by-name was a much-maligned feature of the original Algol 60, omitted from both C and Pascal. It allowed you to write a function like the following:

```
double sum(double &fi, int &mut i, int start, int stop) {
  double total = 0.0;
  i = start;
  while (i != stop) { total += fi; i++; }
  return total;
}
```

This could be invoked, for example, as `sum(&a[i], &mut i, 0, 10)` or as `sum(&i**2, &mut i, -5, 6)`, assuming `**` is exponentiation. As with C preprocessor macros, each evaluation of the name of a call-by-name parameter would re-evaluate the expression that had been passed as an argument, and this even worked in lvalue context, so `i = start;` here would change the value *in the caller*. In these cases, changing `i` also changed `fi`.

This is clearly somewhat bug-prone (though I think some kind of marking at the callsite like the `&` in the above might help to clarify what's going on) and its implementation is slow (it involves creating one or more closures per parameter; they are not garbage-collected, but invoking them involves all the usual call/return overhead); but it is not without its merits. One is that, unlike C pointer parameters, it can be passed down the stack, but cannot be stored in a data structure or returned up the stack, so they do not pose a risk of pointer lifetime bugs. The other, in combination with CLU iterators, is subtler and more powerful.

The block arguments I have in mind are similar to CLU iterators; they are a restricted form of the block arguments you see in Smalltalk or Ruby. You can use them as iterators:

```
each_line(&mut line, somefile) { print(line); }
```

Or as resource managers:

```
with_file(&mut f, filename, "r") { results = read(f, fstat(f).st_size); }
```

Or as conditionals:

```
button("Invert") { invert(&mut image); }
```

The block is passed in to the function being invoked as a closure, just like a name parameter. That function can invoke the block (in CLU or Ruby, `yield`) zero or more times, including with arguments. But the block does not have its own stack frame; instead, it stores all its state in the subroutine it's lexically nested inside of. And the subroutine they're passed to cannot return them or save them in a data structure.

Together with call-by-name, this facility allows you to implement control structures as library functions:

```
forto(&mut x, 0, 10) { printf("%d² = %d\n", x, x**2); }
while (&x < 10) { x++; }
```

You *could* implement `while` something like this, at least as a prototype, assuming proper tail-call elimination is at play:

```
void while(&b) {
    if (!b) return;
    yield;
    while(&b) { yield; }
}
```

Syntactically you probably need facilities to handle multiple block arguments, though; in particular you want to be able to do things like map and filter as such iterators, which take one block to describe the mapping or filtering and a second block to send the result to.

Null pointers are the bane of C, and ML-family languages avoid them by instantiating their data structures fully populated and by using variant tags. If you define a string-search-result type in OCaml:

```
type loc = Found of int | Not_found
```

then if you have a value of type `loc` you cannot get any data out of it without a pattern-match:

```
match p with Found x -> x * 2 | Not_found -> 1
```

The variable `x` is only in scope in the part of the conditional where it has matched something. This eliminates the need for NULL.

(I'm not sure how this fits into C's nested-data model, but I don't think there's a clash.)

Uninitialized data in C is almost invariably a result of the traditional Algol block structure with declarations at the top of the block, followed by statements. C at least allows you to initialize things in the declarations. But there's very little reason, even in modern C, to declare a variable before giving it a value; you can delay declaring it until you have the value ready. This eliminates another source of null data.

In Python, list comprehensions are just a ridiculously useful way to write code. The translation to imperative code is straightforward, so why should we do it by hand? (This might involve some tricky questions about allocation.)

The thing I was really trying to figure out here, though, is how to put a static stack size bound on code that uses these features, in particular block arguments and dynamic dispatch. Dynamic dispatch with method names is potentially less troublesome than arbitrary function pointers, since a function pointer could take you to any function whose address has been taken, while method dispatch can only take you to one of the 6 functions named toTuple or whatever. Block arguments are a little tricky because any calls from within the block get stacked on top of the activation record of the "iterator" that's calling it --- but not at the same time as the iterator's own callees.

There are two different reasons for wanting a static stack size bound. One is to statically guarantee that the program won't crash. Another is to insert a minimal number of stack overflow checks which, if they fail, do something like invoke the Cheney-on-the-MTA trampoline, or allocate a new segment to expand the stack into.

Topics

- Programming (p. 1141) (49 notes)
- Safe programming languages (p. 1172) (11 notes)
- C (p. 1194) (8 notes)
- Higher order programming (p. 1196) (7 notes)
- Pascal (p. 1247) (4 notes)
- OCaml (p. 1249) (4 notes)
- Dynamic dispatch (p. 1259) (4 notes)
- Memory models (p. 1285) (3 notes)
- Smalltalk (p. 1326) (2 notes)
- Call by name (p. 1382) (2 notes)
- Block arguments (p. 1383) (2 notes)
- Typing
- Golang
- Failure free

Fiberglass CMCs?

Kragen Javier Sitaker, 02021-07-15 (updated 02021-07-27)
(8 minutes)

Fiberglass for insulation is super cheap (Potential local sources and prices of refractory materials (p. 566) reports that it's around US$3/kg) and has substantial tensile strength, even if it's not as high as S-glass and M-glass. Chopped glass fiber for mixing into plastics as reinforcement (probably E-glass) also sells for about US$3/kg (AR$9975 for 20kg). (I think fiberglass insulation is A-glass.) Glass fibers are commonly used to reinforce organic polymers like epoxy to make them roughly as strong as steel, but this matrix is itself fairly expensive in bulk; 6 kg of epoxy from Tienda Bepox costs AR$21000 , US$129, or US$21/kg, seven times the price per unit mass of the fiber reinforcement. Polyester resin is significantly cheaper, at AR$7300/10kg, US$4.50/kg, but also much more rigid and, I think, more fragile. (Vinyl ester and phenolic casting resins are apparently harder to find, locally at least.)

For polyester thermosets, Matweb gives useless ranges of 10–123 MPa for ultimate tensile strength and 54–265 MPa for flexural yield strength (implying that at least one polyester thermoset has a flexural yield strength twice its ultimate tensile strength), and for "epoxy cure resin" 5–97 MPa and 76–1900 MPa, and for "epoxy, cast, unreinforced" 8–97 MPa and 14–131 MPa. And I have no idea where in these very wide ranges these resins are.

Matweb says fused quartz (Saint-Gobain Quartzel) is 6000 MPa UTS and generic A-glass fiber is 3310 MPa UTS, while Micarta RT500M glass-reinforced epoxy is only 269 MPa UTS and "Goodfellow E-glass/Epoxy composite" is only 490 MPa UTS. So there's a lot of room for improvement.

Could you make a ceramic-matrix composite from insulation fiberglass, getting the usual cracking-resistance performance improvements of CMCs, even if the improved performance doesn't approach the kind of performance you get out of things like SiC/SiC CMCs? Could you do it *cheaply*?

You clearly can't heat it up to the high temperatures normally associated with CMC manufacturing; those are over 1000° and glass wool craps out around 230–260° (reportedly, I haven't tried, and I suspect this may just be the max temp of the polymeric sizing — soda-lime glass normally doesn't soften until around 700°), while rock wool (which is also about US$3/kg) should maybe be good to 700°–850°. So you might need some kind of hydrothermal process. Moreover, the hydrothermal process needs to be gentle enough to not eat the glass reinforcing fibers, so in particular ordinary portland cement and carbonation of slaked lime are probably off the table, and so is anything that involves making silica highly soluble in water.

A few candidate matrix systems:

- Calcined alabaster, of course, though this will never reach very high

strengths.

• Geopolymer cements with low alkalinity, such as milled metakaolin polymerized with modulus-4 sodium silicate (though Davidovits recommends 1.45–1.85).
• Low-alkalinity waterglass binders.
• Aqueous phosphate cements such as the phosphates of zinc, magnesium, or aluminum.
• Sorel cement (pH 8.5 to 9.5 according to Wikipedia).

To any of these binders you could add functional fillers such as clays, talc, mica, quartz sand, quartz flour, carborundum, perlite, vermiculite, glass foam, or sapphire.

Aside from the spatula layup approaches normally used to produce fiber-reinforced plastics, chopped-fiber mixing is a possibility, and spin-coating of the binder/filler/fiber mixture would tend to produce a biaxial orientation of the solids, which would tend to be very advantageous to both solids density and to applying the strength of the solids in the appropriate directions. This approach would be best at increasing the density of the solids if the binder decreases greatly in volume after the spinning, for example by dehydrating.

Another way to increase the toughness and flexibility of such a stiff composite material is to fabricate it in thin sheets and then laminate them together with a softer or weaker binder. The binder would tend to stop crack propagation or at least redirect it parallel to the surface of the material, and if it is soft rather than just weak it would tend to shear to allow the overall composite sheet to flex, like the pages of a book. Sodium chloride and highly-hydrated sodium silicate are two candidate binders that might be capable of such shear, especially over time.

Sizing

Normally when you're making glass-fiber-reinforced things you're concerned with sizing the fibers to improve adhesion to the matrix, because otherwise the low-modulus, low-strength matrix can't effectively transfer load to the fibers, and you get pullout failures at much lower loads than would be needed to actually break the fibers.

In CMCs the objective is the opposite: the matrix has strength comparable to the fibers, but both the fiber and matrix have the very low elongation at break characteristic of ceramics. Instead, the fibers are incorporated to arrest crack growth by bridging cracks, for which a much longer length of fiber must be recruited to stretch, for which the fiber needs to slide through the matrix. So sizing is necessary to *weaken* the bond between the fiber and the matrix, *causing* material failures to be pullout failures — still at a lower stress than would be needed to break a solid, defect-free block of the material, but with enormously improved toughness. This can even translate to higher tensile and flexural strength in the case of non-defect-free brittle materials, where microcracks can reduce their theoretical tensile strength by many orders of magnitude.

So, what kind of sizing could you apply to the glass fibers to weaken their bond to the matrix? You want it to be much softer than the other materials involved, and to completely cover the glass fibers, or at least almost so, which suggests that you'd like it to be

amorphous. You want it to be water-insoluble so it will survive when the matrix is infiltrated into the fibers, but applicable hydrothermally rather than with some kind of white-hot gas or something. And you want it to be cheap, which eliminates most organics. Precipitating amorphous calcium phosphate or aluminum trihydroxide from aqueous solution might work; there are also a number of softer insoluble phosphates and carbonates of polyvalent cations, such as rhodochrosite, siderite, vivianite, wolfeite, smithsonite, hopeite, malachite, azurite, magnesite, struvite, and calcite. Oxide and hydroxide minerals might also be candidates.

Amorphous deposition isn't essential, but if we hydrothermally deposit a crystalline sizing material, there's a good chance there will be gaps between the crystals that leave the glass exposed.

Another approach would be to expose the fibers to something that reacts enthusiastically with the exposed glass (for which there are relatively few candidates at ordinary temperatures, mostly silanes) and forms a monolayer on it, and then something else that reacts with the monolayer, perhaps repeating the process several times.

Infiltrating loose glass fibers very briefly with a low-density hot plasma of something that's solid at room temperature might work to deposit a thin, even layer on the surface of the glass without damaging it, if the heat in the plasma isn't sufficient to melt the glass. The layer would tend to be of relatively constant thickness because the places on the glass where the plasma has already frozen will be hotter than the places it hasn't frozen yet. For example, tenorite and maybe cuprite (Mohs 3.5–4) will tend to precipitate from a plasma formed by passing too much current through a copper wire which is allowed to mix with air. In an inert atmosphere, maybe you could use vapor of aluminum or zinc.

If the sizing layer is thin compared to the fiber diameter, which would be necessary in this crazy plasma process but maybe not some of the others, the cost of the sizing is less of a concern.

Topics

- Materials (p. 1138) (59 notes)
- Pricing (p. 1147) (35 notes)
- Strength of materials (p. 1164) (13 notes)
- Composites (p. 1187) (9 notes)
- Ceramic-matrix composites (CMCs) (p. 1265) (4 notes)
- Fiberglass

My ideal workshop (unfinished)

Kragen Javier Sitaker, 02021-07-16 (updated 02021-07-27)
(2 minutes)

My ideal workshop would have a roll-down steel door (*cortina metálica*) 3 meters wide and 4 meters tall. Inside, it would be 5 meters wide, 25 meters long, and 5 meters tall. Its walls would be solid brick, shared with neighboring industrial buildings, and its ceiling corrugated galvanized sheet steel, interrupted with the occasional translucent fiberglass panel, supported on welded arch trusses. The floor would be sandy soil, and it would be on a rise or hill some 15 meters above the usual water table. It would have electricity but no water, sewer, gas, telephone, or internet service. The roof would slope down toward the street, and on its front edge, there would be an old sheet metal sign with the name of a former occupant, obscuring the view from the street of the roof. It would have a couple of sheet-metal chimneys.

I would build a meter-thick wall up to the ceiling, some 7 meters inside the front door, separating a 35-m² "front office" from a 85-m² "back office". I would dig out 500 mm of soil in the front office, grout the soil beneath it to solidify it, and replace it with broken-stone construction aggregate, thus providing a surface on which a delivery van could park safely. The front-office/back-office firewall would extend deeper than the broken-stone aggregate, and considerably deeper than that I would have to grout the soil that supported that wall to keep it from sinking in, because the wall is about 12 tonnes per square meter (120 kPa).

I would insulate under the steel roof with fiberglass batts. In the back office, I would replace 10m² of the steel with translucent fiberglass panels, under which I would install photovoltaic panels, with fiberglass batts beneath them.

The front office would XXX

Topics

- Fiction (p. 1368) (2 notes)

Can you 3-D print Sorel cement by inhibiting setting with X-rays?

Kragen Javier Sitaker, 02021-07-16 (updated 02021-07-27) (1 minute)

"Kiran K" of Larsen and Toubro reports that Sorel cement is sensitive to X-rays and strong DC electric fields — they prevent it from setting. (Even more astonishingly, he claims polarized laser light affects its setting.)

What if this is true?

Ring and Ping give more detail:

> Cements formulated near the stoichiometry of the 5-phase [i.e., $5MgO + 1MgCl_2$] reaction, e.g. 0.8987 kg/L (7.5 lbs/gal) and 28.9±0.15% $MgCl_2$ brine, are the strongest. Kinetic experiments observed that the 5-phase is formed quickly, but the kinetics are not complete for several days. The cement does not set by the 5-phase reaction when exposed to Cu Kα x-rays but gives a putty like form having $MgCl_2*6H_2O$ crystals that do not set into a cement. These are the first observations of x-ray altered cementation reaction kinetics.

Does this mean you could use X-rays to 3-D print an object out of Sorel cement?

Topics

- Materials (p. 1138) (59 notes)
- Digital fabrication (p. 1149) (31 notes)
- 3-D printing (p. 1160) (17 notes)
- X rays (p. 1310) (2 notes)

Tetrahedral expanded metal

Kragen Javier Sitaker, 02021-07-16 (updated 02021-07-27)
(3 minutes)

To make a three-dimensional honeycomb with the open-cell structure of the diamond crystal lattice, you make many sheet-metal strips of the same width, and you make a stack of two of these layers of strips, running at right angles, either with or without spacing between the strips. This gives you a grille of intersections between the strips. Color these intersections chessboard-style and spot-weld the black ones. Add a third layer at right angles to the second, each of its strips in the same position as a corresponding strip in the first layer (except displaced in Z by twice the sheet metal thickness), and spot-weld the white squares. (This may require a spot-welder with both electrodes on the same side of the workpiece, like those used for welding nickel strips to lithium batteries.) Now at every intersection the second layer is welded to either the first layer or the third layer. Add a fourth layer parallel to the second, and spot-weld it to the third layer at the intersections where the third layer isn't welded to the second (but the second *is* welded to the first). And so on.

Once you have added enough metal, pull the layers apart, permanently bending the strips so that each welded intersection becomes a tetrahedral lattice point.

If you have two-dimensional square mesh available for free, you can do this with half as many spot welds. Place a second layer of mesh on top of the first layer, with its X and Y axes parallel, but offset in both X and Y by half a cell, so that each wire in one layer of the mesh crosses a wire in the next layer exactly halfway between the two nearest intersections in each of their respective meshes. Spot-weld all these crossing points; repeat.

This isn't a very rigid structure, which is why it's possible to bend it into shape by pulling on it once it's welded up. If impact energy absorption is the goal, then that's fine; it should work great for that. However, if higher rigidity is desired, it's possible to take advantage of metals' work-hardening tendencies to get it. Say that two intersections are "metamours" if they are directly connected to a single common intersection. The trick is to add additional members to the structure connecting each pair of metamours which get further apart during the pulling process, of which I think each intersection has 8; these extra members are initially not straight, but the initial expansion of the matrix straightens them, which work-hardens them. If the expansion is done rapidly enough, the mass of the centers of these members comes into play, makng them straighter than the same amount of pulling force could have made them under quasistatic conditions.

This may be enough to get an ideal omnitriangulated mesh like an octet truss. There are still 4-cycles in the structure that have no diagonals, one between each pair of metamours in the same layer, but that's true of the standard octet truss as well.

Topics

- Materials (p. 1138) (59 notes)
- Manufacturing (p. 1151) (29 notes)
- Welding (p. 1181) (9 notes)

Glass foam

Kragen Javier Sitaker, 02021-07-16 (updated 02021-08-15)
(17 minutes)

You can get neutral waterglass to foam up into an open-cell-foam glass just by heating it to its softening point (I didn't measure with a thermometer, maybe 500°?) over the course of a few minutes. The structure of the resulting foam is fairly coarse, and the density is not that low.

How

A couple of possibilities occur to me to improve the structure.

Water is probably the best blowing agent for this kind of thing, just based on its low molecular weight. Ammonia might be a possibility, too.

First, waterglass solution doesn't solidify at room temperature until it's something like 35% waterglass by mass, and so when you heat it up slowly, any water in excess of that 65% (or even a bit more, since solubility increases at higher temperatures) will bubble out in the liquid state, before the solid foam can form. Adding small amounts of polyvalent cations (such as calcium or magnesium, or maybe even aluminum or boron) is well known to decrease its solubility to by orders of magnitude, so perhaps they could permit the formation of a solid hydrogel with much higher water content, perhaps 90%, 99%, or 99.9%, with a corresponding decrease in the density of the final product. There must be some density limit below which you get glass mist rather than glass foam, but given how viscous glasses can be just above their softening point, the limit might be pretty low.

Second, if the expansion is carried out very quickly rather than over a period of time, bubbles will have less time to coalesce into larger bubbles and convert the foam into the open-cell form. So perhaps the solid hydrogel precursor could be foamed more effectively in the way that rice and breakfast cereals were originally puffed (and still are by roadside vendors in China and Korea), by first heating it under pressure to its softening point, then suddenly releasing the pressure. For most cereals this is done by heating the cereal to the proper temperature inside a cereal cannon, then whacking the valve on the front of the cannon with a hammer, allowing the cereal to blow the cannon door open; but for popcorn it is instead done by encasing a small ball of the starch hydrogel in a hermetic hull which contains the pressure until it ruptures. Both possibilities might work to produce lighter-weight glass foam.

In the cannon case, there's the risk that the glass hydrogel will completely melt inside the pressure chamber and stick together into a single mass, and also to its walls. If the hydrogel is initially produced in granules, their surfaces can be treated to prevent this, for example by rolling them in quartz flour (which can be made to stick either by doing it before they have finished solidifying, by making the quartz flour hot enough to partially melt the surface, or by a hot air blast) or by replacing most of the alkali ions near their surface with polyvalent

cations like those mentioned above.

Many minerals might serve as viable alternatives to quartz flour here, including chalk, quicklime, alabaster, magnesia, zeolites, clays, talc, feldspar, aluminum hydroxide, mica, or rutile — almost any mineral used in formulating ceramic clay bodies or glazes will have a much higher melting point than the waterglass hydrogel, except perhaps oxides of lead. Boric acid and borax don't have a higher melting point, but still might be an alternative, by forming a borosilicate network in the surface of the granule.

Most of these treatments to increase the melting point of the granules' surfaces would also increase the surface's strength, which would permit the use of the cannon-free popcorn process.

An alternative means of puffing rice is "hot salt frying", in which salt is heated to a high enough temperature to puff rice (but not melt the salt), and then the rice (often parboiled) is mixed into it. The hot salt transfers heat to the rice much more rapidly than hot air would at ordinary velocities, so the rice puffs instead of dehydrating through vapor diffusion as it normally would. The coarse-grained puffed rice is then easily sieved out of the fine-grained salt. Modern continuous-process cereal puffing works the same way, but using high-speed air or steam rather than solids. This might be a viable alternative way to rapidly foam waterglass hydrogels, but media you wouldn't want in your food could be used instead of the salt — anything from the foregoing litany of quartz, chalk, quicklime, etc., and also carbon. These would permit the use of higher puffing temperatures. Using air, of course, would permit puffing glass at higher temperatures still, but its low thermal conductivity and thermal density means you need high-pressure air jets.

You could load a refractory mold full of many such beads of the hydrogel, but without any such surface treatment, then run hot air or steam through the mold cavity at high speed in order to expand all the grains and cause them to fuse together, like expanded polystyrene. This could be a pretty quick process if the grains are small; the hot air could convert them into a sort of fluidized bed.

One crude kitchen experiment

I placed a thin irregular slip of air-dried waterglass on a bed of expanded vermiculite, then dumped a small bowlful of preheated construction sand on it. The bowl and sand were not hot enough to glow visibly, but hot enough to immediately char the towel I was using to hold the bowl and the towel on the floor that I spilled some of the hot sand on. The expanded waterglass foam retrieved from beneath the sand was about 47 mm long at its longest, 27 mm wide at its widest, 20 mm wide at a narrower point, and 10–12 mm thick over most of its area. It weighed some 600 mg. If we approximate the volume as 45 mm × 22 mm × 10 mm, we get 9.9 cubic centimeters, so 0.06 g/cc, suggesting that it is about 98% air. Upon immersing it in a glass of water weighing 421.4 g while supporting it from above with chopsticks, the measurement increased to 430.4 g, suggesting it was displacing 9.0 mℓ of water, and upon weighing afterwards, its weight had increased to 1.5 g, suggesting that it had absorbed 900 mg of water, also giving a volume of 9.9 cubic centimeters. Although an extra decimal place of accuracy would give more confidence,

particularly given how the scale readings were drifting upwards while the foam was held underwater, it seems safe to say that it's probably 60 ± 30 mg/cc, so maybe 97%–99% air. (But contrast that 0.06 g/cc to the 0.74 g/cc reported for pumice in Material observations (p. 633) .)

The resulting piece of white foam can be easily handled without breaking it, but it's fragile enough that dropping it on the table sometimes breaks off a corner. Contact and rubbing make sounds similar to the sounds of extruded polystyrene, suggesting that the velocity of sound through the material is similar (and thus a similar density to stiffness ratio) and that acoustic coupling to air is pretty good, as you'd expect it to be if its density is only 60× higher than air's.

For context, a silica aerogel produced in the 01990s was the record-holder for least-dense solid material for a while at 3 mg/cc, more recent aerogels have reached 1 mg/cc, and extruded polystyrene panels for building are typically 20–80 mg/cc.

I was concerned that the sand or vermiculite might stick to the waterglass as it foamed up, but this didn't seem to happen — there were three pieces of vermiculite stuck to it, but I think they were there in the unexpanded waterglass, which had oozed onto a polyethylene sheet out of some vermiculite I was trying to glue together earlier.

Later, I reheated the foam to drive out the water and verify that it still weighed 600 mg. After removing it from the hot bowl on the stove, I could hear crackling inside the foam as the thermal shock induced fractures inside the foam.

Fragments of this foam melted and collapsed when heated with a butane torch to red or orange heat (say, 700–900°). In an effort to raise this softening point, I tried boiling another foam fragment made in a similar way in aqueous magnesium chloride for a while, then boiling it in tap water for a while to get rid of the magnesium chloride. The idea was that the magnesium, without changing the structure of the foam, would replace some or most of the sodium to produce some kind of mostly magnesium silicate, which would be water-insoluble and have a much higher melting point. After a first such tap-water boiling, there was some white deposit around the water (probably $MgCl_2$) but the foam no longer tasted noticeably like magnesium chloride. Just in case, I boiled it in fresh tap water for a while longer, which left less of a white deposit around the water.

Wet, the fragment sank in bottom of water and weighed 400mg. Dried, it weighed "0": under the 100mg low end of my crappy scale. After heating it with the blowtorch to an orange-yellow heat (800°–1000°?) for several minutes, which produced some sodium yellow in the flame at first, it appeared slightly smaller and less white, more translucent. Upon putting it in water, rather than floating at first as before, it immediately sank, and its wet weight was still "0", so, under 100mg. This suggests that its pore space had diminished by at least a factor of 4 from this treatment, and that the pores had become larger.

This survival for several minutes contrasts strongly with the behavior of the freshly prepared foam, which shrivels up to a tiny

bead in seconds upon being heated in the same way.

I'm repeating the $MgCl_2$ procedure with another piece from the 9.9mℓ chunk of foam just to make sure I'm not fooling myself. I kind of fucked it up because I let it boil dry, and then didn't let it cool back down before adding water, so water was boiling fiercely as it re-wet the foam, which probably did some real damage to its structural integrity. Also, the first time I did it in the cut-off bottom of an aluminum can; this time I'm doing it in a steel bowl with a badly oxidized plating of something like nickel, so there may be different compounds running around. And the water is cloudy white (perhaps due to particles of insoluble silicate broken off when I re-added the water) rather than transparent as before.

The second magnesium-infused chunk of foam did indeed withstand the blowtorch flame; but I think it lost significant weight during the magnesium-infusion process, and it looks significantly smaller. At any rate, once dry, it weighed in at "0 g" again. Wet, it weighs 1.1 g.

To test it further, I put it back in the bowl of vermiculite and built an aluminum-foil arch over the top of it, then heated it to yellow heat (parts to white heat) for several more minutes with the butane torch. It appeared unharmed, but after cooling was evidently more fragile than before, and its wet weight was 1.0 g, so it may have lost 10% or 20% of its pore space through this treatment. Also, the aluminum part of my butane torch nozzle melted and the brass tip fell out on the floor, so I had to stop.

To test compressive strength, I cut a 13 mm × 18 mm rectangular piece of the non-magnesium-treated foam and sanded two surfaces fairly flat, making it about 6 mm thick. I placed it on a larger scale and crushed it by pressing down slowly on a slab of granite placed atop it with my hand; it began to crush around 1 kg and completed crushing around 2 kg. 1 kg gravity / 13 mm / 18 mm is about 40 kPa, so the compressive strength of the foam is probably somewhere in the neighborhood of 20–80 kPa. This is an order of magnitude lower than construction-insulation styrofoams, which are typically in the 150–700 kPa range, measured by the DIN 53421 standard, which evidently specifies 10% deflection as the limit.

(See also Synthesizing amorphous magnesium silicate (p. 617).)

Why

The glass foam resulting from foaming waterglass can be abraded or sawn (or crushed) very easily, and the possibility of cutting it with a hot wire, like styrofoam, is very appealing, especially if its density can be decreased further. Although it's quite weak, it could be very useful for supporting granular slightly-denser materials such as perlite, vermiculite, alumina foam, and cheap carbon foam, for example while an adhesive sets (see Leaf vein roof (p. 600) for one use for this). Because of its relatively low melting point and very low density, it might be possible to "burn it out" in such cases, leaving only a thin layer of residue.

It was easy to bore a hole through one of the scraps prepared in the crude kitchen experiment described above using an 850-μm spring-steel wire, just by poking at the foam with the end of the wire.

The material did not visibly shatter or chip as the wire poked through the back side.

A different way of using this glass foam to support stronger materials is to first get it into the right shape (whether by expanding beads into a mold, gluing together a bunch of pre-expanded beads and pushing them into a mold, by cutting or abrading at low temperatures, or by hot-wire cutting) and then use the resulting form as either a mold or a stucco substrate. This is more or less the same way styrofoam is used for molding, for example, concrete, or as a base for a fiberglass layup. By painting, spraying, wrapping, laying up, or otherwise depositing a stronger material onto its surface, you can make a strong, hard shell. This may need to be done in stages, first building up a lightweight shell that can be supported by the foam, then a stronger shell supported by the lightweight shell, then perhaps a solid object filling the whole shell.

Why would you use glass foam for this rather than styrofoam? Well, it doesn't require any organic materials, so it's potentially much cheaper. Being more rigid means you can cut it to precise dimensions more easily, and it will bend less when you're doing things to it that impose slight side loads, like painting or stuccoing it. It can withstand common solvents without any complaint, unlike styrofoam, and it can withstand higher temperatures than any organic polymer. Counterintuitively, it might be possible to get it to a *lower* density than styrofoam, because silicate glass (even waterglass) has a much higher strength-to-weight ratio than polystyrene; however, experiment so far has not realized this possibility.

Also, if the density gets low enough, you can use glass foam for lost-foam casting, particularly for casting of materials like basalt, fused quartz, lead glass, or soda-lime glass that will happily dissolve the glass-foam residues.

It might be possible to raise the melting point of the foam once it has been shaped through ion exchange, for example with salts of magnesium, iron, boron, or aluminum. Saturating the foam with an aqueous solution of such soluble salts might be enough. This could enable it to be used directly as a refractory.

The standard established approach to foaming glass is to mix carbon and an oxygen source such as MnO_2 into it, so that as the glass starts to soften they react to form CO_2 (and, say, Mn).

Topics

- Materials (p. 1138) (59 notes)
- Manufacturing (p. 1151) (29 notes)
- Experiment report (p. 1162) (14 notes)
- Strength of materials (p. 1164) (13 notes)
- Foam (p. 1185) (9 notes)
- Waterglass (p. 1189) (8 notes)
- Glass (p. 1254) (4 notes)

Leaf vein roof

Kragen Javier Sitaker, 02021-07-16 (updated 02021-09-11)
(9 minutes)

Cheap roofs are mostly made out of corrugated galvanized sheet steel, but this has several disadvantages. A typical price is AR$2070/1.1m² (US$12/m²) for 500-μm-thick (25-gauge, in the Argentine system) corrugated galvanized steel, so the cost is not insignificant. The metal has no real insulating properties, reradiating all the heat of the sunlight that it absorbs, and although its finish is initially quite reflective, soon after installation it corrodes enough to absorb a lot of sunlight. It's kind of heavy (4.4 kg/m²). It tends to make a lot of noise when things fall on it. You have to drill holes in it to fasten it to things, which creates water leaks when it rains. It's a pain to bend or cut.

Maybe you could make a better material out of a sandwich panel. Take a layer of aluminum foil (50¢/m², see Aluminum foil (p. 413)) and lay down a layer of aluminum window screen on top of it; a 1.2 m × 30 m roll costs AR$13244, US$81, US$2.25/m². Or 200-μm-thick galvanized steel window screen, at AR$9053 for 1 m × 30 m, US$1.90/m². Or fiberglass window screen; a 1 m × 30 m roll costs AR$5377, US$33, US$1.10/m² — though that vendor says it's actually not glass but plastic! On top of the window screen, add a 50mm layer of expanded vermiculite (0.1 kg/ℓ, US$0.23/ℓ, according to Potential local sources and prices of refractory materials (p. 566)), previously moistened with waterglass (US$2/kg) and a second layer of window screen. Now press down the whole pile to ensure good contact among the vermiculite particles, and solidify the waterglass, either by letting it dry or by gassing it with CO_2.

(Dehydrated alabaster is a possible alternative binder, and it's cheaper at US$0.30/kg, but it won't coat the vermiculite grains as nicely, so you might end up using a lot more of it.)

(A layer of chicken wire (US$40 per roll of 1 m × 25 m, thus US$1.60/m²) or hardware cloth (US$82 per roll, thus US$3.30/m²) might provide additional strength and stiffness. 220g/m² woven fiberglass cloth for composites is US$3.50/m².)

The window screens provide tensile stiffness, and the somewhat springy vermiculite provides shear strength and impact absorption. The aluminum foil reflects the sunlight and adds a little extra stiffness, and the vermiculite provides insulation. The waterglass sticks it all together, and in particular keeps the aluminum foil from flapping in the wind.

Let's guess the weight of the waterglass is about the same as that of the vermiculite, which turns out to be 5 kg/m². So 1 m² is US$0.50 (foil) + US$4 (screens) + US$11.50 (5 kg vermiculite) + US$10 (5 kg waterglass) = US$26. So our panels weigh 10 kg/m² and cost US$26/m², each twice as much as the corrugated steel we were hoping to improve on. But now we have insulation and corrosion resistance, the panels absorb sound and can be cut with a box cutter, and we can drive screws into them without impairing their water

resistance. And they're still fireproof.

Loose vermiculite might conduct heat at 0.06 W/m/K, but with the waterglass it's probably more like 0.1 W/m/K. So if our vermiculite roof is at 45° and the indoors is at 20°, it will conduct about 50 W/m^2, which seems like a lot, even if it's only 5% of what enters through an open window. This is a U-value of 2 $W/m^2/K$.

What's the flexural strength of the panels? It seems like it ought to be something reasonable, but I'm not sure how to calculate it.

In this form, we haven't yet achieved a Pareto improvement but only a tradeoff; however, we've come within a stone's throw of the price and weight of the standard approach. Can we improve these panels further?

Because we don't need super high temperature resistance, it's probably better to use fiberglass insulation (US$0.026/ℓ, about 0.025 W/m/K, and also lower density, 0.02 kg/ℓ) or, if properly fireproofed, styrofoam at US$1/$m^2$ for 20mm (US$0.05/ℓ, 0.033 W/m/K); both of these have several times lower thermal conductivity than vermiculite. (Fiberglass is much less rigid, though, so perhaps it should be installed below the roofing panels rather than integrated into them.) Then maybe we could use a thinner vermiculite layer, just to provide a little stiffness, or a layer of gypsum. Or two layers of gypsum separated by a lower-density layer.

You could use perlite, LECA, or pumice, instead of vermiculite; all of these are stiffer than vermiculite but also denser and more expensive. Fired-clay ceramic made porous by the burnout of organic materials like sawdust or yerba mate is another possibility; though not commercially available, it's easy to make, and even at 75% or 80% burned-out filler, it's still pretty solid.

In a similar way, it ought to be possible to dilute the vermiculite with another granulated material of about the same granulometry, but much lower density and strength, without interrupting the continuous network of vermiculite grains in the finished composite. Let's call this bulking additive "filler". Crude reasoning suggests that the dilution could be up to a factor of 3: in a close packing of spheres, each sphere is in contact with 12 others, and to form a 3-dimensional network rather than a 2-dimensional one, at least 4 of those spheres need to be non-filler. See Glass foam (p. 595) for one filler possibility; styrofoam and foamed starch are two others.

If the vermiculite grains form a continuous network, these filler grains could be removed once the material has solidified. Organics could be simply burned out; foamed starch could be washed out with water much more quickly than the water would affect dried waterglass; anhydrous calcium chloride is a candidate filler that could be washed out with water and would also instantly, irreversibly harden any waterglass it came in contact with during the mixing process, keeping the bulk of each calcium chloride grain from being dissolved. Calcium chloride is fairly cheap (US$1.60/kg) and could be reused.

There are other candidate aqueous binder systems that could be similarly activated by surface contact with grains of a water-soluble "filler", including aqueous solutions of soluble carbonates and phosphates (I wrote about some of these in my Dercuano note on

powder-bed 3-D printing processes) and Sorel cement (where the "filler" grains would be magnesium chloride). Reversing the roles, aqueous calcium chloride itself is capable of forming a thin coating on grains of aggregate such as vermiculite, and then being hardened by surface contact with "filler" grains of anhydrous soluble carbonate or phosphate, or possibly of solid *potassium* silicate.

By carrying out such a "dilution" at multiple granulometry scales, it might be possible to get much lower vermiculite densities. Suppose 2-mm expanded vermiculite grains mixed 1:1 with 2-mm filler grains can successfully form a continuous vermiculite network, with a little binder, which seems likely. Then if the resulting 50%-vermiculite mixture is mixed 1:1 with 5-mm filler grains, the same process should repeat at larger scales: the vermiculite-network regions should be able to form a continuous-phase network around the 5-mm filler grains, for a 25%-vermiculite solid. A third stage of dilution with 12-mm filler grains would repeat the process, leaving a 12½%-vermiculite solid network with "pores" at different scales of 12 mm, 5 mm, and 2 mm, filled with useless filler particles, effectively a foamed foam.

Such high-porosity solids might be useful for a variety of purposes other than construction insulation.

A thin layer of quartz sand under the aluminum foil, once bonded with waterglass or something similar, would harden the aluminum-foil surface greatly against abrasion and impact, as well as providing a great deal more stiffness per dollar than metal reinforcement could. Construction sand (relatively pure quartz) costs US$0.03/kg and weighs about 2.4 g/cc. A 500-μm-thick layer would thus cost about US$0.04/m² per side and weigh 1.2 kg/m² (2.4 kg/m² on both sides). This would probably require the panel as a whole to be very stiff, because this layer of effectively mortar would crack off very easily if the surface flexed much. Gypsum is more expensive, but lighter and more flexible, so it might be a better option. Gypsum used as a binder for other lightweight, stiff aggregate such as expanded perlite might be better still.

Sandwich panel optimization (p. 754) has notes on how to get the cheapest sandwich panels for a given stiffness.

Topics

- Pricing (p. 1147) (35 notes)
- Strength of materials (p. 1164) (13 notes)
- Clay (p. 1179) (10 notes)
- Composites (p. 1187) (9 notes)
- Ceramic (p. 1193) (8 notes)
- Vermiculite (p. 1238) (4 notes)
- Life support (p. 1251) (4 notes)
- Insulation (p. 1290) (3 notes)
- Alabaster (p. 1309) (3 notes)
- Sandwich panels
- Roofing

Aluminum fuel

Kragen Javier Sitaker, 02021-07-17 (updated 02021-12-30)
(2 minutes)

Aluminum is a high-density fuel, 83.8 MJ/liter, though a little lower specific energy than, say, diesel fuel --- only 31 MJ/kg instead of 46 MJ/kg; but diesel is only 38.6 MJ/liter. That volumetric density is comparable to graphite (72.9 MJ/liter) and, in that table, exceeded only by beryllium (125.1 MJ/liter) and boron (137.8 MJ/liter). (I'm guessing anthracite is basically equivalent to graphite.)

All three of these materials (aluminum, beryllium, and boron) burn to solid oxides rather than producing gas emissions. In this context it's interesting to note that aluminum can be burned in aluminum-air fuel cells, typically at about 25% efficiency (comparable to that of heat engines), and that scrap aluminum is readily available; retail scrap buyers buy it at AR$100/kg from *cartoneros*, and at AR$172/US$, that's US$0.58/kg or US$0.019/MJ. This price is comparable to other fuels; crude oil is currently US$72/bbl, which works out to US$0.012/MJ at 1700 kWh or 5.8 million BTU per barrel, and retail refined fuel prices are often twice that. The US$0.04/kWh that is typical on the wholesale electrical market works out to US$0.011/MJ, though current solar energy costs one fourth of that, and retail prices are commonly ten times that.

Practical aluminum-air batteries can be made from very simple, inexpensive materials like table salt or potassa, carbon black, nickel, water, and paper, and they can tolerate the impurities that are common in scrap aluminum. More philosophical aluminum-air batteries are more efficient.

So you could very reasonably buy scrap aluminum as a fuel for when grid power is unavailable, such as for aviation or submarines.

Topics

- Materials (p. 1138) (59 notes)
- Energy (p. 1170) (12 notes)
- Aluminum (p. 1180) (10 notes)
- Independence (p. 1215) (6 notes)
- Batteries (p. 1302) (3 notes)

Boosters for self-propagating high-temperature synthesis (SHS)

Kragen Javier Sitaker, 02021-07-17 (updated 02021-12-30)
(4 minutes)

If you wanted to carry out self-propagating high-temperature synthesis, you could in many cases mix the synthesis feedstocks you desire to heat up with a redox fuel mix that will produce the desired heat, with a grain size chosen to produce the desired reaction rate.

The conventional reducer for this sort of thing is aluminum, with its oxide's enthalpy of formation of -1675.7 kJ/mol (-559/mol O). Magnesium at -601.6 (-601.6), beryllium at -599 (-599), and lithium at -595.8 (-595.8) excel it. Other candidate reducers include zirconium at -1080 (-540), silicon at -911 (-456), zinc at -350.5 (-350.5), and boron at -1254 (-418). All of these have extremely stable and unreactive oxides, at least at room temperature.

The conventional oxidizer is hematite at -824.2 kJ/mol (-274.7), yielding 283.8 kJ per mole of O1 when oxidizing aluminum. This is 5.33 kJ per gram of hematite at 159.687 g/mol. At 5.25 g/cc, that's 27.99 kJ/cc of hematite. But of course you also need 2 mol Al (26.9815 g/mol, so 53.96 g) at 2.70 g/cc (20.0 cc) to react with each mol (30.4 cc) of Fe2O3, so you actually only get 3.98 kJ/g of mix or 16.9 kJ/cc.

Hematite is abundant and cheap, but it has several disadvantages. Though in pure form it gets hot enough to boil the iron, you can't dilute this fuel mix very far with your actual desired reagents before the reaction stops being self-propagating, maybe a factor of 2 or 3. And the metallic iron produced is itself fairly reactive. Consider, by contrast, black cupric oxide (tenorite) at -156 (-156), thus yielding 403 kJ/mol; at 79.545 g/mol, you get 5.1 kJ/g of oxidizer, and the copper produced is considerably less reactive than iron (though easier to boil if the fuel is dangerously concentrated). Other promising alternative oxidizers include NiO at -240 (-240), chromium trioxide at -589.3 (-196.4), lead dioxide at -274.47 (-137.24), manganese dioxide at -520 (-260), and potassium permanganate at -813.4 (-203.4).

Consider a stoichiometric mix of lead dioxide with magnesium. We need 2 mol of magnesium (24.305 g/mol, so 48.610 g, occupying 27.895 cc at 1.737 g/cc) per mol (239.1988 g) of lead dioxide, occupying at 25.5 cc at 9.38 g/cc, for a total of 287.808 g in 53.4 cc. Burning it produces 2 * 601.6 - 274.47 = 928.73 kJ (464.37 kJ per mol of O1), which is 17.4 kJ/cc or 3.23 kJ/g.

This is sort of disappointing, although it is at least a little more heat per volume. I feel like I must have miscalculated something; the enthalpy yield per oxygen atom is *double* the conventional system, but the yield per cc is only slightly higher, and per g it's actually lower. I guess the lead is just too heavy and bulky?

How about lithium with manganese dioxide? You need 4 mol lithium (6.94 g/mol, so 27.8 g, occupying 52.0 cc at 0.534 g/cc) per mol of MnO2 (86.9368 g, occupying 17.30 cc at 5.026 g/cc), for a

total of 114.7 g in 69.3 cc. It burns to metallic manganese and 2 mol of Li_2O and 2 * 595.8 kJ - 520 kJ = 672 kJ, 9.69 kJ/cc or 5.86 kJ/g. Super disappointing!

Still, not so disappointing as to be useless for boosting SHS. Even fairly small admixtures of these boosters may be adequate to permit SHS of feedstocks that would be hopeless on their own.

Topics

- Materials (p. 1138) (59 notes)
- Self-propagating high-temperature synthesis (SHS) (p. 1241) (4 notes)
- Enthalpy (p. 1369) (2 notes)

Compressed appendable file

Kragen Javier Sitaker, 02021-07-19 (updated 02021-07-27)
(5 minutes)

PDF is a mess because it tries to support incremental updates by way of appending, random access, and compression, in a way that's deeply intermeshed with the application-layer data structures it uses and unnecessarily inefficient as a result. Each indirect object has an identifying number that can be used to make references to it. Indirect objects include independently compressed streams for the contents of each page, and you have an xrefs table at the end of the file that tells you the byte offset at which to find each indirect object.

In PDF 1.5, they added compressed object streams containing a bunch of non-stream indirect objects that all get compressed together, preceded by a variable-length free-form header that gives the byte offset of each indirect object in the decompressed stream, but of course the objects don't have byte offsets in the top-level file that could be listed in the traditional xrefs table; instead they added a new xref stream format which has type-1 xrefs (the traditional kind) and type-2 xrefs that give the object number of an object stream and an index into it.

Despite all this, you still can't compress the contents of multiple pages together, so even with all the bells and whistles, PDF files are still significantly bigger than the corresponding gzipped PostScript files.

I think the right solution is a layer separation: generic data compression should be provided by a layer underneath the application data structures. For example, a filesystem supporting transparent compression would allow PDF files to just forget about compression entirely, just using text and byte offsets and relying on the filesystem to map byte offsets to the proper place in the transparently decompressed data; and the files would occupy less space than using the current PDF Rube Goldberg scheme.

Filesystem compression doesn't help with network transmission, though. For that you need a file format, implemented by a library. But efficient traversal of graph structure as in PDF requires random seeks, and traditional compression libraries like gzip don't support random seeks.

This is pretty much the same problem the dictzip program from Rik Faith's dictd solves: it resynchronizes the gzip compression algorithm every sub-64-K "chunk", and provides a gzip-compatible format that stores an "Extra Field" in the gzip header that tells how many bytes each chunk compressed to, and how many bytes the chunks were originally. The man page reports that with 64kB chunks, the file is only 4% larger than with unchunked gzip.

However, PDF files, like many other candidate uses for such a file format (such as debug logs and database journals), need to be able to append data to the end of the file. I think dictzip can't provide this, because it would have to expand its header field with data about the newly added chunks, which I think would involve shifting the data in

the rest of the file a few bytes to the right to make room.

PDF and PKZip both take the approach of appending a file "directory" or "xrefs" trailer or footer that permits you to efficiently find everything else in the file; in the case of a compressed bytestream format, the only thing to find are the seekable compressed byte offsets most closely corresponding to particular uncompressed byte offsets. The .xz file format also does this, including an "Index" just before the "stream footer" which gives the compressed and uncompressed size of each of the data blocks in the stream. This allows you to parse an .xz file backwards from the end to seek randomly in it.

Moreover, an .xz file can consist of multiple concatenated streams:

```
$ (echo hi | xz -9c; echo bye | xz -9c) | xz -dc
hi
bye
$
```

So in theory you could use .xz as a compression format that supports appending and random reads. However, your reads (or opens) are going to eventually take time proportional to the total number of appends that have been done, rather than constant time, as is desirable. And if an append operation is interrupted, for example by a process being killed or a server going down, it's likely that the file will no longer end with a stream trailer, eliminating the ability to read it safely from the end.

If we add a single fixed-size field in the file *header* (like dictzip does) that points at the latest trailer (unlike dictzip), we can atomically update that field after appending a new commit to the file. And all the stuff we append in a single commit can be compressed together.

Topics

- The Portable Document Format (PDF) (p. 1227) (5 notes)
- File formats (p. 1233) (5 notes)
- Compression (p. 1263) (4 notes)

SHS of magnesium phosphate

Kragen Javier Sitaker, 02021-07-22 (updated 02021-07-27)
(3 minutes)

According to doi:10.1021/acs.chemrev.5b00463, one of the difficulties with magnesium cements using phosphoric acid is:

These reactions are highly, often violently, exothermic, which raises practical challenges regarding the use of this process on a large scale.

Specifically they're talking about $MgO + 2H_3PO_4 + H_2O \rightarrow Mg(H_2PO_4)_2 \cdot 2H_2O$, which is soluble, and $MgO + H_3PO_4 + 2H_2O \rightarrow MgHPO_4 \cdot 3H_2O$, which is not.

"Highly, often violently, exothermic" sounds great for self-propagating high-temperature synthesis, and both MgO and H_3PO_4 are solid at ordinary temperatures. But where do you get the water?

Aside from preparing the mix well below 0°, a possible alternative is to use a hygroscopic magnesium salt rather than the oxide. Magnesium chloride is annoyingly hygroscopic, as I've been experiencing today (normally a hexahydrate), and Epsom salt (normally a heptahydrate) actually has an *undecahydrate*, Fritzsche's salt, which melts at 2°. So even a little bit of the hygroscopic salt might be adequate. Magnesium nitrate is so hygroscopic it can't even be dehydrated by heating. The bromide is outright deliquescent.

Another possibility is the oxalate, which is normally a dihydrate and decomposes to the oxide on heating.

So, for example, maybe you could use $5MgO + 2(MgSO_4 \cdot 7H_2O) + H_3PO_4$.

A different way to use this rapid cementation reaction might be to spray aqueous phosphoric acid (or dissolved phosphates of ammonia) onto a powder bed containing an MgO or $Mg(OH)_2$ binder, and a filler such as quartz, in order to 3-D print. The paper also mentions rapid-setting magnesium phosphate cements:

Although earlier patents and articles used liquid polyphosphates or diammonium phosphate, by the late 1980s, MgO and powdered monoammonium phosphate were the preferred materials, shipped as dry powders, principally forming a crystalline struvite binding phase when mixed with water, according to eq 12.134 ... Addition of water to a blend of monoammonium phosphate and MgO results in a mass that sets too rapidly to be of use, and thus, early MAP patch repair cements used a separately packaged ammonium polyphosphate solution that reacted more slowly with the MgO.

However, these form the wimpy struvite rather than phosphates of just magnesium.

It also mentions that sodium dihydrogenphosphate (monosodium phosphate) is twice as soluble as the ammonium salt (MAP), which is 57% more soluble than the potassium salt.

The magnesium salts that are easy to find here are the citrate, the carbonate (gym chalk), the chloride, and the sulfate, which (being fertilizer) is cheapest: US$1/kg, oddly cheaper than even dolomite. The oxide *is* available but only in an expensive food-grade form. The nitrate is also available as fertilizer, but costs much more than the

sulfate.

See Synthesizing reactive magnesia? (p. 615) on getting different oxides and hydroxides.

Topics

• Materials (p. 1138) (59 notes)
• Pricing (p. 1147) (35 notes)
• Phosphates (p. 1184) (9 notes)
• Magnesium (p. 1213) (6 notes)
• Cements (p. 1235) (5 notes)
• Self-propagating high-temperature synthesis (SHS) (p. 1241) (4 notes)

Back-drivable differential windlass

Kragen Javier Sitaker, 02021-07-23 (updated 02021-07-27)
(15 minutes)

(This note has several calculations I've noted errors in that need redoing.)

It's often stated that one of the advantages of the Chinese windlass mechanism is that it's self-locking, or not back-drivable. Another advantage is that, being a differential mechanism, its mechanical advantage can be arbitrarily large. A third advantage, rarely remarked upon perhaps because of its obviousness, is that since most of the mechanism is purely in tension, it can extend over a great distance while containing very little mass; it is practical to construct a Chinese windlass that applies substantial forces between points a hundred meters distant, weighing only 200 grams, half of which is 1-mm UHMWPE cord. A fourth is that most of the contact in the mechanism is not sliding contact and thus does not cause abrasive wear or frictional losses; only the bearing of the windlass drum has sliding contact, and possibly the bearing of the pulley.

In a sense the third advantage above is the opposite extreme from Reuleaux's definition of a machine as something that imposes motion in desired degrees of freedom but prevents it in all others — that is, has effectively infinite rigidity in all other degrees of freedom. The differential pulley in the windlass mechanism is imposes motion in one desired degree of freedom, employing motion in a second degree of freedom (usually attached to a bearing so this can be ignored); it is somewhat restrained in a third degree of freedom, though not very rigidly; and has effectively infinite *compliance* in the remaining four degrees of freedom. In a well, that is, the bucket can swing back and forth and twist around while receiving only very small restoring elastic forces from the rope's minuscule rigidity.

One disadvantage of the mechanism is that the differential mechanical advantage can be somewhat imprecise; as layers of rope build up on the drums, they change the drums' effective radius and potentially their difference in radius. Grooved drums can prevent this from happening, but only if the drums are long enough.

The M.A. of the mechanism is the ratio between the crank arm and the *difference* in drum radii. This implies that the absolute drum radii can be as large or small as desired without changing the M.A. However, if the difference in radii is, say, 1mm, you only get 6.28mm of elongation per revolution, regardless of whether that revolution is running 100 mm of cord through the differential pulley or 100 m of it. So, this allows you to increase the rigidity of the drum, which might allow you to increase its length, thus permitting more unlayered cord, but not to use less layers of cord in the same length.

The wrapping of the cable on the drum can be protected from side-loadings by running the cable through a grommet in between the drum and the differential pulley. That way the angle at which the cable rolls onto the drum only depends on where the cable is on the drum and the tension on the cable, not on any other side-loadings. If

the grommet is sort of hyperboloid-of-one-sheet-shaped, it will avoid kinking the cable there and avoid any concentrations of force at one point on the grommet, and if it is made of a hard material such as sintered sapphire, like sparkplugs, it will not suffer much from abrasion.

The self-locking nature of the mechanism is an advantage for some uses, but being able to use an arbitrarily large mechanical advantage in reverse would be useful in some situations. The reason it is usually self-locking is that the frictional torque is the side loading on the bearing, multiplied by the bearing's radius, multiplied by the bearing's coefficient of friction μ; and the frictional force resisting the differential pulley is that torque divided by the effective moment arm, which is the difference in radii. Typical coefficients of friction for dry journals are 0.2–1.6; for example, bronze on cast iron is 0.22, wood on dry wood is typically 0.25–0.5, steel on steel is 0.5–0.8, and copper on copper is 1.6. Usually the side loading on the bearing *is* the force on the differential pulley. So, if $\mu = 0.22$, then 100 N of pull on the differential pulley will generate 22 N of friction at the bearing.

Suppose, for example, that the journal is 20 mm radius (40 mm diameter), and the radius difference is 10 mm (say, the drums are of radii 50 mm and 60 mm). So the torque on the shaft from the differential pulley is 100 N · 60 mm - 100 N · 50 mm = 100 N · (60 mm - 50 mm) = 100 N · 10 mm = 1 N m. And the torque from the friction is 22 N · 20 mm = 0.44 N m. So in this case the mechanism will not be self-locking. (It will be somewhat efficient: 1.44 N m from the crank will be converted to 1 N m at the drum and thus 100 N, for 69% efficiency.)

If we increase μ past 0.5, though, (the reciprocal of the M.A.), the frictional torque rises past 1 N m, the mechanism becomes self-locking, and its efficiency falls below 50% (assuming static and dynamic friction are equal), because the frictional force has exceeded the force from the slow/strong end of the mechanism, the differential pulley. The same thing happens if we leave μ at 0.22 and increase the M.A. past 4.55, for example by increasing the journal radius from 20 mm to 46 mm or by decreasing the difference in radii from 10 mm to 4.3 mm.

This is a case of a general phenomenon where mechanisms with efficiency above 50% are backdrivable, and those with lower efficiencies are instead self-locking, under certain simplifying assumptions.

But a M.A. of up to 5 is a far cry from an arbitarily large (inverse) M.A. What can we do if we want to use this mechanism to make things go fast instead of slow? Like, what if we want to pull a string to spin something at 15000 rpm by hand to generate electricity? Maybe something only 120mm in diameter, so it can be comfortably handheld?

First, consider the parameters of the problem: a person can pull about 200 N at about 3 m/s for about 1 m. (Think of someone trying to start a lawnmower or chainsaw with a pullcord.) The edge of the disc, which will be generating the alternating magnetic field that generates electricity, will be moving at about 94 m/s. So we need a M.A. of more than 62, probably at least 200 to be safe. We're

missing a factor of 40.

With modern UHMWPE fiber and its 3-GPa tensile strength, 200 N requires a 290-μm-diameter cord; better say 500 μm to be safe, which should be good to 580 N. This means that a single layer of wrapping on a 20-mm-long barrel holds 40 revolutions, and each layer adds 500 μm to the radius. An 8-mm-diameter barrel with its 25-mm circumference would hold about 1 m of cord per wrapping layer.

We can make some progress on the problem by using better bearing materials. Steel on polyethylene has $\mu \approx 0.2$, on graphite, $\mu \approx 0.1$; on teflon, 0.05–0.2; and on tungsten carbide, 0.4–0.6. Tungsten carbide on tungsten carbide is listed as 0.2–0.25. Materials with lower μ might be worse if they require a larger journal radius to compensate for lower compressive strength; WC's 3+ GPa compressive strength would theoretically allow it to bear 200 N on a 250-μm-long 250-μm-diameter shaft, if it doesn't bend too much, for example.

XXX I've confused circumferences and radii so there's a missing factor of τ from some of the below

Or a 500-μm-long 125-μm-diameter WC shaft, maybe somewhat tapered to reduce the risk of breakage. That would give you a 62-μm bearing radius, so each 100 N of side load produces, say, 25 N of friction, but only 1.55 millinewtonmeter of friction torque. So if your difference in radii is, say, 30 μm, say because one drum is 8.00 mm in diameter and the other is 8.06, the 100 N will produce 3 millinewtonmeters of torque, which is twice as much as the friction, so it will be able to backdrive the mechanism. The M.A. to the magnet disc, then, will be 60 mm / 30 μm, or 2000. Our 3 m/s human pullcord would be able to spin the rim of the magnet disc at 6000 m/s, or Mach 18, spinning at 955000 rpm. Also, the ⅓ of the 200 J lost to friction, or 60 J, would be deposited in the tungsten carbide bearings, which are some 0.1 mm³ or about 1.6 mg of WC; with its room-temperature specific heat of 200–480 J/kg/K its temperature would rise to between 78000° and 188000°.

Although the bearings could withstand the mechanical pressure at room temperature, they would vaporize and the disc would explode. Also, any practical mass of disk would make the pullcord too hard to pull. And extending the pulley pull handle by 1 m at 30 μm of difference per revolution and 25 mm of cord per revolution would require 33000 revolutions, unwinding some 800 m of cord from one drum and winding it up on the other. Clearly this is taking things too far!

Let's try backing off to our planned M.A. of 200: a difference of radii of 300 μm. And let's consider ordinary cast iron on bronze: $\mu = 0.22$. Cast iron can withstand 500+ MPa of compression, but UNS C93200 (SAE 660) bearing bronze only some 300 MPa, with a fatigue strength of only 110 MPa. Also, consider that there are two bearings, one at each end, so each can bear half the 200 N load. 100 N ÷ 110 MPa gives 950 μm diameter × 950 μm length, say, or 450 μm diameter × 2 mm length, giving 225 μm radius (probably, again, an average over a slight taper). 200 N · 0.22 · 225 μm = 9.9 millinewtonmeters of friction, and 200 N · 300 μm = 60

millinewtonmeters of applied differential torque. The leftover 50 mN m manifests as an 0.8 N resistance at the 60-mm-radius generator ring, which should be quite straightforward to produce by, for example, generating electricity through pancake coils.

This is an improvement, but we still face the dismaying prospect of 3000 revolutions of the main spindle unrolling 75 m of UHMWPE thread from one drum and rolling it onto the other. That's a lot of revolutions! And the 17% power loss in the main journal bearings is worrisome not only because it's wasteful but also because of the heat problem.

By adding an additional idler pulley behind one of the differential drums, so that the side loadings from the two strings are in opposite directions, you can reduce the side loadings on the drums' bearings. The idler adds some friction, requiring a journal similar in stoutness to that of the main wheel, but it can be quite narrow (<1 mm) and large in diameter (40 mm, say). So the string running around the idler has a lever arm of 20 mm with which to counteract the 0.225-mm lever arm of the journal's friction, an M.A. of 89, which when divided by $\mu = 0.22$ gives us a factor of some 400. So the idler will consume about ¼% of the energy; and it ought to be able to reduce the friction in the bearings of the main differential wheel considerably, perhaps by a factor of 2, which matters a great deal more, because their friction has a great deal more M.A.

Instead of just having one idler pulley, one differential pulley on the pull handle, and one spindle, we can improve the situation further: by adding three more idlers in the body and three more in the pull handle, with an additional 1.5% efficiency loss, we can run the cable back and forth between the main body and the pull handle eight times instead of twice. This enables us to use a much smaller M.A. in the differential windlass mechanism itself — 50 instead of 200, realized by paying out 1.2 mm of differential cable per revolution of the drums rather than 0.3 mm. By pulling the two parts of the mechanism apart by 1 m, 8 m of cable is demanded of the differential windlass itself, which is still (!?) 6700 revolutions. But now each cable only bears 25 N (XXX, I was overspeccing it above by a factor of 2) and so can be hair-thin: 150 μm diameter at 3 GPa should withstand 50 N. And the side-loading imposed by these threads on the spindle, and the frictional losses in the journals, are correspondingly lower.

It's important to design the handle so the pulleys can pivot to equalize tension among these threads; otherwise you will accidentally put all the tension on one thread while yanking the handle and break it.

Suppose our windlass barrels are 30 mm long and 7 mm in diameter. Each layer of this thread on the barrel is 200 revolutions, 22 mm in circumference, and thus holds 4.4 meters of thread. If this circumference remained unchanged, there would be 33 layers, totaling 4.95 mm in thickness, on a fully charged barrel. This is clearly a practical volume of thread.

I'm still concerned, though, about the thread thickness changing the barrel radius and thus the M.A. 1.2 mm of circumference difference is only 190 μm of radius difference: barely more than a single layer of thread.

It's worth mentioning that in such a mechanism the electronics could be entirely sealed away from the rotor and pulleys, communicating exclusively through magnetic fields.

Topics

- Materials (p. 1138) (59 notes)
- Physics (p. 1157) (18 notes)
- Mechanical (p. 1159) (17 notes)
- Strength of materials (p. 1164) (13 notes)
- Tungsten carbide
- UHMWPE
- Friction

Synthesizing reactive magnesia?

Kragen Javier Sitaker, 02021-07-25 (updated 02021-08-15)
(4 minutes)

(See also Synthesizing amorphous magnesium silicate (p. 617).)

Suppose you want reactive magnesia in order to make, for example, Sorel cement for X-ray 3-D printing (see Can you 3-D print Sorel cement by inhibiting setting with X-rays? (p. 592)), or exothermic refractory magnesium phosphate cement (see SHS of magnesium phosphate (p. 608)) , but it isn't sold locally, but other soluble magnesium salts are available. What can you do?

Well, calcining the carbonate or the hydroxide produces magnesia at only 350°, and according to WP:

> Magnesium carbonate can be prepared in laboratory by reaction between any soluble magnesium salt and sodium bicarbonate... If magnesium chloride (or sulfate) is treated with aqueous sodium carbonate, a precipitate of basic magnesium carbonate—a hydrated complex of magnesium carbonate and magnesium hydroxide—rather than magnesium carbonate itself is formed

So magnesium oxide can be made with a soluble carbonate or bicarbonate (such as baking soda) plus the citrate, the chloride, or the sulfate of magnesium, thus forming the carbonate of magnesium (possibly, or possibly the bicarbonate), and just a bit of heat. Though WP also says that if you don't heat it to at least 700° it's so reactive that it will recarbonate from the air.

An alternative alkali-free approach is taken in Kiwami Japan's "sharpest Seawater kitchen knife in the world". They begin by boiling down the filtered seawater twice to crystallize some salts (presumably sodium chloride?) which are filtered out, leaving a solution of mostly magnesium chloride (solubility 72.6 g/100mℓ at 100°, twice NaCl's 36 g/100mℓ).

Then they calcine seashells in a microwave-oven kiln made of insulating firebrick, using charcoal as the susceptor rather than (or perhaps in addition to) the more conventional silicon carbide, for 15' at 1000 W, reaching a bright yellow heat, twice. This yields pure white quicklime seashells, which are ground to a granulometry around 100 μm using a mortar and pestle and a meat grinder. This heats from 25° to 65° upon hydration with water from a cow-shaped pitcher. A little limewater (1.5 g/ℓ slaked lime) is decanted and filtered off.

Adding some limewater to concentrated seawater (containing a significant amount of magnesium chloride) precipitates some magnesium hydroxide (solubility 0.00064 g/100mℓ) over the next 30', which is filtered off, along with some sodium chloride. This is washed with water from another cow-shaped pitcher to remove the salt, dried, and ground to get dry magnesium hydroxide, which is then moistened with magnesium chloride solution to produce a weak Sorel cement (11 Shore D). Calcining the hydroxide in the microwave-oven kiln (with charcoal as susceptor, 15', 1000W, heating to a yellow heat measured as 970° with a pyrometer) yields the oxide. Moistening the oxide with the chloride solution yields a much harder

Sorel cement (102–109 Shore D, but I think that may be as high as his meter goes).

An open rectangular mold was filled with this cement and vibrated with a Handy Massager to remove bubbles; when set, the demolded rectangle was cut with an abrasive wire saw into the rough shape of a kitchen knife. While wearing rubber gloves equipped with reptilian claws, a gavel was used to drive a scrap of Sorel cement through the top of an Altoids tin, damaging the face of the gavel. The knife blank was then ground with a series of files and abrasives to form the knife edge, which then successfully cut a cucumber extruded from the reptilian glove.

David Reid's experiments with "Microwave melting of metals" found that magnetite works well as a susceptor for microwave-oven kilns, more efficiently than graphite, up to 900° (though its Curie point is only 580°, and you would expect it to become transparent to microwaves above that temperature). He reports that the magnetite tends to flux the sand and melt it, though. His use of uninsulated crucibles explains the need for more-efficient susceptors; Kiwami Japan's use of a few centimeters of insulating refractory ceramic avoids this problem. Previously Reid had used ceramic-fiber furnaces.

Some other recent work has used graphene paint as a susceptor.

Topics

- Materials (p. 1138) (59 notes)
- Magnesium (p. 1213) (6 notes)
- Microwave heating

Synthesizing amorphous magnesium silicate

Kragen Javier Sitaker, 02021-07-25 (updated 02021-08-15) (6 minutes)

See also Firing talc (p. 576) for notes on magnesium-silicate ceramics processing, Glass foam (p. 595) for notes on using magnesium chloride to raise the softening point of waterglass foam, and Synthesizing reactive magnesia? (p. 615) on the oxide.

In a crude kitchen experiment, I mixed 136 g of a waterglass solution (35%?) with 136 g of tap water in a cut-off Monster can, then took 72 g of kinda gummy magnesium chloride from the sack where it had been deliquescing and melted it on the stove. (If this were the hexahydrate that would be 34 g of anhydrous.) It stubbornly refused to dissolve completely, so I added 48 g of tap water, which did the trick. Upon dumping this into the diluted waterglass, it immediately (<500 ms) formed a solid white slushy material that I couldn't really stir with the chopstick, with only a little bit of milky liquid to be squeezed out of it. I was able to pour off 62.7 g of this syrupy liquid with a few chunks, so there's about 330 g of solid "gel" in the can, of which only about 14% is waterglass and 22% is the hydrated magnesium chloride, so probably something like 10% is actual anhydrous magnesium chloride. So it's 76% water, 24% solid matrix. It feels gritty, but it's easily crumbled by hand. The stuff at the base of the can, where probably less magnesium penetrated, was even grittier.

Rather than any sort of glassy material, this seems more like a crystalline precipitate that's got a lot of water locked up inside its crystalline structure (a fairly well-defined amount, not the kind of loosey-goosey anything-goes relationship you see with a gel) and some more water in the interstices between the numerous sand-sized crystals. The obvious candidate would be talc, but although the crystals are soft, they don't feel as soft as talc. More gritty, like baking soda. I can't identify the crystal habit, and I wonder if maybe the hard substance formed along a mixing boundary between the two solutions rather than in any sort of crystalline way. It doesn't get less gritty when I rub it around on my hand for a while, and it doesn't dissolve in (96%) ethanol. Heating it at 500° for half an hour leaves it whiter, and it's still super crumbly, but tells me nothing else useful.

Apparently this is the recipe for synthetic magnesium silicate, which forms porous, amorphous masses.

Roughly guessing at the stoichiometry, 136 g of 35% waterglass would be 47 g of waterglass. If it's 3:1 SiO_2:Na_2O by weight then that's 35 g of silica, which is 60.08 g/mol, so 0.58 mol of silica (and of silicon). Anhydrous $MgCl_2$ is 95.211 g/mol so 34 g of it would be 0.36 mol of $MgCl_2$ (and of magnesium). So that's about 1.6 silicons per magnesium (Mg/Si molar ratio 0.62). Forsterite has 2 magnesiums per silicon (0.5 silicons per magnesium), serpentines have 1.5 magnesiums (or irons) per silicon, enstatite (the eutectic) has 1 silicon per magnesium, and talc has 1⅓ silicons per magnesium. So on

average this material is a little closer to silica gel than talc is. Synthetic magnesium silicate for food has 2.5 silicons per magnesium.

It gets a little crunchier when I wash it, maybe because a little residual sodium silicate was lubricating it.

Apparently at low enough Mg/Si ratios you get crystalline talc but only above 200°. The paper suggests that maybe I have some silica mixed in with my magnesium silicate because I didn't add enough magnesium.

I stuck a chunk of the stuff in a bowl with a little sunflower oil in it, and it provided a pretty usable oil-lamp wick, so I guess it must be pretty porous. A deposit of porous black carbon built up on the top surface, and the flame is pretty smoky once it gets going, a process that often involves a certain amount of spluttering. There's a bit of an acid smell to it, like a lit match or the exhaust from high-sulfur diesel. I tried sand and vermiculite as alternative wick materials, which worked much more poorly. The amount of smoke is high enough to discourage me from using it in practice, but cutting it to a better shape might solve that problem, and of course blowing the flame into a combustion chamber with extra air would solve it.

The microporous nature of the material suggests that it might be useful for filtering; a major commercial use of the stuff is adsorbing polar radicals from used frying oil.

One paper describing the synthesis of a magnesium silicate with a particular molar ratio added the silicate and the magnesium salt dropwise to a continuously stirred solution of ethanol thickened with PEG. I guess the idea was to keep the concentration of unreacted feedstocks low enough that they would produce a mineral of a consistent composition.

Topics

- Experiment report (p. 1162) (14 notes)
- Minerals (p. 1210) (6 notes)
- Magnesium (p. 1213) (6 notes)

Ropes with constant-time concatenation and equality comparisons with monoidal hash consing

Kragen Javier Sitaker, 02021-07-27 (15 minutes)

Avery Pennarun's bupsplit and Silentbicycle's Jumprope, used in his Tangram filesystem, are something like a hash-consed general-purpose rope structure.

It occurred to me that by extending this approach, you can build a string type that supports in-practice-constant-time concatenation and comparison operations, at least in the absence of adversarial input. Traversal is linear time; extracting substrings by numeric byte position is logarithmic.

Hash consing

Hash consing is an interesting technique: it allows you to reduce all equality comparisons to pointer comparisons, extending the magic of garbage collection, which lets you reduce all copying of (immutable) values to copying of pointers, to equality comparison. This is clearly very desirable for hashing and associative storage: you can hash the pointer, for example by interpreting it as an integer, rather than the data structure, thus obtaining constant-time indexing of properties by arbitrarily complex data structures.

Recent advances in hashing, such as cuckoo hashing and Robin Hood hashing, reduce some of the historical downsides of hashing.

Hash consing is not called that because it's useful for hashing, though, but because you have to hash the values you're consing in order to find out whether the cons already exists. If it does, you just return it, rather than allocating a duplicate.

Ropes

Ropes represent strings as, basically, the fringe of an arbitrary cons tree structure, typically with the size of each subtree cached at each node. This permits linear-time traversal, logarithmic-time indexing by byte offsets, and constant-time concatenation. However, if we want to use hash consing for ropes, we have a problem; string concatenation is supposed to be associative: "a" || ("b" || "c") == ("a" || "b") || "c". If we just use regular hash consing with ropes, we may end up with lots of duplicate ropes that represent the same string.

Lua hashes a new string in constant time: at least in Lua 5.2, `luaS_hash` examines at most 32 bytes of the string in order to compute its hash value. But it can't *construct* the new string in constant time, or usually constant time, because it has to copy all of its bytes. Similarly, it can't test strings for equality in constant time, or even usually constant time—even if their hash values are equal, it has to iterate

over all their bytes.

The Jumprope approach is to use a rolling hash to split the string into chunks in a consistent way, so that strings that contain common substrings will usually split into the same chunks. But Jumprope does not aspire to support string concatenation or, I think, constant-time string comparison.

Monoidal hash functions

Here's one way you could get *usually* constant-time string concatenation and always-constant-time string equality tests. You compute the hash used for cons hashing in such a way that equivalent ropes will always have the same hash value, even if their internal structure is different. This is done by choosing the hash to be a monoid under a constant-time operation applied at concatenation time.

A particularly appealing kind of monoid for this purpose is functional composition of linear forms $mx + b$; the hash of the concatenation of string s_0 with hash $m_0x + b_0$ and string s_1 with hash $m_1x + b_1$ is then $m_1(m_0x + b_0) + b_1 = (m_1m_0)x + (m_1b_0 + b_1)$. So to compute the hash of the concatenation of the tuples (m_0, b_0) and (m_1, b_1), we just compute $(m_1m_0, m_1b_0 + b_1)$. Since the empty string is the identity element of the string-concatenation monoid (which is the free monoid on the characters) the hash of the empty string must be (1, 0), while the hashes of the characters can be chosen to be anything convenient, such as (3, c).

It isn't necessary for the multiplication and addition operations here to be integer multiplication and addition, and in fact it's undesirable, because integers can be arbitrarily large, and we want our hash values to be of constant size, and the computation of their composition to be constant-time. We only need the addition and multiplication operators to be associative, the multiplication to left-distribute over the addition, and for them both to have identity elements (written 1 and 0 above); that is, they each need to be a monoid, with one left-distributing over the other. The usual approach in such cases is to use "machine integers", say multiplication and addition mod 2^{64}, and that would work fine here. Reducing modulo a large prime, such as $2^{64} - 59$, might be advantageous, in order to keep bits well mixed. Vectors or matrices would work too.

But any semiring with identity (and in particular any ring and any field) would work; we don't even need the addition operation to be commutative, as semirings guarantee. Rings in general will tend to work better, because the addition operation of a ring is information-preserving, so you can't end up in a trap where all strings with a given prefix have the same hash. (Idempotent semirings and especially tropical semirings would probably be bad choices.) Multiplication in a ring isn't information-preserving, so with the above it's possible to get into a trap where all the strings with a given *suffix* have the same hash, or where a lot of information is lost, but this should generally be easy to avoid; when the multiplicative inverse exists for nonzero elements, for example, you just need to prevent the hash of any primitive rope from containing a 0 multiplier.

It might be desirable to prevent an adversary from choosing strings to produce a high collision rate, and the linearity of this simple family of monoids might pose a problem for that.

In some sense you could use a much more general function, such as secure hash functions using the Merkle–Damgård construction, but it's really desirable to use sets that have some kind of constant-size representation that can be efficiently composed. The whole idea of the Merkle–Damgård construction is to limit that kind of thing. Infogulch suggests that random invertible matrices over finite fields (Julia notebook) might provide an answer, but others suggest that it would probably be easy to break:

> Consider an ordered sequence of elements a_n, a function h that derives an invertible matrix over finite field $_{256}$ from a single element's cryptographic hash, and a function that finds the product of all such matrices from a sequence:
>
> $(a)=\prod^{n}_{i=1} h(a_i)$
>
> Define (a) to be the hash of the sequence a_n.

Rope append algorithm

So, to concatenate two ropes in this scheme, you first compute the hash of the concatenation, then check to see if a rope with that hash already exists. If so, you must check to see if it is actually equal; in most cases, this will be rapid, because you won't have to descend very far down the respective trees to decompose them into nodes of the same size, which can simply be compared by pointer. If it is actually equal, you can simply return the existing rope; otherwise, you must allocate the new concatenation.

An alternative approach to checking for equality in such a candidate-equality case is to slice the candidate new rope into new candidate ropes. If the existing rope is (A || B) and the new candidate rope is (C || D), then either #A == #C, #A < #C, or #A > #C. If #A == #C it is sufficient to compare the identities of A and C, and B and D. If #A < #C, then we can continue by constructing C[:#A] and comparing its identity to A's; if they are equal, then we construct C[#A:] || D and compare it in the same way to D. Analogously, when #A > #C, we can check whether C || D[:#A-#C] would be equal to A, and if so, check whether D[#A-#C:] would be equal to B.

This would seem to pose a troubling problem of infinite regress: to construct a new rope, we must apparently construct three more new ropes, and, worse, these extra new ropes might sit around in the hash cache and waste space until they get garbage collected. I think that's one way to solve the problem, but maybe a better way is to "proto-construct" these ropes but not stick them into the table; I think this reduces to the approach of traversing both trees in parallel.

In the usual case, if we approximate the hashes as random, the probability of a hash collision is on the order of 2^{-100} or lower, so, with a reasonably good hash function, such double-checking deep comparisons will almost always return true; new strings will almost always have new hashes. Even if the root has an equal hash, its children almost certainly won't (except in the case of an intentional attack). So maybe it isn't worth worrying too much about how many ropes get allocated in those cases, except to bound the cost to logarithmic.

Here, I'm supposing that each concatenation node includes a left pointer, a right pointer, two 64-bit words of hash, and a length field, so about 5 words, 40 bytes. On a 32-bit machine maybe it would be 20 bytes. You might need another word if you can't steal a bit for a type field somewhere, either in the node pointer or from one of the words.

Input splitting and leafnode coalescing

When importing bytestrings into the hash-consed-rope system, you can't get any storage sharing if you just import them as single leafnodes. You could imagine, for example, importing two versions of this Markdown file into the system as two strings, one before and one after inserting a blank line at the beginning. Ideally you would like these two versions to share most of their storage.

Now, if you construct the second version within the system, by appending the old version to a newline character, then you get storage sharing automatically, like the Ent of Xanadu. But what if you don't? What if someone sends you the second version over a network?

Jumprope and bupsplit have an answer here: run a rolling hash over the file (a hash like the CRC, that's not just monoidal but has an inverse), and use some criterion on the rolling hash to pick out a hierarchy of special cut points determined by their surroundings.

If you have working hash consing of ropes, though, I don't think that's necessary; you just need some way of splitting big input blobs into leafnode chunks of about 64–512 bytes that tends to reproduce the same cut points when it encounters the same data in different contexts. After that, the size of the rope tree internal nodes is going to be relatively small, maybe 20% of the size of the leaf nodes, and if you build it in a relatively random and balanced way, you'll probably be able to share a fair bit of that, too.

One simple approach is to start with a counter at 576 and decrement it every byte, then split and reset the counter when encountering a byte whose value is more than half of the counter value. So the byte 'a', whose value is 97, will start a new block if the counter is less than 194, setting the counter back to 576. This clearly guarantees chunks of 64–576 bytes, and it will clearly stay resynchronized if it ever manages to split in the same place (which is likely), but the average block size will depend on the data — if it's all zero bytes, it'll be 576 bytes per block, and if it's all 0xff bytes, it'll be 64. In Python:

```
def split(infile, maxsize=576):
    i = maxsize
    buf = []
    while True:
        c = infile.read(1)
        if not c:
            yield ''.join(buf)
            break

        i -= 1
        if ord(c) * 2 > i:
            i = maxsize
```

```
            yield ''.join(buf)
            buf = []

        buf.append(c)
```

You could take some steps to whiten the distribution a bit; for example, you could use the sum of all the bytes in the block so far mod 256, which will still have a much higher probability of creating a break at a 'y' than at a '!', or you could use the sums or bitwise ANDs of pairs of adjacent bytes. All such measures are necessarily trading off consistency of split location against consistency of split frequency.

On similar economic considerations, when you're concatenating small ropes, say less than 64 bytes, it's probably worth it to just copy the bytes into a new leafnode. This means that if you're typing into a hash-consing-rope editor, you'll be allocating a new leafnode of 16–72 bytes on every keystroke (averaging 44 bytes), which is clearly 44 times slower than it needs to be, but still better than allocating a 16-byte leafnode *and* a 40-byte concatenation node. (Of course you also need to allocate another couple of concatenation nodes that join the new character to the previous and following parts of the editor buffer, but that's true whether the keystroke is in a leafnode of its own or not.)

Along the same lines, it's probably good to coalesce concatenation nodes into a B-tree to some degree; a 4-child concatenation node would have a length, two words of hash, and 4 child pointers, 7 words. If you built it out of binary concatenation nodes, you'd need three 5-word concatenation nodes, 15 words, more than twice as much space.

Topics

- Programming (p. 1141) (49 notes)
- Performance (p. 1155) (22 notes)
- Algorithms (p. 1163) (14 notes)
- Real time (p. 1195) (7 notes)
- Hashing (p. 1293) (3 notes)
- Ropes (p. 1333) (2 notes)

Compilation of block arguments to high-performance code

Kragen Javier Sitaker, 02021-07-29 (updated 02021-12-30)
(19 minutes)

What would a calling convention optimized for block arguments look like?

Block arguments

Smalltalk and PostScript use block arguments for all their control structures, though Smalltalk cheats a bit on this. This makes user-defined control structures first-class. Ruby method calls can include a “block argument” after the other arguments, to which you can “yield” once or many times within the method, for example to automatically clean up after running some code, or to provide iteration over a data structure, echoing CLU’s iterator construct:

```
irb(main):012:1* def xs(y)
irb(main):013:1*   yield y + 1
irb(main):014:1*   yield y + 2
irb(main):015:0> end
=> :xs
irb(main):016:0> xs(3) { |z| puts(z) }
4
5
=> nil
```

This sort of makes the caller and callee into coroutines rather than subroutines. Python’s generator construct, adapted from a more general construct in Icon and now adopted by JS as well, provides a similar iteration facility.

Tcl procedures can evaluate an argument in the context of the caller, which is how Tcl control structures work, making user-defined control structures first class in Tcl as well.

ALGOL-60 provided call-by-name parameters allowing the callee to assign to an arbitrary expression in the caller; this, it turned out, required thunks to implement correctly and with reasonable efficiency.

Pascal permits passing nested subroutines as parameters, and these nested subroutines had access to the variables of the subroutine they were nested within. Though more awkward, this has the same sort of power as block arguments in Smalltalk, PostScript, Ruby, and Tcl. To preserve stack discipline in memory management, subroutines cannot be returned or stored in variables or fields, only passed. Such “downward funargs”, rather than unrestricted closures, were also the only kind of closures implemented in old dynamically-scoped Lisps; dynamic scoping did not provide real closures. They provide much of the power of real closures.

Aside from the well-known use of this for iteration, my notes on

IMGUI languages suggests that there is great utility in block arguments and at least pass-by-reference parameters, which cannot be implemented in Lisp-family languages like Python, PostScript, Smalltalk, and Ruby:

```
text_field("First name", &firstname, &firstname_len);
button("Submit") { send_form(); }
```

And of course it's common for control structures to want to assign to local variables in the caller, which in Ruby and Smalltalk is usually handled with block arguments:

```
mydict.each(&k, &v) { print "$k: $v"; }
```

So, how efficiently can we implement this kind of thing?

Calling conventions

The usual calling convention establishes where arguments are passed and divides registers into "caller-saved" and callee-saved. When a callee returns, it's entitled to have clobbered all the caller-saved registers it pleases, generally including all the arguments, but must have restored the callee-saved registers (on amd64, that's %rbx, %rsp, %rbp, %r12, %r13, %r14, and %r15; on RISC-V, according to both the user-level ISA manual and the ELF psABI, that's x2 (sp), x3 (gp), x4 (tp), x8 (s0/fp), x9 (s1), x18–x27 (s2–s11), f8–f9 (fs0–fs1), and f18–f27 (fs2–11)) to their values on entry.

On RISC-V, interestingly, tp and gp are supposed to be preserved even for signal handlers, so the callee can't even temporarily use them for something else unless it's willing to disable signals.

There are tradeoffs in how many registers you preserve across calls.

Generally, if you're going to support recursive functions, you need to at least preserve a stack pointer or a frame pointer; otherwise the function you're returning to has no way to find its return address or other local variables. And making very rarely changed registers like a thread pointer or global pointer callee-saved is basically free: since callees won't normally change them, preserving them incurs no cost.

Generally the caller doesn't really save the caller-saved registers. A better term might be "scratch registers": any result you compute in them must be consumed or stored somewhere stable before any call. That might be, as the "caller-saved" term suggests, by pushing it on the stack or storing it into the stack frame, but there's no particular reason to restore it to the same register again afterwards, or indeed any register; you might also have stored it into a data structure or used it as a pointer or an arithmetic operand.

You could argue about whether argument registers are copy-in-copy-out or pass-by-reference, but at any rate the caller is able to see whatever changes the callee made to them. Sometimes this may be true of arguments passed on the stack (at least if the callee isn't responsible for popping them) and aggregate return values are commonly provided by allocating space for them on the stack.

I don't have a great handle on the performance tradeoffs involved. If a function uses a callee-saved register, it must save it whether or not

its caller was actually using it; if it uses a scratch register for a value it wants to preserve across a call, it must save it whether or not its callee was actually going to clobber it. Having only a few callee-saved registers makes context switching (including function calls) faster, because context switching is precisely a question of saving and restoring the callee-saved registers in a way that violates stack discipline. Having many callee-saved registers makes leaf functions slower, because they have to save and restore however many callee-saved registers they use, but never have to save and restore scratch registers.

On the other hand, having enough callee-saved registers makes higher-level functions more convenient to write, and possibly faster, since it doesn't have to pay the cost of saving those registers unless it calls a callee that uses them.

Jeremiah Orians tells me that he finds it much less cognitive overhead to make all the registers, except of course return-value registers, callee-saved.

Different ways to represent closures or blocks

Indirect-threaded (ITC) Forth normally does a function call as explained in Moving Forth by the following sequence (my syntax, not his):

```
w := *pc  -- load virtual machine instruction (xt) into W
pc++      -- increment program counter
x := *w   -- load address of machine code from where xt pointed
goto x    -- invoke code
```

Note that this leaves the execution token in the W register. So, for example, all colon definitions share the same machine code (their first cell is a pointer to DOCOL) but have different data, and by incrementing W they can access that data. So this double indirection gives you a sort of automatic closure: the invocation sequence passes the code an address from which its code pointer was loaded, where it can find whatever other data it's interested in. Rodriguez explains, "CODE words [primitives] don't need this information, but all other kinds of Forth words do."

He points out that direct threading, where instead of putting a pointer to DOCOL you put a call or jump to DOCOL at the beginning of each word, is a better option on most machines:

```
w := *pc
pc++
goto w
```

Note that the execution token is still in W, so the instruction being jumped to can be simply a jump; it doesn't have to be a call. (Rodriguez attributes this discovery to Frank Sergeant's Pygmy Forth.) This is a faster way to invoke primitives ("CODE words") and generally no worse for other cases, though it involves a mixing of executable code and writable data that is difficult in many contexts

(Harvard machines, operating systems hardened with W^X).

But suppose you want to pass CLU-style iterators, Ruby-like block arguments, or ALGOL-60-style call-by-name thunks to a subroutine. These are also closures, but normally the data they want access to is in the lexically enclosing stack frame. GCC handles this by building a trampoline on the stack which sets the context pointer register before jumping to the implementation code. If you wanted to use the Forth approaches to this, you could imagine representing an ITC closure as a two-word struct:

```
struct closure { void (*c)(); void *d; };
```

If this struct is passed by reference, invoking `c` from it involves assembly code something like this:

```
mov -24(%rbp), %rax    # load struct pointer into %rax
mov (%rax), %rcx       # load code pointer into %rcx
call *%rcx
```

Then, *if* this struct were passed in memory (rather than in two registers), and *if* the pointer to it can be guaranteed to always be in a known register (in this case %rax), then the callee can access `d` as something like `8(%rax)`, just like DOCOL in an indirect-threaded Forth.

The direct-threading thing would amount to dynamically generating a jump instruction instead of a code pointer, and as with GCC's trampolines, would allow the invoker of the block argument to not indulge in an extra level of indirection in case it's invoking a closure.

However, this requires the caller to build a closure structure in memory for each thunk argument or block argument. This requires storing a code pointer in memory, which is a huge pain on amd64. If you have three such arguments, that might be something like this (in the ITC-like case; the DTC-like case is just slightly messier):

```
    movabsq $foo, %rdi      # build first closure
    mov %rdi, -24(%rbp)
    mov %rbp, -16(%rbp)
    movabsq $bar, %rdi      # build second closure
    mov %rdi, -40(%rbp)
    mov %rbp, -32(%rbp)
    movabsq $baz, %rdi      # build third closure
    mov %rdi, -56(%rbp)
    mov %rbp, -48(%rbp)
    lea -24(%rbp), %rdi     # pass first closure argument
    lea -40(%rbp), %rsi
    lea -56(%rbp), %rdx
    call quux
```

It's not *too* terribad how the callee invokes one of these (remember that this is a non-standard calling convention that guarantees the closure pointer to be in %rax):

```
mov %rsi, %rax          # invoke second closure argument; realistically this would probably be in the stack frame
mov (%rax), %rcx
call *%rcx
```

If we were to use the standard C calling convention and the usual userdata `(*k->c)(k->d)` approach, closure invocation is instead something like:

```
mov %rsi, %rcx
mov 8(%rcx), %rdi    # first parameter to closure
mov (%rcx), %rcx
call *%rcx
```

Despite all this hassle, the code for each of these closures needs an additional instruction to load its context pointer:

```
mov 8(%rax), %rcx
```

Or three, if it wants to use %rbp as its context pointer.

A calling convention that makes block arguments efficient

For the specific case of thunks and block arguments, where the context pointer is the parent's call frame, it would be much nicer to be able to just pass the code pointers and have the caller's frame pointer just implicitly flow through to the block arguments. Then the initial call sequence would look like this:

```
movabsq $foo, %rdi
movabsq $bar, %rsi
movabsq $baz, %rdx
call quux
```

The block/thunk invocation would just look like this, at least if no registers need to be saved or restored first:

```
call *%rsi
```

Then, within the block or thunk, %rbp is just available as a frame pointer as normal.

In the case where registers do need saving/restoring, for example because the callee `quux` was also using %rbp, it might look like this:

```
push %rbp
mov -8(%rbp), %rbp
call *%rsi
pop %rbp
```

Normally you might expect this to index off the stack pointer or the frame pointer instead of using push/pop instructions, but the

whole point of block arguments is to allow the stack pointer to be arbitrarily far away from the stack frame of the current lexical context, by allowing callees to yield back into callers. (Probably this whole thing would be totally impossible on something like a SPARC.)

This suggests dividing "callee-saved" registers into thunk-preserved registers, %rbp in the above example, which the callee is obligated to restore to their caller's values upon entry to any caller-provided thunk, and return-restored registers, which are guaranteed to be restored to their original value when the callee returns but may have different values when thunks are being run (notably, %rsp).

This kind of thing makes it practical to implement control structures as library functions. To take an extreme example that will obviously frustrate desirable optimizations:

```
while (*s) { *t++ = *s++; }
```

This might be compiled, using a library `while` function, as something like the following:

```
        movabsq $thunk_1, %rdi
        movabsq $thunk_2, %rsi
        call while
        ret
thunk_1:
        mov -8(%rbp), %rax      # s
        mov (%rax), %cl
        xor %rax, %rax
        test %cl, %cl
        setnz %al
        ret
thunk_2:
        mov -8(%rbp), %rax      # s
        mov -16(%rbp), %rcx     # t
        mov (%rax), %rdx
        mov %rdx, (%rcx)
        incq -8(%rbp)
        incq -16(%rbp)
        ret
```

I mean, this is obviously not a good way to compile non-NUL-terminating-strcpy, but at least so far the overhead is not ridiculous. `while` itself might be implemented as follows:

```
while:  push %rsi               # thunks can hork %rsi
        push %rdi               # and %rdi
        push %rbx               # but not %rbx
        mov %rsp, %rbx
1:      mov 16(%rbx), %rsi      # load first thunk
        call *%rsi
        test %rax, %rax
        jz 2f
        mov 8(%rbx), %rdi
```

```
        call *%rdi
        jmp 1b
2:      pop %rbx
        retq $16
```

We could improve this a bit by guaranteeing %r12, %r13, %r14, and %r15 to the thunks as well as %rbp. This would require no changes to our `while` but would allow us to rewrite our thunks:

```
thunk_1:
        mov (%r12), %cl         # s
        xor %rax, %rax
        test %cl, %cl
        setnz %al
        ret
thunk_2:
        mov (%r12), %rdx
        mov %rdx, (%r13)        # t
        inc %r12
        inc %r13
        ret
```

Suppose we decide to guarantee that thunks preserve %r8 and %r9 (normally caller-saved argument registers, which is to say, the callee owns them; here we're extending the callee's ownership over them to require blocks in the caller to preserve them while the callee is yielded). Then we can improve `while`, rearranging it a bit as well:

```
while:  mov %rsi, %r8           # save first thunk
        mov %rdi, %r9           # save second thunk
        jmp 1f
2:      call *%r9
1:      call *%r8
        test %rax, %rax
        jnz 2b
        retq
```

This amounts to 14 instructions per loop. GCC -O is emitting this code for me for the real C version:

```
    endbr64
    movq    %rsi, %rax
    movzbl  (%rdi), %edx      # s
    testb   %dl, %dl
    je  .L2
.L3:
    addq    $1, %rdi
    addq    $1, %rax
    movb    %dl, -1(%rax)     # t
    movzbl  (%rdi), %edx
    testb   %dl, %dl
    jne .L3
.L2:
    ret
```

This is 6 instructions per loop, so we can crudely guess that the overhead introduced by this coroutine mechanism is around a factor of 2–3 in this sort of worst-case scenario.

So instead of dividing registers into caller-saved and callee-saved, we divide registers into four groups: caller-saved that thunks must preserve, caller-saved that thunks can clobber, callee-saved that thunks can rely on, and callee-saved that thunks must preserve (but cannot rely on).

You can simplify this by eliminating the "callee-saved that thunks must preserve" category. Or complexify it by by having registers, like sp and RISC-V's tp and gp, which thunks can rely on, but must preserve.

One upside of just using ordinary closures for this kind of thing is that, even though it takes four instructions to construct and pass the closure, three or four instructions to invoke it, and an extra context-pointer instruction within the closure, you can pass it down through any number of levels of calls with just the usual single instruction needed to pass on an argument to a callee, and invocations from however deep down the stack are always equally efficient. With this coroutine mechanism, by contrast, if you want to pass on a thunk you received to a callee, you can't; the best you can do is make a thunk of your own that invokes your thunk.

Bounding stack depth

This kind of mechanism is useful in cases where you want to avoid garbage collection, either for micro-efficiency or to avoid possible failures. In the cases where you care about possible failures, it's important to be able to statically bound the stack depth. In the normal kind of C programming, which doesn't have block arguments, you would do this by just making a call graph, which would need to be acyclic, and computing the highest-weight path from the root to a leaf, where the weight of each node is the size of its stack frame.

Actually, a better way to do this is to compute the maximum stack size of each function as its stack frame size plus the maximum of the maximum stack sizes of its callees, rather than explicitly constructing the graph, but conceptually it's the same thing.

That runs into difficulties with function pointers, where you have to conservatively approximate the set of functions that can actually be called from a given callsite, for example using the function type. This can very easily give you unwanted recursion.

I think this block-argument thing is more tractable to make fairly precise, because when a function invokes a block argument, you know it can only be one of the block arguments it was actually passed.

Now each subroutine S has not only a maximum stack size of its own M[S], but also a maximum stack size M[S, P] with respect to each of its block formal parameters P, which expresses the maximum size of the stack of that function and its callees when that block formal parameter is invoked. If it only invokes the block formal parameter

directly (not from a block nested inside it and passed to some other function) its stack size for that parameter is just its own stack frame size F[S]. But if it invokes block formal parameter P inside some block that it passes to subroutine T as the actual parameter for block formal parameter Q, then its maximum stack size M[S, P] for that parameter is at least M[T, Q] + F[S]. So M[S, P] = F[S] + max(i, M[Ti, Qi]) for all the callsites (Ti, Qi) for the block formal parameter P. And M[S], which will be at least at large as any of the M[S, Pi] for subroutine S, is F[S] + some other maximum stack size.

Topics

- Programming (p. 1141) (49 notes)
- Performance (p. 1155) (22 notes)
- Safe programming languages (p. 1172) (11 notes)
- Assembly-language programming (p. 1175) (11 notes)
- Compilers (p. 1178) (10 notes)
- Higher order programming (p. 1196) (7 notes)
- FORTH (p. 1231) (5 notes)
- Pascal (p. 1247) (4 notes)
- RISC-V (p. 1276) (3 notes)
- Tcl (p. 1318) (2 notes)
- Smalltalk (p. 1326) (2 notes)
- Call by name (p. 1382) (2 notes)
- Block arguments (p. 1383) (2 notes)
- PostScript

Material observations

Kragen Javier Sitaker, 02021-07-29 (updated 02021-12-30)
(194 minutes)

Some notes on some things I tried.

02021-07-28

Pouring sodium silicate solution into concentrated oil of lime hardens the surface enough that it doesn't stick to the bottom of the yogurt cup, and you can pick up the resulting ooze by hand. Placing the ooze on a bed of vermiculite and pouring hot construction sand (300°?) over it produces foamed glass, which grows over a longer period of time than the foamed glass produced from dried waterglass, but with about the same density. However, it sticks to the sand and vermiculite as it grows.

Having burned through a cotton towel when pouring hot sand previously, I used a steel wool pad as a hotpad in this case, which worked fine. Obviously there are drawbacks to using such inflammable materials as insulating refractory, but in this case it didn't ignite, and it worked fine to protect my fingers.

Having melted the aluminum parts of the previous butane-torch nozzle (the kind that uses 227-gram cans of butane), I bought a somewhat more robust one from the hardware store for US$6, with no evident aluminum or plastic parts and no striker. This can produce a substantially more exciting flame.

Borax

Bought some 700 g borax from the hardware store for some US$2. Smells slightly acidic, suggesting it's maybe contaminated with boric acid. It successfully cross-links PVA glue ("plasticola") into an insoluble mass. A hot wire causes it to adhere readily; application of heat causes it to foam up quite a bit, suggesting that it's quite hydrated, and then collapse down into a glassy layer of, presumably, mostly boria, which seems to be able to mostly dissolve construction sand when reheated. This suggests that it really is borax.

Heating the borax on low heat on the stove on some aluminum foil produces a prodigious amount of white solid foam, similar to the intumescent foam from sodium silicate (see Glass foam (p. 595)), which can be cut with a hot wire. A drop of water on this foam has no immediately obvious effect, but after a few minutes, has dissolved a hole some 30 mm in diameter.

Quartz-halogen bulbs

The internal bulb in one of these halogen household bulbs can be heated to an orange heat in the butane torch without any softening, and even quenching in water does not crack them. This strongly suggests that it really is quartz. Too bad it's burned out. These bulbs are hard to find nowadays, though some vendors have some remaining stock for under US$1 each, and 150W–500W exterior halogen floodlights still cost only US$2–US$4.

The replacement headlight bulb I got from an auto parts store (US$3) has the same behavior, except that I haven't tried quenching it, suggesting that it's also a quartz-halogen bulb. (Also, the box carries a pictorial warning against touching the glass.) This is a Magneti Marelli "H7 12V 55W PX26d". Cold (10°?) it measures some 7Ω on a shitty multimeter that measures 3Ω short-circuit, demonstrating why household 240V quartz-halogen bulbs would be better for use as RTDs. When reheated to orange-hot, it measures 2Ω but the meter's short-circuit resistance has dropped to 1Ω, demonstrating that the meter is totally inadequate for this purpose. For heating-element purposes such a bulb ought to be adequate up to nearly 1000°.

Pumice and garden stores

I bought 10 liters of off-white pumice from the garden store for US$2. It's so porous that it floats, but it's open-cell enough that sometimes it becomes waterlogged and sinks after a few seconds. It seems to weigh about 3.5 kg. Stone dimensions are on the order of 10–40 mm. It can be heated to orange heat in the blowtorch without apparent change. A hacksaw cuts it almost as easily as wood; a steel wire can bore a hole through it far more easily than through wood, and without cracking it in half. It can be easily sanded on granite or on other pumice stones, yielding a fine floury dust.

Though there is significant variation, it's too hard to crush between my fingers, break with my fingers, or carve with thumbnails in its default form, though you can drill through it with a bamboo chopstick and enough effort. However, heating it to orange heat in the blowtorch renders it somewhat more fragile (presumably by thermal shock inducing microfractures), so that it can be broken by hand, though still not crushed between my fingers. Simply soaking it in water does not have such a weakening effect, but quenching it in water after such heating weakens the stone to an even greater degree.

One stone weighs 6.7 g. When added to 151.3 g of tap water (including yogurt cup), the reading was 158.1, a reading of 6.8 g. Later the reading was 157.9, and upon depressing it below the surface of the water with wires, 159.1, suggesting 1.2 cc of volume out of the water, or some 7.9 cc of total volume. Upon removing the wet stone from the water, the reading dropped from 158.0 to 150.2, a difference of 7.8 g, and the wet stone weighed 7.7 g, suggesting there was 1.05 g of water in its pore volume. This suggests a total volume, including pore volume around 9.0 cc and thus a dry density of 0.74 g/cc.

I stopped by a grow shop and was disappointed to find very little in the way of pure fertilizers and pesticides, less even than in the local garden store.

Carbonates and phosphates of iron

The garden-store green vitriol looks quite pure, though clumping together like a gorgeous green deliquescent thing, but the solution in tap water is a bit cloudy. Adding an excess of baking soda produces, after a while, voluminous green-brown bubbles that grow quite large and do not pop, overflowing my inadequate yogurt cup. A gray-green gritty slime with a smell like metal precipitates; this is probably a mix of carbonates and hydroxides of iron. Effervescence

continues for an hour or more; probably boiling the soda first would have avoided it. Adding diammonium phosphate fertilizer instead in solid form produces no visible reaction at first, but upon heating produces a milky white precipitate, which later proceeds to form convection cells like those in miso soup.

02021-07-29

Iron products

Neither of the insoluble iron preparations have defecated to anything like transparency. Gravity filtration of the carbonate through a coffee filter leaves a dark green-brown mud on the filter and a fairly cloudy filtrate, which defecates. The mud smells like sulfur or metal, but the filtrate smells like dust. A second filtration still leaves a cloudy filtrate.

The phosphate is instead partly white, partly gray, with some solid chunks which presumably result from the green vitriol remineralizing the fertilizer prills without letting them completely dissolve. It has a similar metallic/sulfurous smell, and similarly the filtrate through coffee filters is cloudy. No ammonia smell is detectable. Excitingly, some of the phosphate product required a bit of work with a chopstick to scrape free of the epoxy (?) liner of the aluminum can. But this may have been because I overheated it when heating the mixture. I should maybe recrystallize some of the ammonium phosphate and figure out some of the stoichiometry.

Pumice grinding

Grinding the surface of the pumice with, for example, bamboo, tends to make the surface harder and less porous, because it fills the exposed bubbles with packed powder. When a hole thus drilled with a bamboo chopstick has recently broken through, there is greater risk of breaking the rock by wedging.

Vendors

Nearby vendors selling muriate of lime and similar materials include Kubra Quimica (JUANMANUELCURRA) in Haedo, ROAL4512404 in Ramos Mejía (aimed at making cheeses and yogurts), DUROPAVIMENTO in Ciudadela (aimed at making cheeses), and NAMECO QUIMICA in Villa Bosch (Kaiser 921, 5 km north, doesn't list products on website), Química Kraff (KRAFF QUIMICA) in Morón (Av Estanislao Zeballos 1820, "cerca de la cancha del club dep morón"), Insumos Nahum in Ciudadela, ACUARIO MAKARIOS in El Palomar (Sargento Cabral 1205? El Rodeo 1416?), Isiquim SA (El Rodeo 1355 in El Palomar), Laboratorios Condisal SA in Lomas del Mirador (Tapalqué 543), Planeta Verde (TIENDAONLINEPLANETAVERDE) in Tapiales, maybe Distriquím (Soldado Caballero 8664?), Biosix SA (S. Ortiz 2498, piso 2, Ramos Mejía, surely the closest, doesn't list products on website), Victor Eduardo Simo S.A. at Carlos Tejedor 5345 in Caseros (4750 6644, no website, Monica Barraza?, magnesium sulfate and fluorescein and phosphoric acid), Productos Químicos Alcesa) in Haedo (Paraguay 939), Silap (Sialp? Bacterint?) in Haedo (San Luis 661), Astra SA in Haedo (Argerich 536), Inquimec S.R.L in Haedo

(Inalican 1154), Guillermo A. Rodríguez (Transquimia Chemist Cia Química) in Haedo (La Fraternidad 590), Ferar Química (B. Márquez 1235, Loma Hermosa), Química Morón (Hipólito Yrigoyen 1227, Morón), Productos Químicos SRL in Morón (H. Yrigoyen 625). Also there's an Easy at Juan Bautista Alberdi 4950 in Caseros.

02021-07-30

Muriate of lime

Put 108.0 g of water in a 10.9 g cut-off can and started heating it up gently. Too bad I don't have a thermometer. After heating to boiling it weighed 117.8 g. Adding (hopefully anhydrous) muriate of lime it had all dissolved up to a total weight of 235 g, so the amount of muriate was also 117 g. At 282.0 g total weight some of the crystals remained undissolved, but then did dissolve, with a weight gain to 287.6 g. I added more crystals to 322.1 g and a little bit seems undissolved; gentle heating dissolved them, reducing the weight to 321.9 g. Adding more crystals to get up to 341.9 g leaves some crystals undissolved. Further gentle heating leaves weight at 341.6 and dissolves all crystals; adding more crystals to 366.0 g leaves some crystals undissolved at first. Further gentle heating dissolves all crystals and leaves the weight at 364.1 g; adding more crystals brings the weight to 387.5 g. Further gentle heating dissolves all crystals and leaves the weight at 385.3 g; I added more crystals to 442.2 g.

And then I gave up. I think those dissolved too. I'm letting it cool now; a thin crust of solid has formed at the surface, preventing further evaporation.

After a couple of hours, the material in the can has the appearance of a solid, transparent block, but this is an illusion; once the crust is broken, the inside can be seen to be a mix of rather jagged crystals of muriate of lime and warm, syrupy oil of lime, which stick nicely to the finger.

NightHawkInLight's intumescent refractory

Now I have borax so I should be able to make resistant carbonizing intumescent refractory materials. One person reports success with 10 tsp cornstarch, 1 tsp baking soda, 4 tsp PVA glue, and no extra water. NightHawkInLight is reported by WP to have switched from this recipe to cornstarch, flour, sugar, and borax, which was less prone to mold; in his video on the improved formulation he explains (3'47"):

> I've also found an optional ingredient that adds both mold and insect resistance, as well as an improvement I did *not* expect, which I'll talk about in a moment. These are the ingredients i used in my improved composition: 40 grams of flour, then 20 grams each of corn starch, powdered sugar, and baking soda, combining this together with approximately 25 grams of water creates a workable dough.
>
> It may seem like a lousy cookie recipe, but each of these ingredients serves a specific and important purpose. The flour is my new binder which holds all the rest of the ingredients together. Cornstarch reduces stickiness and allows the putty to better hold its shape. And sugar is the secret to generating a carbon foam even when the resulting material is bone dry.
>
> When heat is applied to my composition, the sugar inside will melt, providing elasticity and lubrication between the other chemicals so that gas bubbles can form. Baking soda is the final ingredient, which, when heated, releases CO_2 and water, which inflate the carbon bubbles and give us our final result: an extremely heat resistant putty which can be used as is or dried into solid tiles.

… The most obvious weakness to this composition is that it's very edible, both by insects and by mold, so I decided to try adding borax to the recipe. Borax kills most insects that would eat it, and also kills mold. Similar to baking soda, it releases water vapor as gas when heated, so borax could replace baking soda in my formula entirely.

At this point he shows his notebook with something like eight alternative recipes:

• 5 g baking soda, 10 g borax, 10 g sugar, 5 g corn starch, 25 g flour, 5 g diatomaceous earth;
• 50 g corn starch, 25 g erythritol, 25 g ammonium diphosphate [sic] + shellac as a possible coating;
• 10 g sugar, 20 g flour, 10 g borax, and "mod + 5 g baking soda". A second column says that using 10 g flour instead of 20, it was "a little melty".
• On an earlier page, .5 g alum, 2 g iron phos[phate], 2.5 g flour; or
• 3 g iron phos[phate], 1 g urea, 3 g alum, 1 g "cs" (cornstarch?), 16 g flour, 2 g sugar; marked "almost as good →" with an arrow pointing to:
• "Best": 10 g corn starch, 20 g flour, 10 g baking soda (or borax, as suggested by a video edit), 10 g powdered sugar, 25 g water for 2× batch; one Xed out column instead suggests 10 g corn starch, 50 g flour, 20 g baking soda, 20 g powdered sugar; a third column suggests 5 g corn starch, 25 g flour, 10 g baking soda, and 10 g powdered sugar. And, finally,
• 1 g "Kno3" (!), 2 g iron II phosphate, 10 g corn starch, 20 g flour, 10 g sugar, 20 g water.

The resulting carbon foam [using borax] is slightly less insulating to heat, but the added resistance to rot and insects may be worthwhile. One discovery that surprised me in my tests with borax was that the carbon foam generated by a borax-containing composition is significantly stronger physically than any foam I had generated before. The hardness, if you can believe it, is about the same as an alumina fire brick, which may open some new opportunities for how these normally very fragile carbon foams could be used.

So, I guess I'll try his recommended recipe: 40 g flour, 20 g corn starch, 20 g sugar, 20 g borax, and 25 g water.

I mixed:

• 40.2 g 000 white flour (Cañuelas brand)
• 20.4 g borax
• 20.0 g sugar (granulated, Ledesma azúcar común tipo "A", freely flowing and so apparently fairly dry)
• 20.4 g manioc starch (I couldn't find the cornstarch)
• 37.4 g water (I poured too fast and overshot by 50%)

The total ball of dough was 133.9 g, so I probably lost about 4.5 g that stuck to the mixing container or my hands or whatever. It smelled like cookie dough and had a tendency to kind of slump, probably due to having too much water. I was able to use it to cover the inside of a Monster can with the ends cut off and heat the can from the inside with the hand butane torch, and it did indeed foam up and produce an aroma similar to burnt toast and a carbon foam similar in hardness to insulating firebrick.

After letting the can cool, I turned the flame back on and ran it through the can again while holding it between my fingers; after

about 30 seconds, the can became too hot to hold comfortably.

I heated up a small, thin piece (initially on a bed of waterglass foam, then inside the Monster-can "forge") to see if I could cook it thoroughly into foam to get some kind of read on its material properties. During this process I noticed a second objectionable feature of this composition, aside from the burnt-toast smoke: it has a tendency to continue to smolder for a while after the source of heat is removed. (I wrapped the piece in aluminum foil to put it out without wetting it.) Also, significant cracks were opening up in the lining of the Monster can, although they seem to have been exposing uncooked dough rather than unprotected aluminum.

After all that, the piece turned out to be under 100 mg and so too small for my shitty scale to weigh, even when wet. It floated in water mostly below the surface, similar to the pumice, but it's hard to tell how much of that is a question of water absorption.

In terms of fragility, it is definitely less fragile than the waterglass foam, and definitely more fragile than pumice.

Heating some more of this stuff with he blowtorch on top of the waterglass foam, I notice that each little black particle has a little crater around it in the waterglass foam, like snow around de-icing salt on a sidewalk. This suggests that either the carbon particles are producing heat to melt the waterglass around them, for example by burning (quite possible), or that the borax in them is fluxing the waterglass foam and allowing it to soften and collapse at a lower temperature (also possible).

One of the resulting chunks of carbon foam ranges from 15 mm in thickness to 25 mm in length and weighs 1.0 g.

Density measurement: I placed a 64.5 g cup of water on the scale. Immersing the foam in it beneath the surface with chopsticks yielded a reading of 68.1 g; removing it produced a reading of 61.8 g; weighing the foam separately yielded 3.7; immersing it a second time produced a reading of 67.8; weighing it a second time produced a weight of 4.0; a final weighing of the cup produced a reading of 61.4. Apparently in the first immersion it absorbed 2.7 g of water (both the difference in the foam and in the water), having displaced 3.6 cc of water; after the second immersion it had absorbed 3.0–3.1 g of water and was displacing 3.3 cc of water. So its total volume was 6.3 cc, including both the pore spaces that filled with water and those that excluded water, but not including spaces from which the water drained immediately upon removal. This is a pretty reasonable volume, given that 15 mm × 20 mm × 25 mm would have been 7.5 cc, and the shape was quite irregular. So the density of the foam as such is about 0.16 g/cc.

The water in this process turned dirty brown and had lots of black crap floating in it, so presumably there was a significant amount of lost mass. On soaking the foam a little longer, the water turned quite black.

After driving the water off from the foam that hasn't dissolved with the torch, it weighed 800 mg, so only about 20% was lost.

I've flattened out a remaining piece of the dough to let it air-dry so that I can test its intumescent properties after drying.

Waterglass and starch intumescent refractory

It occurs to me that a 1:1 molar ratio of carbon sources such as cornstarch with silicon sources such as waterglass might be able to produce a foam that is partly or wholly carborundum (melting point 2830°), which would have mechanical and oxidation-resistance properties superior to carbon foam, and mechanical and melt-resistance properties superior to silicate glass foams. Boron carbide B_4C would probably be superior (harder than carborundum, softer only than CBN and diamond) but I think it's probably even harder to produce than carborundum; Moissan's original synthesis involved heating boria with carbon over the melting point of boron carbide, 2763°, though apparently magnesium permits the reaction under much less violent conditions.

Mixing a little waterglass with manioc starch produces a mixture with the consistency of warm chewing gum; I kept adding starch and kneading it in until it seemed like adding more starch would make it fall apart. (But I don't know whether this is 1:3, 1:1, 3:1, or what.) The resulting rubbery, doughy substance is intumescent under the flame, producing a black foam with similar strength to that of the borax-based recipe above, though perhaps with a smaller cell size, but, as I was hoping, this foam has no tendency to smolder. Also, perhaps unsurprisingly, it stuck rather tightly to the foamed waterglass bed I was trying to heat it on, which the carbon/borax foam did not. I've left a fragment of this composition also to dry. While intumescing it produced a scent vaguely reminiscent of pine pitch, which strikes me as slightly alarming.

Both this foam and the waterglass-free version can be very readily cut with a hacksaw.

Waterglass, borax, and starch intumescent refractory

It occurred to me that maybe borax would be compatible with waterglass, and indeed you can mix quite a lot of borax and water into this waterglass solution without coagulating it in the way calcium or magnesium ions do. (I was concerned that perhaps the borate ions would be sufficiently amphiphilic to cross-link the silicate, but apparently not.) Mixing up manioc starch as before with this mixture produces another white doughy mixture which produces intumescent black foam when heated — though there are also bits of white foam, which may have been undissolved borax crystals. I've left a couple of these samples also to dry and see how they behave. One I heated immediately; it also has the unnerving pine-resin smell, and the foam it produces is the hardest yet — I can barely carve it with my thumbnail. I ran out of butane for the torch in the process, so I heated the sample for a longer period of time on the kitchen stove's gas flame. Hacksawing through it reveals that although it has foamed up throughout its thickness, only the outermost 5 mm charred, leaving an inner circle of white uncharred material separated from the fissured outer carbon foam by a brown ring.

As with the waterglass/starch mix, the foam shows no tendency to smolder.

I can still break this foam material by hand, but it is hard enough that I cannot crush it between my fingers. Taking a 12-mm-diameter

bit of it that weighs in at 0.65 g, I tried repeating the earlier density measurement but screwed up my measurements; I think the water weighs 81.9 g after pulling the wet foam out, 83.6 with it immersed, and the wet foam weighs 1.6 g. Or maybe it's 80.7 g after the foam is removed. So it absorbed 0.95 g of water and displaced 1.7–2.9 g, which is really a damned wide range, so its volume including pores is 2.65–3.85 cc. Which is totally impossible, because $(12mm)^3$ = 1.728 cc, so it's somewhere in the range 0.17–0.72 g/cc. Not a good measurement by any stretch of the imagination. All you can say is that it floats if it's dry. It turns out that it floats if it's wet, too, barely, so I guess it's below 0.4 g/cc dry.

So, I'll have to redo the test with a larger piece, maybe after I get more butane.

The part of the foam that has been entirely charred is dramatically more fragile than the part that hasn't.

It's too bad I didn't weigh out the ingredients, because this formulation in particular would have been pretty great to be able to reproduce.

I think the torch I'm using doesn't get as hot as the torch I melted — it's more for heating larger areas — so I haven't been able to get any of these samples above orange heat. It'll be good to be able to verify that they don't lose structural integrity at higher temperatures due to, say, too much borax, too much waterglass, or too much thermal expansion.

The waterglass-based samples I've left to dry seem to be leaching waterglass out the bottom.

02021-07-31

The unfired waterglass bodies have hardened noticeably. Also, two of them have split, evidently from contracting as the surface dried.

I heated up one of them on the stove flame; it bubbled up pretty much as before.

Made a giant cookie using the flour/starch/sugar recipe, this time using baking soda instead of borax (of course), and adding some oil and vanilla. Also I added too much water, so I'm frying it in a skillet instead of baking it. First conclusion is that this is far too much leavening for a cookie or pancake. After cooking, it was even more inedible than beforehand; the surface caramelized to the point of being too bitter to eat. I discarded most of it, retaining a piece for future torch testing. Also, I keep belching from the uncooked batter I ate. The kitchen definitely smells more appetizing and less alarming now, though.

Grupo Ecoquimica at Avenida de Mayo 1761 in Ramos Mejía has zinc oxide (AR$4500/kg), epsom salt (AR$220/kg), mesh-80 calcium carbonate (AR$80/kg), borax (AR$220/kg), alum (AR$630/kg), citric acid (AR$290/kg), sulfur (AR$300/kg), bentonite (AR$50/kg), glycerin (AR$500/kg), kaolin (AR$150/kg), magnesium chloride (AR$210/kg), dipropylene glycol (AR$528/kg, see Dipropylene glycol (p. 687)), soy lecithin (AR$600/kg), cerium oxide (AR$13200/kg), boric acid (AR$275/kg), copper sulfate (AR$880/kg), titanium dioxide (AR$2100/kg), electrode gel

(AR$360/kg), talc (AR$200/kg), infusorial earth (AR$167/kg), etc. Generally they sell in units of around AR$300–AR$500, so for most materials the limiting factor would be how much I could carry. It's a few km away. Kubra and Insumos Nahum have propylene glycol, these guys don't, just dipropylene glycol.

Oh, and it looks like the place I bought muriate of lime at the beginning of the pandemic was Química Industrial Caseros, at Bartolomé Mitre 4405 in Caseros, which is at the intersection with Lisandro de la Torre, which intersects Alvear at Día, not where I had thought. So that's why I didn't find it the other day when I went walking up that way: I was a few blocks off.

02021-08-02

Bought a new can of butane/propane for AR$340 (US$1.90).

The dried inedible pancake fragment, with its extra sugar and oil, and with baking soda instead of borax, can not only smolder but sustain a flame for a while after the torch is removed. Also, it evidently softens enough to deflect noticeably while the torch is on it.

The other dried/hardened samples (waterglass with starch, waterglass with starch and borax, flour/sugar/starch/borax) all produce hard intumescent foams as before. The waterglass-based samples don't smolder.

The massive waterglass/starch/borax sample cracked in a few places on the outside as its crust shrank while drying.

02021-08-06

I ground up a spoonful of some diammonium phosphate fertilizer using scraps of granite as grindstones, noting an objectionable ammonia odor; sprinkled it onto some construction sand, previously dried with a torch, atop a bed of vermiculite; ground up a spoonful of what I believe to be some aluminum hydroxide, and sprinkled it on top; and heated the whole lot with the torch for a while. I never reached an orange heat for the whole mass, but the fertilizer did bubble out quite a bit of ammonia, and I often reached an orange heat for the surface. At one point I stopped, and the large amount of ammonia that began to escape persuaded me that stopping had been a bad idea, so I flambéd the mass a while longer until it stopped bubbling, about 10 or 20 minutes.

Upon cooling, I had a somewhat porous and brittle but surprisingly strong and hard crust of white material. It did not show any water-solubility or acid or base reaction (that is, upon dripping vinegar on it and letting the vinegar sit for a while, baking soda sprinkled on it would bubble, so it did not neutralize the vinegar; but upon being wetted with water, baking soda sprinkled on it would not bubble, so it did not neutralize the baking soda either.)

It's probably worth making some better aluminum hydroxide, and trying the three pairwise combinations of the fertilizer, the construction sand, and the aluminum hydroxide.

At this point, despite this process having produced some ammonia and presumably a significant quantity of nitrogen oxides, my lungs were feeling fine. I tried heating a piece of the aluminum phosphate

(?) crust thus formed by blowing the torch through the Monster-can “forge” with the borax/flour/etc. insulation described earlier to see if I could get it to a higher temperature and maybe melt it. I did get it to bubble a bit more, but it didn’t seem to melt bodily. After heating it in this way for several minutes, the intumescent insulation was smoldering pretty enthusiastically, and in places was peeling away from the can; after removing the gas flame, it sustained a flame for under a minute, then went back to smoldering. This left the house full of smoke that smelled like burnt toast and left my lungs feeling shitty.

I should probably use one of the non-smoldering, waterglass-based intumescent recipes.

02021-08-07

Still no water-solubility evident from the scrap of aluminum phosphate (?) in water.

Mina’s impression of the phosphate material was that it felt like a “fragment of wall”, that is, fully hardened portland-cement concrete that has exfoliated (maybe because it was improperly adhered stucco). That seems about right, but I think it might be slightly harder than that.

02021-08-08

I mixed some one-component hardware-store silicone (3M 420 *transparente, vidrios y ventanas*) with the borax, then mixed by hand, adding more borax until it seemed like it was going to fall apart, to get a translucent ball some 25 mm in diameter, around 00:40. I poked and prodded it to see if it had become elastic or was still just visco-.

The borax is, I think, too coarsely ground to keep the silicone from being sticky; so I got sticky silicone all over my fingers, despite initial attempts to avoid this by using polyethylene bags as gloves. At least I seem to have avoided getting major skin irritation from the acetic acid. I was hoping for more of an Oogoo-style effect, but I realize that I ① haven’t made Oogoo and ② probably would need to have the borax finely ground. The idea at any rate was that maybe the water of crystallization of the borax would serve as a hydroxyl source for the silicone polymerization, allowing it to harden all the way through like Oogoo, and then maybe if the resulting object was heated, the borate would form a borosilicate network with the silicone backbones of the polymers. That evidently didn’t happen.

At 01:00 the ball was still pretty visco- and not very elastic, but at least it wasn’t sticking to my fingers anymore. Likewise at 01:24. The temperature in here is probably around 15°. By 07:30 the ball was mostly returning to its initial shape after being squished, and no longer cracking apart. My impression is that this is very similar to the behavior it would have if I had mixed in sand instead of borax — if it didn’t stick to the sand at all.

(Unfortunately I didn’t bother to note whether it had an acetic-acid smell at the time.)

I’ve been thinking about how to make these tests more rigorous, having just read Kingery’s earthshaking dissertation on

phosphate-bonded refractories from 01950; right now I can't even reliably tell if one mix is twice as strong or weak as another, or contains twice as much sand. It would be great to be able to do quantitative compressive and flexural strength tests, for example, which would require being able to cast bars of reproducible dimensions and fire them at reproducible temperatures. The tests mentioned previously with the phosphate fertilizer involved training a somewhat inconstant butane torch on some material until it stopped bubbling, which took about 20 minutes.

Doing this in a consistent fashion is going to involve setting up some kind of kiln that can maintain a reproducible temperature, for probably considerably longer than 20 minutes, since the temperature has to penetrate to the center of the bar instead of just a millimeter-thick layer. See Pocket kiln (p. 704) for thoughts on this.

Also, it would probably be useful to pre-purify the phosphate fertilizer, maybe using that recrystallization protocol by that African research group. And maybe pre-dissociating it to get 99% crystalline phosphoric acid would be useful for non-intumescent recipes, and also eliminate the problems associated with ammonia emissions. The phosphoric acid would have to be carefully granulated, though, to prevent it from being a serious inhalation hazard, and if it overheated (past 42.35°, assuming no freezing point depression from impurities) it would glom together into just a sticky mass. And I don't have a prilling tower.

02021-08-09

Attempted to repeat the production of the apparent aluminum phosphate without success. I suspect the sand may have been wet enough to keep the temperature too low, but stoichiometry is another possible culprit. This time I had four areas heated to orange heat at different times: nothing (just construction sand), possible aluminum hydroxide, possible aluminum hydroxide with diammonium phosphate, and just diammonium phosphate. The grey aluminum compound remained inert to the heat in both cases; the phosphate bubbled orange as before, but only produced a cemented mass in the place where it was alone on the sand, not where it was mixed with the aluminum compound. This time, though, the cemented mass was acidic and black, and produced an acid gas while being flambéd. The aluminum compound looked for all the world as if no phosphate had ever been there. The phosphate flames were tinted green, a phenomenon I could barely see and hadn't noticed before.

The waterglass+borax+starch pieces I'd made previously have become brittle and fragile as they dried out; I can break a massive (12mm) bar of it with my thumb, and upon dropping another piece on the floor it broke. Also, one piece that had initially hardened on a piece of PET is off-white on the air side and pure white on the plastic side, suggesting that the water accumulation I'd observed on that side led to a permanent compositional difference, perhaps the dissolution of all the borax. However, they haven't developed visible cracks, and they still form an intumescent refractory foam upon flaming. It smolders for a few seconds upon flame removal, though, and upon continued flaming the foam melts somewhat; this suggests both that the composition doesn't contain enough waterglass to suppress

smoldering entirely, and that it doesn't contain enough carbon to form a continuous carbon network that can remain solid even when the silicate network melts.

This probably suggests that I should consider using way less borax in these mixes. WP says non-alkaline-earth borosilicate glass is 12–13% boria, alkaline-earth types are 8–12% boria and 5% alkaline earths and alumina, and high-borate types are 15–25% boria with lower amounts of alkaline earths and alumina, the remainder being silica in all cases. So even a small amount of borax is probably adequate.

02021-08-10

I attempted to make three crude pieces of pumice flat by rubbing them together by hand for about 20 minutes, without external abrasives. The resulting deviations from flatness over distances of some 30 mm were on the order of 1 mm. In one of the three cases this was large enough to easily detect by touch, which seems to be due to an embedded hard stone; the other two look and feel flat (except for bubbles), even in glancing light, until you test them against a really flat surface. A great deal of white pumice flour covered the floor, its adherent nature posing difficulties for broom cleanup; also, it is slightly gritty between the teeth. Some of it has gotten into my keyboard and mouse buttons.

I filled a plastic shopping bag with methane, but to my disappointment it did not rise into the air, though it did burn nicely, leaving an unobjectionable candle-wax aroma. On calculation it seems to be about a sixteenth of the required volume. See Methane bag (p. 710).

02021-08-12

The borax silicone from four days ago is rubbery all the way through. Washing it in water reveals that it is spongy and hydrophobic; drops of water can be squeezed out of its void spaces and bead up on the surface; evidently the water is washing the borax out of internal spaces, so it's like a silicone sponge. Upon washing it in 95% ethanol instead, gritty borax comes out on my hands; a drop of water immediately dissolves it away into slippery wetness, confirming that it's borax, or at least something alkaline. Cutting it in half with a razor knife makes some white noise as the knife encounters hard grains and leaves crystalline grains of borax on the razor knife and the exposed surfaces; they dissolve immediately in water. So on one hand there seems to be no uncured silicone, as desired, but on the other hand the borate mostly doesn't seem to have been available to interact with the silicone. I suspect that probably I used enough crystalline borax to make the silicone effectively an open-cell foam, with the pores full of borax, perhaps allowing moisture from the air to penetrate. I can't smell any acetic acid smell, but it *has* been four days. Perhaps a better-controlled version of the experiment would put the silicone and borax inside an airtight membrane, such as a party balloon or ziploc bag; this would prevent acetic acid from escaping or moisture from entering.

I mixed some Oogoo in the same way with cassava starch and the

same silicone, inside a pair of polyethylene shopping bags. It never did reliably stop being sticky; when there was enough starch on the surface it would be nonsticky, but sufficient kneading would make it sticky again. I have left it to possibly harden. I was hoping I could mix it by kneading the inner shopping bag from outside, but instead I got silicone all over my fingers, which I washed off with dish detergent and water. No skin irritation was evident four days ago, and none is evident now.

After sitting for a while, its surface stopped being sticky at all; further kneading made it a little sticky again, but I didn't continue kneading long enough to see if it returned to its original stickiness. I just rolled it back into a ball.

A few hours later (4?) the Oogoo thing had thoroughly solidified into a non-sticky smooth white rubber ball, which bounced nicely. Slicing it with the razor knife revealed that it was solidified all the way through (30–35 mm), so the cassava starch did succeed in avoiding the standard problem with household one-component silicone where it only cures in a very thin layer. It still has a noticeable smell of acetic acid.

Even though it's cured all the way through, it's still noticeably viscoelastic.

It easily sustains a flame once ignited. Burning it thoroughly with a butane torch converted it to a fragile carbon foam slightly larger than the original rubber, with the surface converted to a white ash, and produced a slight annoying smell of acid. The vermiculite near where it was burned received a gray deposit from the fumes. I wet three of the vermiculite stones in a (not pure polypropylene) bottle cap with a drop of water; baking soda then added did not fizz. So whatever the gray stuff is, it's not noticeably acidic in water.

02021-08-13

In the end my lungs hurt for about an hour last night from, I guess, the smoke from the burning silicone.

The 3M 420 silicone I've been using costs AR$310 for 280 g, which at AR$180/US$ is US$1.72, or US$6.15/kg. The bisected ball from the other day weighs 19.0 g, probably half of which is cassava starch (AR$190/kg or US$1.05/kg), so this ball cost about AR$3 of material.

We have a glass microwave turntable we've been using as a plate, but it's not very stable (it tends to tilt one way or the other, or spin around the middle, because it has three little glass projections near the center) so I've cut part of the ball into three little feet and glued them onto the bottom with more of the silicone. We'll see if that works.

The Monster can of muriate of lime I dissolved/melted on 02021-07-30 has finally accumulated an appreciable amount of liquid on its surface by dehydrating the air.

At about 21:15, in one plastic shopping bag I put about 12.0 g cassava starch and 23.2 g silicone and kneaded them together through the bag, which produced a mass too sticky to remove from the bag; in another I similarly mixed 11.8 g silicone with 21.5 g starch, but despite extensive kneading I cannot get it to cohere into a single mass.

It just acts like slightly damp cornstarch, easily crumbling away. I can't smell the acetic acid anymore, but I have the impression that the lower-silicone crumbly version had a lot less acetic aroma to it at the time.

So I think the silicone-to-starch ratio has to be pretty close to 1:1; neither 2:1 nor 1:2 works well. I might try a third batch where I start with a ball of silicone in a vast oversupply of starch, then progressively knead in more starch, then measure how much starch remains unincorporated once I've produced a reasonable product. That seems like it might work better than trying to knead it through the bag. Like how you flour your hands when kneading bread.

After an hour and a half, though, at 22:47, I can peel the 12g-starch, 23.2-g silicone mix off the bag and roll it into a ball, which weighs 30.2 g out of the 35.2 g originally placed. So there's probably about 5.0 g of silicone (and starch) still stuck to the bag that I didn't manage to peel off. The ball is still annoyingly sticky to the touch. I can smell the acetic acid again, and it's sort of chokingly strong.

After another half hour, at 23:13, the ball is no longer fully plastic; it tends to return to its original form after even fairly extreme deformations, so it is no longer kneadable. If this is typical behavior it's fairly annoying: it went from annoyingly sticky on the hands half an hour ago to unworkable now. I wonder if a larger amount of starch would mitigate this behavior, because the ball last night was a lot more kneadable when reasonably unsticky. But maybe I just got lucky?

My fingers sure do feel smooth.

I mixed some more sodium silicate (waterglass) with cassava starch on another shopping bag to form a dough, which I tried to use to form a little pocket oven. This was unsuccessful, as over the following 20 minutes the oven gradually slumped down into a flat object; evidently sodium silicate with cassava starch behaves as a very viscous fluid, not a viscoelastic solid. Maybe gelatinizing the starch first would help.

I flambéd a small piece of it with the butane torch. It produced a little bit of smoke, which smelled like pine pitch — Mina reported that it smelled like mentholated wood. Upon long flaming it was able to remain smoldering for a few seconds after the flame source was removed, but not sustain a flame on its own. The carbonizing stuff swelled up like a sopaipilla in the torch flame. I was mostly able to hold it just in my fingers even though it was only a few cm long; at some points I resorted to needlenose pliers, which broke through the surface and exposed the hollow interior.

Mina reports that the sodium silicate bottle has a very strong smell resembling that of toilet cleaner (the kind based on muriatic acid), one which provoked nausea. I can't smell anything from it, but I suppose there must be a lot of hydroxyls flying around if she can.

In between the hydroxyls from the sodium silicate, the acetic acid from the Oogoo, and the pine-scented smoke from the intumescent, we're both kind of clearing our throats a lot.

By 00:00 the Oogoo seemed to be fully hardened but of course still

preserves a quite strong acetic-acid aroma. I stuck a hardened steel shaft through it.

To estimate the water content of the liquid that had formed on the muriate salt, I took a shiny steel bowl (nickel-plated?) weighing 42.3 g and poured the salt solution out of the Monster can, bringing the weight to 47.3 g, so about 5.0 g of liquid had formed. Then I heated the bowl on the stove until it was apparently just dry white deposits of muriate of lime, though a little hissing was still audible. At this point the bowl weighed 45.0 g, so only about 2.7 g of muriate of lime remained; 2.3 g of water had boiled away. (Also, a little bit had been thrown out of the bowl by the violent sizzling bubbling — white specks were visible all around the rim of the bowl, so some must have also overshot.) After strongly heating the bowl on the stove a few minutes more, bringing parts of its floor to red heat while covering the top of it partly with aluminum foil, turning parts of the shiny nickel (?) black, producing a strong smell of metal, it still weighed 45.0 g, so if more water was lost, it was compensated by oxidation of the bowl. After washing the bowl with water, during which a bunch of the black stuff came off onto my fingers ($NiCl_2$? But that would be green and water-soluble. $CrCl_3$? Would be purple (but maybe I wouldn't notice the difference)) and drying it, the bowl weighed 42.2 g, so evidently there wasn't a lot of oxidation going on, so probably the white stuff from when it boiled dry was pretty much just the anhydrous salt, as you'd expect.

That would mean 2.3 g of water was enough to hydrate and dissolve 2.7 g of anhydrous muriate of lime, so something is clearly wrong here! According to Heating a shower tank with portable TCES? (p. 714) the solid hexahydrate should weigh 219.07/110.98 of the anhydrate form, or 5.3 g/2.7 g, so 2.6 g of water should have been required just to get to the hexahydrate. Moreover Wikipedia says the hexahydrate only dissolves 81.1 g per 100 mℓ of water at 25°, so to dissolve those 5.3 g we should have needed 11.9 g of total solution, another 6.6 g of water, or more at temperatures below 25°. It's definitely cold enough here for the hexahydrate to solidify.

I'm not sure where my error is exactly. The Wikipedia numbers predict that 11.9 g of saturated hexahydrate solution should have been required to boil down to 2.7 g, not 5.0 g. Given how exactly the scale was able to reproduce previous measurements, it hardly seems likely that it had a 6.9 g error in measuring the weight of the solution I dripped into it, even if maybe it does have a little bit of silicone on the table under it. It measures the blob of Oogoo at 30.8 g, the one it measured as 30.2 g a few hours ago, but also I think a little more Oogoo got stuck to the ball after I initially weighed it. Maybe this stuff isn't really muriate of lime — but then, what else *could* it be?

I should probably see whether, say, 20 g of salt from the bag loses mass if I heat it to red heat, and then how much water it takes to start to dissolve it at room temperature, which should tell me how much mass the hexahydrate has.

Interestingly, when I pulled the shaft out of the Oogoo to weigh it, it left a perfectly round hole through the middle of the blob. I guess it's still flowing a little bit (or creeping a lot, depending on how you look at it). It's 01:22, and I've stuck the original shaft back in, and

I'm sticking a second shaft through it to see if the hole *it* made remains open, or round, when I pull it out tomorrow.

02021-08-14

It took a couple of hours for the respiratory irritation to die down.

The Oogoo is noticeably harder this afternoon. The second shaft left a hole smaller than the first, about half the diameter (maybe 1mm instead of 2mm; the calipers say the shafts are 2.94–2.98 mm, so I think that even the first hole is smaller than the shaft that made it); I've taken both shafts out and inserted one of them to make a third hole. It still smells of acetic acid.

The waterglass starch dough has mostly hardened in the position I left it in. It occurred to me that last night it was flowing liquidly when left alone for long periods of time, while breaking brittlely when I tried to form it rapidly, which is pretty much the shear-thickening behavior you'd expect for a suspension of starch in water, just at lower flow rates.

We found a stone mortar that had been missing. It was in the kitchen cabinet. Unfortunately, vinegar testing on the bottom shows that it's marble, not agate, so it will contaminate anything ground in it with marble dust. Nevertheless I will grind some of the borax. Agate mortars are available for around US$800, which seems excessive when a chunk of agate of the right size is only US$30.

I used the mortar to grind some borax to a mostly flour-like consistency. I weighed it into the same bowl I used yesterday, which now weighs 42.3 g; after taring the bowl, the borax weighs 52.5 g. I took a Nutella jar which weighs 30.2 g; upon adding the borax, it weighs 82.6 g, giving a 52.4 g weight of finely ground borax. I'd probably need to screen the borax to separate out the remaining large crystals I missed in the grinding.

The borax tastes like baking soda (something I haven't been brave enough to try previously) but does not bubble in vinegar. After adding vinegar to it, baking soda that is added still fizzes, but very slowly, which is consistent with the borax having converted the vinegar into sodium acetate and much weaker boric acid.

To try Oogoo with a little of this new finer borax, I mixed 10.2 g of cassava starch with 0.5 g of this flour-ground borax in the bowl, then transferred it to a shopping bag. In another bag, I mixed 10.3 g of cassava starch with 2.0 g of the same borax. To the shopping bag with the 0.5 g of borax I added 10.4 g of the same silicone; to the other bag with the 2.0 g of borax I added 13.2 g. In both cases I smooshed them around by kneading them through the bag until they seemed pretty uniform. Now it's 01:48 by the laptop clock.

The idea is that the borax may do one or more of: provide borate ions that cross-link the silicone to make it harder; neutralize the acetic acid into sodium acetate, converting the borax itself into relatively inert boric acid, thus reducing the annoying outgassing; or release extra water, speeding the acetic cure. And by being more finely ground, maybe it will work better for these things than last time, and also not add so much porosity to the final material as my first borax silicone test on 02021-08-08.

At only 01:56 (8 minutes later) the two Oogoo batches are noticeably harder. I can already peel them off the bags. The ball from the other bag (with 2.0 g of borax) weighs 25.8 g, rather than the 2+13.2+10.3=25.5 g expected, suggesting a weighing error. The ball from the shopping bag weighs 20.9 g rather than the 0.5+10.4+10.2=21.1 g expected. I think I left more of the silicone stuck to the very thin shopping bag because it was harder to peel the bag away from the silicone, perhaps because it was much more wrinkly. But I guess it probably wasn't more than about 3%.

It's 02:09 now, 21 minutes after initial mixing, and the two chunks of Oogoo are very moldable and kneadable and still kind of annoyingly sticky, but at least they're not leaving chunks of silicone stuck all over my fingers any more. They both have a strong acetic acid smell. So far, there's no evidence that the borax has had any effect. That does sort of mean that it was ground finely enough to eliminate the obvious grittiness I had in the first borax silicone test. I took advantage of the kneadability to fold each one in half 20 times to mix them more homogeneously.

At 02:20, 32 minutes after initial mixing, they're still very moldable and kneadable, and noticeably less sticky. The two versions seem pretty identical though. Most of the acetic-acid smell seems to be gone from the shopping-bag ball, the one with only 0.5 g of borax. Both balls look wet when being left without kneading for a few minutes, suggesting that the silicone is smoothing out rough surfaces with viscous flow. Kneading makes them look dry for a little while.

I think I'll try kneading some construction sand into part of one of the balls. I dried some sand in the bowl, which weighed 70.7 g with the dry sand, and thus this is about 28.4 g of sand. I'm pulling off about half the other-bag ball, 11.9 of 25.8 g of Oogoo. After having kneaded in sand to 21.4 g (presumably 9.5 g of sand) it was starting to have noticeably less structural integrity, but I managed to incorporate essentially all the sand and still have a brown plastic mass weighing 39.0 g. I must have lost about 1.3 g of sand along the way. Mina commented that it felt like whole-wheat flour.

As plastic molding compounds go, this sand mix is not quite as cheap as pottery clay, but it's probably watertight once it hardens at room temperature, and it wouldn't be surprising if it were enormously more impact-resistant as well, though weaker and softer. The proportions are, I think:

10.3 g cassava starch × 11.9/25.8 = 4.8 g (12%)
2.0 g borax × 11.9/25.8 = 0.92 g (2.4%)
13.2 g silicone × 11.9/25.8 = 6.1 g (16%)
27.1 g sand (70%)
39.0 g total

1 kg of the stuff would cost 2.1¢ for the sand (US$0.03/kg according to Potential local sources and prices of refractory materials (p. 566)), 98¢ for the silicone, 7¢ for the borax at US$3/kg, and 12¢ for the cassava starch, for a total of US$1.19/kg.

It's now 02:52 by my laptop clock. It's still possible to fold the Oogoo over on itself an arbitrary number of times, but there's noticeably more resistance, and quite a bit of elasticity, where it tends

to return partway to its previous form. So at 64 minutes past initial mixing it's already past its prime for molding.

02021-08-15

My hands and upper respiratory system were a little irritated last night; some of my fingers itched. I suspect exposure to acetic acid exacerbated by a lot of handwashing was to blame.

It's 19:17.

All three Oogoo samples from last night are fairly hard, as rubbers go. The sand-filled silicone is noticeably harder than the other two, which otherwise seem pretty similar. An acetic-acid smell is evident from all three.

Attempting to cut the sand-filled silicone with a razor knife quickly destroy the knife, within a few millimeters. I think that if I were to push it hard enough I could probably get it to keep cutting, because I can cut the stuff with my thumbnail. However, it's rubbery enough that when I let go of the cut piece, it flaps back into place. Note that this is very different from the behavior I predicted for sand-filled silicone on 02021-08-08, when the borax didn't stick to the silicone.

The sand-filled silicone definitely wets with water better than the other silicones. I haven't measured a contact angle or anything, though. When heated up to 100° in a boiling-water bath, it wets completely with water.

At 19:55, I've been boiling the sand-filled silicone for about half an hour. It seems slightly changed: the surface feels slimy and leaves a slimy substance on the fingers, and it no longer smells of vinegar. But it feels about as hard as before.

I think the seal made between the 2.94-mm ground hardened steel shaft and the 2-mm-or-so hole it made in the 02021-08-13 Oogoo (2 hours and 45 minutes after being mixed) amounts to a usable sliding seal. It's easy to spin the shaft with the fingers when it's pressed through the silicone, and I'm reasonably sure that the hydrophobic nature of the silicone would keep water from sneaking past it. You'd want to use a thinner silicone washer in practice than the 30+ mm. A casual test with a drop of water on the slice of that block of Oogoo suggests that this would work, but of course that doesn't tell us anything about how much pressure the seal could withstand.

Mina discovered that the Oogoo from 02021-08-13 is capable of erasing pencil marks from paper, much to my surprise. In fact, it seems to work better than a normal pencil eraser, although the graphite remains on the surface of the silicone instead of being carried away in crumbs. But getting the stuff to flake off in crumbs is presumably just a matter of adding enough of some kind of friable filler or filler with poor adhesion to the silicone, like the borax-filled silicone on 02021-08-08 that ended up making a fragile sponge.

At 20:22 the sand-filled silicone has been out of the boiling water for nearly half an hour but still feels wet. I suspect I gelatinized the remaining starch in its surface layer, and that's now retaining water, and smears off on your fingers when you rub it on them. Rubbed on a wine bottle, it leaves a visible smear which does not wipe off with a

dry paper towel, but does wipe off with a wet one, suggesting that it is indeed some kind of water-soluble thing like gelatinized starch. It still feels the same way another hour later, at 21:31. I wonder if it will grow mold (you'd think the borax would prevent that, but the borax might have been leached out of the surface layer by the boiling).

This slimy water-soluble stuff makes me wonder if maybe with so much starch the stuff might not be waterproof even at room temperature. It might be worth trying a wider range of ratios and fillers.

A permanent marker (Du Hu brand, UPC 9-930691-011013) easily marks the Oogoo (tried on the two sand-free samples from last night and the one from the other day), but it easily rubs off. A smiley face thus drawn on one of the Oogoo balls rubbed onto paper transfers onto the paper, suggesting the possibility of flexographic offset printing with this material.

By 02:41 the sand-filled Oogoo had stopped feeling wet on the top side (and stopped depositing water-soluble slime on wine bottles when rubbed) but still felt wet on the bottom (and still deposited slime).

02021-08-16

The 39 grams of sand-filled Oogoo had dried out by this afternoon. I pounded the crap out of it with a hammer against some bricks (maybe 20 hammer blows of 50 J each); it crumbled somewhat and became softer and porous, but mostly retained its structural integrity. The only pieces that came off were due to a couple of hammer blows where the corner of the hammer punched all the way through it into the brick on the other side. The surfaces are now whiter and rougher. It now weighs 38.0 g, and I regret not having weighed it before pounding it with the hammer, because I suspect that most of that weight loss was from boiling starch out of its surface, not from hammering damage.

This seems like noticeably worse impact toughness than I would normally expect from this kind of silicone, but then again, this object is only 16% silicone by weight. It's kind of in the ballpark for what I'd expect from non-ferrous metals, although of course this impact toughness was achieved mostly through large deformations rather than large stiffness.

As I suspected yesterday, I can cut through the stuff with a razor knife just by pushing it hard enough through the material if I carefully avoid sawing back and forth, which immediately destroys the edge of the razor knife. By contrast, the knife edge is mostly intact when I just push through, though not without nicks from the sand. Pushing hard even works when the knife edge has been thus dulled by sawing, although it requires more force.

Using the unsharpened *back* of the razor knife, I can cut through the pumice quite readily — not quite as readily as cutting soft materials like cardboard or silicone, but very nearly. This further reinforces my impression that Foams Are A Miracle.

02021-08-19

The Oogoo samples from 02021-08-15 are of course fully hardened through, and can easily be cut with a razor knife. Perhaps more interestingly, with some sawing, they can be cut with a tin can lid as well, which takes more effort and leaves a scalloped cut. An acetic-acid smell is evident upon cutting, but it is not chokingly strong.

I got a new, hopefully less melty blowtorch head, made of brass (US$14), for those 8-ounce (227-g) butane bottles.

I also got some 5-mm carbon arc cutting electrodes (11.2 g, 5.11–5.19 mm diameter at the short unclad tip, 5.20–5.32 mm diameter in the copper-clad section, 306 mm length, thus average density around 1.69 g/cc). Treated as pencil leads, they leave only very light marks on paper (about 4H or 6H pencil lead hardness, say.). With significant difficulty I can break the end off the carbon rod with my fingers, revealing an earthy fracture, dull black with a few tiny bright particles, probably cleaved graphite grains. The sparkly carbon cladding resists water, ethanol, and nail polish diluent (probably ethyl acetate) with no damage; when heated, it forms iridescent coatings and then dull black copper oxide; and my multimeter can measure no resistance (though it bottoms out at about ½Ω), so it may be just copper plating rather than any kind of paint. The ohmmeter measures the graphite itself as highly conductive, but with significant contact resistance of a few hundred milliohms. Measuring the resistivity of the carbon rod itself would require removing the copper from it.

I heated one of the electrodes to orange heat with the brass torch. Mina used the hot end to burn holes in a cardboard box, which curiously smoldered and went out rather than igniting.

I flamed one of the waterglass/starch/borax samples from before, and it intumesced lightly as before.

I took one of the old waterglass-foam samples and melted it a bit with the new brass torch, which was relatively easy. Then I poured a bit of oil of lime (it keeps accumulating in the Monster can) onto the sample and tried again; the oil of lime seems to have been effective at increasing the heat resistance of the foam by, presumably, converting it to amorphous calcium silicate; now it apparently suffers no degradation even at white heat. Since muriate of lime melts at 772–5° the white-hot solid residuum cannot be merely muriate of lime.

The process filled the kitchen with a cockroach-like aroma, which, as Mina pointed out, is almost surely smoke from the bamboo chopsticks I was using to hold the waterglass foam sample in front of the flame.

I tried washing the sample with water after heating to remove the salt, which seems to have caused it to crumble without removing all the salt, because it still left a sticky residue on my hands after washing (more like oil of lime than plain salt) and created a white efflorescence on subsequent heating. This is probably a problem that could be solved by sufficient heating time and washing time.

This suggests the following process for making a tiny waterglass foam oven:

- Pour the appropriate amount of waterglass onto a flat sheet of

polyethylene and let it dry to a thin layer of glass.
• Build the overall structure out of some moderately heat-resistant material: beer cans, aluminum foil, calcined alabaster paste, mud, that kind of thing.
• Paint the thin layer of glass with an even thinner layer of waterglass to make it sticky, then break it into small pieces.
• Stick the small pieces of waterglass onto the inside of the oven structure, several layers thick, to form the intumescent refractory layer.
• Blow enough flame or hot air through the inside to foam up the waterglass.
• Pour or spray oil of lime onto the inside to provide lime to the foam.

• Blow enough flame or hot air through the inside to thoroughly dry out and possibly melt the muriate of lime, and possibly drive off some marine acid air.
• Possibly, after it cools, wash the foam with water to remove the potentially corrosive salt.
• Add a hardfacing layer on the inside of the foam, maybe paper coated with waterglass and silica sand or aluminum hydroxide, or waterglass mixed with starch and charred, to form a solid layer that can support solid objects without damaging the surface of the foam.

You could also paint the waterglass directly onto the structure, but you might probably need to paint several coats, waiting for each one to dry in between, to get enough thickness. It would be pretty cool if you could include the oil of lime between the layers of waterglass instead of applying it afterwards, so that's probably worth a try to see if I can get reasonably low densities that way.

It's likely that such a low-density foam will have a more open-cell structure and thus more convection through it than the denser closed-cell foams that firebrick normally consists of, and therefore a lower thermal insulance. Even then, though, it might have better insulance *per mass*, so it might still be useful for portable pocket ovens.

The pieces of waterglass/starch dough from 02021-08-13 have thoroughly hardened. I flamed one of them with the brass blowtorch for a few minutes, and it intumesced, swelling by perhaps a factor of 2, forming a black foam which mostly did not melt, maybe just a little bit around the edges. Both the uncharred material and the charred material take noticeable effort to break by hand; the charred material is a fine foam with mostly bubbles around the size of 100 μm which can be indented with the thumbnail with some effort, while the uncharred material does not appear to be a foam and cannot be thus indented. It did smolder for a few seconds after the flame was removed, producing the pine-pitch-like smell I mentioned previously.

I had previously mixed up calcined alabaster powder with a retail baking powder (double-acting I think, and including bicarbonate, but I don't know what else) and baked it in the oven into a white biscuit. It had formed an open-cell foam with about 50% porosity and pore sizes on the order of 2 mm. I took half of this and heated part of it with the brass torch to white heat for 7 minutes. This formed a black circle, with a dark gray circle inside of it, with a 20-mm-radius white

circle inside of that, with increasing cracking visible in the gray and white parts. Upon removal of heat, it continued to glow visibly through the cracks for a minute or so. The white part flaked away somewhat under finger pressure; on breaking the biscuit, it seemed to have penetrated some 5mm deep. Unexpectedly, most of the biscuit thickness had turned gray, and it fell apart in my hand.

On repeating the process, I noted a slight acid smell, and my lungs became irritated, perhaps due to failing to maintain a judicious distance from the proceedings and producing a bit of vitriol from the alabaster.

I used the torch on the Monster can I'd previously (02021-07-30) lined with borax/flour/starch/water intumescent refractory to heat up some eggshells to a yellow or white heat for a couple of minutes. Unfortunately, the refractory material began to smolder, and the paint on the outside of the can started to change color, indicating that it was burning, so I aborted. There was also a distinct smell of ammonia, so perhaps the can was sitting on top of some undecomposed diammonium phosphate from a previous test. (Also, a smell of burning plastic from the can.) The lining had broken apart into many pieces, exposing much of the walls, and it continued to smolder for several minutes, even after I dumped it out of the can, so, again, the borax was not as effective at preventing combustion as one might hope.

I picked the eggshells out of the smoldering rubble with chopsticks (some of them had stuck slightly to the carbon foam) and dropped them into a plastic yogurt cup, then dripped a few drops of water into it. There was no visible reaction or evident heating, but after a few minutes the water looked milky. This seemed promising. The milkiness was due to tiny white particles. However, several drops of this milky liquid were unable to neutralize a couple of drops of vinegar to the point where baking soda wouldn't fizz, so the milkiness probably wasn't slaked lime as I was hoping it was.

Properly calcining lime probably requires longer heating, maybe 15 minutes to 15 hours. 15 days is more traditional, but I think I can do better than that.

I had the remaining fragments of waterglass foam floating in water for a few hours. They did not crumble any further, so whatever caused them to crumble earlier, it probably wasn't the water — either it was my fingers' roughness, or the process of boiling water *out* of them, or maybe there were parts subject to water attack and parts that weren't.

I placed them on a bed of vermiculite and heated them with the brass torch to white heat. Once white heat was reached, they slowly melted, but they retained substantial foam structure for several minutes of such heating. (As before, untreated waterglass foam disintegrates upon being heated to merely orange heat in well under a minute, collapsing down to a glob of glass.) I don't know if this is because the calcium treatment was incomplete or because lime silica glass can't withstand the temperatures involved. The vermiculite remained unaffected, so the temperature wasn't exceeding its melting point.

Some acid gas was evolved from the vermiculite, so I think it may

be contaminated from a previous test.

02021-08-20

Mixed up some new Oogoo, this time with coloring. In a 126.5-gram glass jar Mina mixed 11.9 grams of dish detergent (see below) with a tiny amount of food coloring paste (see below). This produced a brilliant dark violet color. On weighing it afterwards, the scale read 137.9 g, which is 0.5 g less than before adding the coloring (so there's only 11.4 g of detergent/coloring mix); I think there are about 0.5 grams of detergent mix left on the Q-tips.

For the Oogoo, I mixed 27.6 g of the same 3M 420 silicone as before with 27.6 g of cassava starch inside a tiny plastic shopping bag by kneading it through the bag. I think a Ziploc bag would be a much better approach; the thin (10-μm?) bag developed a hole in it during the kneading process. After it was thoroughly kneaded, we waited 9 minutes, then kneaded the resulting deposits into a 43.5-gram ball of Oogoo. To this we added 2.9 grams of the detergent/coloring mix, which I kneaded thoroughly together, getting a uniform light lilac color.

The dish-detergent-added Oogoo (≈6% commercial detergent mix, maybe ≈1% sodium lauryl sulfate and sodium laureth sulfate, but we don't really know) was noticeably softer during molding and seemed noticeably less prone to stick to our hands, but also seemed to be less cohesive during kneading. (I now realize I should have kept some aside without detergent to compare post-cure properties.) I'm astonished that it was able to mix with the silicone at all; Mina told me it would work but I didn't really believe her. I thought the water either wouldn't mix in or would immediately polymerize all the silicone. (Surely the silicone polymerization does consume some of the water from the detergent.)

Unfortunately I neglected to weigh the ball after mixing the detergent into it, which would have been a useful clue as to how much of the detergent and coloring went into the Oogoo and how much soaked into my hands instead.

Mina shaped most of the ball into an eight-petaled flower, then painted part of the outside with food coloring/detergent mix left over on the Q-tips. I shaped a small mushroom from it.

I also neglected to measure how long it took to harden the silicone, but it did seem to be quicker than on other occasions.

Dish detergent: "5× concentrated", lemon scent, Magistral UPC 7-500435-137900; biodegradable active surfactants *alquilsulfato de sodio, alquiletoxisulfato de sodio y óxido de amina*; other ingredients: *etanol, agua, coadyuvantes, agente de limpieza, conservantes, colorante y fragancia*, which is roughly as nonspecific as you could possibly get. " *Materia activa minima 20%*" might mean it's 20% surfactants, or it might not.

Food coloring paste: Fleibor.com.ar; *Azul T: azul brillante al 3.17%* , bottled April 25, 02016, ingredients: *azul indigotina, azul patente, tartrazina, propilen glicol, glicerina, azúcar, dióxido de silicio*, which is a pretty intense cyan, and also *Violeta L: azul indigotina y amaranto al 14.51%*, bottled January 4, 02016, ingredients: *entrocina, amaranto, azul*

indigotina, azul patente, propilen glicol, glicerina, azúcar, dióxido de silicio, which is a pretty intense magenta.

It is now 16:58.

At 17:20, the color/detergent mix painted onto the outside of the flower is still wet, but the Oogoo is quite firmly set.

At 17:26 I lined a dry steel bowl with aluminum foil (the 10μm stuff I might have mentioned previously, maybe Ecobol shitty store brand; folded in half 5 times to make 32 layers, I measure 0.32 mm with the calipers) and poured waterglass into it, then covered the top with aluminum foil and put it on low heat, the intent being to make some more waterglass foam, but this time with an aluminum foil backing on one side. At 17:31 and 17:37 and 17:45 it's crackling in a threatening fashion. At 17:51 it has died down quite a bit. At 18:05 it seems to have stopped completely, and the top aluminum foil is distended upwards; I can feel a hot, hard round shape through it, which is presumably waterglass foam, suggesting that this approach to foaming waterglass has actually been enormously more successful than I had imagined it could be. I might have had 2 mm of waterglass in the bottom of the bowl and now it's apparently like 70 mm tall! That suggests a considerably larger void fraction than I've been able to achieve in the past.

I turned it off at 18:56 after cooking on low heat for an hour and a half, at which point it was maybe making some tiny little crackles you could barely hear if you put your ear next to it. I'm eager to see what the foam structure looks like, but I want to let it cool slowly to reduce cracking from thermal deltas (I can hear slight crackling, which is probably precisely that). The whole bowl, foil, foam, and all, weighs 57.6 g, but I'm not sure if this is the same bowl I previously weighed empty at 42.3 g or a different one. It's about 170 mm in diameter. If the loaf's volume is ⅓ of the cylinder it fits inside, ø170 mm × 70 mm, that would be 530 mℓ. If it really weighs 15.3 g and occupies 530 mℓ, that's 29 mg/cc, *half* the density of the foams I made previously. I wonder if it's just one giant bubble, though.

At 19:24 it seems to be silent and the hard shape distending the top of the foil is cool to the touch.

At 19:27 there is only the occasional click from it. I removed it from the bowl. It weighed 15.5 g. I had hoped it was filling the whole interior of the bowl, but that's not what happened at all; it's probably only about the same density as the previous samples. Because the heat was being applied from below, the bottom part of the waterglass, stuck to the lower foil, hardened first. Then, as the rest of it gradually hardened, the upper part expanded, causing the waterglass layer to curl and form a large empty space below it, some 30 mm deep and 70 mm across. The overall mass of waterglass froth, which looks very much like soap-bubble froth from well-shampooed hair, is some 90 mm across.

With some care I was able to peel almost all of the upper aluminum off of it, tearing off aluminum tags in only three places (removed successfully with the fingers), and leaving invisible fragments of waterglass on the aluminum in a few more places that can be felt with the fingers. The outer soap-bubble-like film is very fragile, and on a couple of cases as I weighed it and opened the

package, I have seen bubbles of glass floating around in the air. A small puff of air from my lips can shatter some of the foam into glistening fragments.

https://ibb.co/4WrfjQs

On the tongue the delicate bubble surfaces feel a bit like gelatin bubbles, but have no taste, leaving a slight slipperiness and tingling for a few minutes like some soaps. (At this point it's worth mentioning that this is not a very alkaline waterglass formulation; it has a lot of silicon per sodium.) When touching the denser parts of the froth with my tongue, they do not immediately dissolve, but remain solid, as you would expect for sodium silicates. However, nothing gritty comes off in the mouth.

The aluminum foil I peeled off the top weighs 1.7 g, and the remaining waterglass foam and bottom foil together weigh 13.8 g. Since the top and bottom foil are about the same size this suggests that the foam alone weighs about 12.1 g.

Concerned that the tiny glass flakes from the bubbles might cause skin irritation, I rubbed the froth lightly on my slighly sweat-damp left thigh to deposit a bunch of shiny bubble tops, then rubbed it around thoroughly until the deposit was just a fine glitter, so that if the microscopic sharp glass edges causes irritation mechanically (like fiberglass) I will be able to observe it there. Either I will get a hell of a rash on my thigh or I can tell people I'm Edward Cullen.

Kinesthetically the foam feels like it's about 25 or 30 mm thick in the middle, maybe 15 mm thick near the edge. This permits a crude volume estimate as a 120-mm-diameter circle (due to the edge curling) that's 20 mm thick, which would be 230 mℓ. 13.8 g in 230 mℓ would be 61 mg/cc, which is about the same density I've gotten in the past. This might be slightly less dense because of the bottom foil.

(Later measurements, after cutting, show that it was about 20 mm thick in the middle and about 10 mm thick near the edge, occasionally as much as 15, but 120 mm is about right.)

I heated one side of the waterglass mass with the brass torch and, as before, the surface immediately melted at orange heat, accompanied by a great deal of crackling and a few flakes flying away in the air, then settling nearby. After 20 seconds of heating with the torch, it looked like styrofoam whose surface has been eaten by gasoline or acetone. I poured some saturated oil of lime over another part of it, waited about 20 seconds, heated it gently with the torch at a distance for about 30 seconds to drive off the water, and then heated the lime-treated surface strongly for about 20 seconds. The treated surface melted only very slightly at yellow-white heat. This left a white efflorescence. Touching it with my fingers leaves the sticky oil-of-lime feeling on them, so I suspect that only some of the oil of lime was consumed.

I tried to film this with my cellphone but can't figure out how to get the exposure right in video, so the blowtorch-affected area is just white. The video has some other problems, like being 310 MB. I was able to take some stills, especially after I learned about the voice-activation feature of the cellphone.

https://ibb.co/PgCdb4p

There was some of the “cockroach” smell from heating the lime-treated part of the waterglass, even though no chopsticks were present, and maybe also some acid gas. This suggests that I might be smelling some kind of muriate or calcium product, or possibly something that’s present as an impurity in my industrial-desiccant source.

After all this it weighs 16.4 g, which suggests that the oil of lime added about 2.6 g, including the part that’s still wet on the still-attached aluminum foil. Removing as much as possible of the aluminum foil reduces the weight to 14.2 g. In the process I noticed that the bottom aluminum foil had actually torn in a few places, looking rather like stretch marks from rapid weight gain. Also, it’s not very firmly adherent; it’s possible to peel it off in bits in places. The waterglass attached to the aluminum foil is noticeably porous, though much less than the surface; I can push my thumbnail through the foil into the waterglass.

At 20:52 my thigh is still glittery and not irritated. It’s been maybe half an hour since I applied the potential irritant, but if I recall correctly from my childhood, skin irritation from fiberglass can take hours to develop. At 21:22 still no irritation.

I passed some water from the faucet over one side of it to see if it would have a similar effect to the fire, but it seems to have only affected a few of the largest and most delicate bubbles.

I cut it in half with a razor knife, mostly using the back of the blade. For this it was convenient to hold the foam between my thumb on the aluminum-foil backing and a finger on the untreated blowtorch-melted area, which is partly covered by a network of dense glass which can withstand finger pressure without crumbling. Cutting through the area that had been treated with oil of lime and then melted was much more difficult. I dropped half of it on the floor from a standing position when I finished; a small piece broke off from the impact.

https://ibb.co/sFVgsXk

With my teeth, I took a tiny bite of one of the areas affected by the water, about 21:34. The material is gritty between my teeth, like toothpaste, rather than dissolving immediately upon being reduced to fine powder. No large sand-like grains are present. It’s still gritty at 21:36. Only a little grittiness is left at 21:44 but probably because I swallowed the grit rather than because it dissolved.

If this foam is 15 mm thick, 120 mm in diameter, and weighed 12.1 g (the guess above), it’s 71 mg/cc, which is a little *denser* than some I’d made in the past (for example by rapid heating). According to Potential local sources and prices of refractory materials (p. 566), liquid waterglass costs US$2/kg, which probably means the solid form costs about US$6/kg (I missed an opportunity to find this out today by not weighing the bowl before cooking the foam). At 71 mg/cc that would be US$430/m³ or 42.6¢/ℓ, which is slightly more expensive per volume as other lightweight insulating mineral aggregates mentioned in that file such as pumice (17¢/ℓ), LECA (29¢/ℓ or 12¢/ℓ for construction), rock wool (31¢/ℓ), perlite (38¢/ℓ), and vermiculite (23¢/ℓ), though these are mostly denser: pumice is

400 mg/cc, LECA is 1200 mg/cc, rock wool is 100 mg/cc, perlite is 128 mg/cc, and vermiculite is 60–160 mg/cc. Plausibly, this waterglass foam could provide better insulation than these other materials, being lighter in weight than most of them, and being a solid mostly closed-cell foam rather than an open-cell foam or a loose particulate aggregate.

This experience suggests a few ways of making sandwich panels out of waterglass foam:

- Sandwiching the liquid between two sheets of aluminum foil and heating it, maybe under some pressure. Preferably you'd use thicker aluminum foil/flashing, like 60 μm, rather than this 10 μm bargain-basement stuff. According to the guess above, the foam's *areal* density is about 110 mg/cm^2. This aluminum foil at 10 μm and 2.70 g/cc is only 2.7 mg/cm^2; one layer on each side would be 5.4 mg/cm^2, which is only 5% of the total weight. According to Sandwich panel optimization (p. 754), sandwich panels of a given areal density are stiffest when their faces are ⅓ of their total mass. (You'd probably need to perforate the foil to allow the water vapor to escape.)
- By heating the surface of the foam, you can melt it into a hard white glass layer, which is very stiff, which is what you want for sandwich panels.

Part of my intent with this sample was originally to carve it into some kind of interesting shape and then melt the shape's surface to make it sturdy (and maybe also cover it in aluminum foil), but I got caught up in other stuff I guess.

I cut out a small cuboid to measure density. It floats in water without getting waterlogged or sinking, which (in combination with its aggressively hydrophilic nature) suggests that most of the structure is closed-cell foam, which is good news for use as insulation. The outermost part does seem to absorb water; perhaps its cells were open when it formed, or perhaps they were originally closed but broke easily. After observing this, I realized I was an idiot, because I hadn't weighed the cube dry, so I rinsed the cube and converted the bowl into a dehydration chamber: I stuck the cube in the bowl on top of a plastic lid on top of some anhydrous muriate of lime. The idea is that if I dry it by just heating it up, I'd possibly damage its foam structure.

It's 02:06 and still no detectable skin irritation.

I tried draping some aluminum foil over the top of one of the cut pieces of the waterglass foam and melting it on with the brass torch, but this didn't work very well; the foam under the melted aluminum foil had a tendency to rip it apart, and aluminum foil after having been melted tends to break easily, and it looks dull and wrinkly. By contrast, the original underside foil remained fairly shiny. I think a better way to put an aluminum-foil surface on a shape carved from this foam might be to coat one side of the foil with *liquid* waterglass, then wrap it around the foam, then cook it.

This process also evolved some acid gas, I think from the vermiculite bed I was resting the foam on for this process.

02021-08-21

The cuboid, probably mostly dehydrated, weighs 900 mg. I really should get a 10-mg-resolution scale! Or I should have cut out a much larger cuboid.

I made Mina coffee and myself mate with the brass torch.

I wonder if I could get waterglass to gel with some agar? Agar is supposed to gel at pH from 2.5–10 and concentrations in the 0.5%-1% range. Then maybe I could make some aluminum-foil waterglass tape with the waterglass gelled with agar, then wrap it to form a thing, then let it dry or treat it with polyvalent cations or with CO_2 before heating. I don't think I have any agar, though.

With great difficulty, I bit through one of the Oogoo samples from 02021-08-15 — the borax one, I suspect, from the taste and the soapy tongue feel afterwards. A slight acetic-acid smell is still evident. After being rinsed, it's relatively straightforward to tear the piece the rest of the way, which is also true with an unrinsed piece that I cut partway through with a razor knife. However, the tearing process is only easy if done very slowly; the fibers of PDMS bridging the growing crack gradually yield, neck, and break.

I folded a bag from aluminum foil (maybe 20 mm × 150 mm) and poured a few grams of waterglass into it. On heating it with the brass torch on a bed of vermiculite, it swelled and crackled, and the waterglass inside foamed up and eventually hardened, and then the aluminum on top melted off. I added another torch, double-fisting the butane. Eventually it seemed to become inactive, so I turned the torches off and turned it over (the bottom was still shiny foil), then heated it some more. Upon turning it a second time the vermiculite stuck to it in a syrupy mass, probably because the waterglass had poured out through the melted-open holes in the aluminum into the vermiculite. Heating the waterglass-cemented vermiculite cemented it together. As usual, the foam melted and collapsed rapidly whenever the torches brought it to an orange heat.

(It occurred to me that the vermiculite might be cemented with moist phosphoric acid from some previous tests, and there was indeed some acid-gas smell present, but I haven't actually seen any syrupy phosphoric acid going around. I got some of the syrupy vermiculite stuff on my fingers and didn't wash it off for several minutes, but have no burns, so it was probably waterglass.)

I was hoping that I could get a chunk of waterglass foam entirely covered with aluminum foil this way, but I think for that to work, I will need to heat it more gently — maybe quickly, as with hot sand, but to a temperature below the melting point of the aluminum.

I never did get any detectable skin irritation from rubbing my thigh with waterglass micro-bubble fragments last night.

The food coloring painted with detergent onto the flower Mina made yesterday is still wet, presumably due to the detergent. The mushroom I made is still much, much softer, especially at large deformations, than the other Oogoo I've made in the past without color or dish detergent, and so is the flower. Upon inflicting large deformations on them with my fingernail, I observe some white discoloration, which is probably permanent. I suspect that the dish detergent is acting as a sort of plasticizer. The feeling is sort of in the neighborhood of stretchy human skin, an illusion that becomes more

pronounced when I rub the mushroom with cassava starch (inspired by vague memories that the “Cyberskin” skin simulant is a silicone with talcum powder added.)

02021-08-22

I broke apart one of the chunks of cemented vermiculite by hand, finding white foam between the vermiculite grains, which I think confirms that it was leaked waterglass and not contaminating phosphoric acid. This is more evident when I cut it with a razor knife instead of breaking it by hand, because breakage by hand tends to cleave it along cleavage lines within the weak vermiculite grains.

Bought more dish detergent, the Unilever Cif version, also 20% surfactants, except they tell you what they are. AR$175 (US$1) for 500 mℓ, which works out to about US$10/kg for the sodium laureth sulfate etc., which seems a bit pricey to me.

02021-08-23

I checked MercadoLibre again today. Grupo Ecoquimica has raised all their prices by about 25–50%, presumably to compensate for recent inflation; for example, alum is AR$850/kg (US$4.70/kg) instead of AR$630/kg, and bentonite is AR$75/kg (42¢/kg) instead of AR$50/kg. It’s about 4 km away. I was planning to go, but didn’t.

The PVA glue I crosslinked with borax on 02021-07-28 has quite thoroughly dried out. I cut the chunk in half with a hacksaw, which took a couple of minutes (it’s about 20mm in diameter) and felt a lot like cutting a solid hard plastic, with no melting on the saw blade, a thermoset. It’s hard all the way through, and it smells slightly of PVA inside. This made me wonder whether it was in fact a thermoset.

Shibayama, Yoshizawa, Kurokawa, Fujiwara, and Nomura published a paper in 01988 which suggests that this slime definitely can gel, and higher pH raises the gel’s melting temperature, though they mostly measured gels that melted around 75° with $0.8–2.5 \times 10^{-2}$ mol/ℓ of boric acid. However, they propose that the borate is only covalently bonded to one of the PVA chains, while the other chelates a sodium ion, rather than forming a di-diol cross-linking bond, and neither that mechanism nor their melting point plots are particularly promising for the prospects of this material being a thermoset.

I heated a small piece of it on vermiculite with the brass torch. Its surface intumesced slightly with white bubbles at gentle heating, and it produced a terrible smell reminiscent of burning polystyrene, and a little bit of white smoke. Stronger heating charred the surface and produced a little flame, which self-extinguished in a second or two and did not continue smoldering. The uncharred material had stuck to the vermiculite and become noticeably plastic; I could pull it apart with my fingers. I conclude that it is not a thermoset, just another thermoplastic (if a particularly hard one, and one that forms a nice gel with water).

Even if it’s a thermoplastic, though, you could still use a PVA solution as a binder for 3-D printing in a powder bed of filler doped with some kind of borate.

A low-concentration thermoset hydrogel would be appealing for the waterglass-tape application mentioned earlier, a great improvement on agar, as well as for molding things with the appropriate fillers and for preceramic polymers and for producing carbon foam (or fibers). All I have available at the moment is wheat gluten; the internet suggests chitosan cross-linked with β-glycerophosphate, dicarboxylic acids such as citrate, glutaraldehyde, divinyl sulfone, epichlorohydrin, or electron beams as possibilities, or cellulose or chitin dissolved in more exotic solvents, which I guess is how rayon/cellophane is made. It seems likely that most materials that could covalently cross-link common biopolymers like starch or chitosan would be pretty toxic; apparently epichlorohydrin, monosodium phosphate, sodium trimetaphosphate, and sodium tripolyphosphate (sodium triphosphate) are commonly used for cross-linking starches. (Could the obesity pandemic be largely caused by modified food starch?)

02021-08-24

I heated a bit of baling wire in the brass torch to orange heat to melt some holes in the five bottom lobes of a plastic soft-drink bottle (Tomasso cola) to turn it into a gardening pot, which yielded curls of white smoke with a burning-plastic smell, but no flames. I burned the remaining PET off the wire with the torch, then set it down to cool on a piece of the waterglass foam, to which it did not noticeably adhere. Although it's only 57 mm long and about 1.5 mm diameter, the handle end didn't get hot throughout the whole process. This is the same wire I had Mina use to burn the mole off my forearm, but she had it gripped in the vise-grips rather than her hand.

I notice that the off-balance weight of the brass torch is making the butane can a little unstable now that it's nearly empty.

It occurs to me that maybe some metal hydroxide could serve as the hydroxyl donor for the polymerization of acetic-cure silicone, and copper hydroxide (which I've made previously by electrolysis) seems like an ideal candidate because of its laid-back enthalpy of formation (-225 kJ/mol OH) and associated low decomposition temperature. By contrast, sodium hydroxide is -425.8 kJ/mol and doesn't boil (decomposing into the elements, I think) until 1388°, and aluminum trihydroxide (78.00 g/mol, 2.42 g/cc, gibbsite, dehydrates in the range 180°–300°) is -1277 kJ/mol, or -426 kJ/mol OH, same as the sodium salt.

Hmm, https://iupac.github.io/SolubilityDataSeries/ is an intriguing URL! Apparently IUPAC has put their solubility data series on GitHub, as well as a number of other datasets, although the link to the source repository seems to be broken. Copper hydroxides are in volume 23. About half of the volumes have analogous URLs; many others (5, 6, 16, 17, 27, 28, 45, 46, 53, and 67–104) are missing.

IUPAC's solubility data is from 01986 and suggests using lye or ammonia to solubilize copper hydroxide. It has this lovely epigraph at the beginning of the Foreword:

> If the knowledge is undigested or simply wrong, more is not better[.]

This seems to be a jab at the CRC Handbook:

> On the other hand, tertiary sources — handbooks, reference books and other

tabulated and graphical compilations — as they exist today are comprehensive but, as a rule, uncritical. They usually attempt to cover whole disciplines, and thus obviously are superficial in treatment. Since they command a wide market, we believe that their service to the advancement of science is at least questionable.

02021-08-25

I bought a kg of electrical copper from the recycler around the corner for AR$1200 (US$6.70/kg). Some of it is very fine (presumably enameled wire from CRT yokes) while other parts are stranded, thicker, and uninsulated, measuring 0Ω on the multimeter. I cut a 1-meter section of a single strand of the uninsulated cable, which measures 0.51–0.57 mm in diameter with the calipers at different places along the strand, thus having a volume of about 0.20–0.26 cc. It weighs 2.1 g, which would put it in the density range 8.2–10.3 g/cc. It can be easily bent by hand.

It melts on the bed of vermiculite at a red-orange heat from the butane torches, a heat which is difficult to attain without placing some heat-reflecting waterglass foam behind the wire, at which point the waterglass also starts melting. Upon heating, it forms a hard, adherent black oxide coating. The melting is sharply defined, so that an intact wire forms a small sphere of liquid at the end of it. No visible fumes are emitted.

All of these characteristics are consistent with the metal really being copper. The density is in the right range; measuring the density of a larger piece of metal, or using a less shitty scale, would give me a more precise number. Zinc (from brass) would boil at 907°, emitting white fumes, and bronze would partly melt around 798°, or, in some flavors, completely melt at below 750°. Either alloy would melt gradually rather than suddenly, and both would probably be harder.

The melted copper wetted the vermiculite grains enough to stick firmly to them, suggesting that it ought to be possible to use copper (in a reducing atmosphere!) to repair broken pottery in the same way as silver and gold. To confirm this, I tried melting a little more copper wire on a bed of sand, and indeed the little ball of melted copper got sand stuck all over it, much of which I could not remove with my thumbnail. The effort abraded away much of my thumbnail stickout, in fact. I attempted to test the hypothesis directly by putting a couple of broken bits of ceramic floor tile with copper wire between them into an oven improvised from bits of waterglass foam, but although the glaze on one of the tile bits melted and flowed a bit, I wasn't able to get them hot enough with the torches to melt the copper. They glowed orange. I *was* able to melt copper wire onto the bottom of one of the tiles by itself, a little, but it was relatively easy to dislodge after it cooled.

In the process I lost the aluminum foil on the bottom side of one of the waterglass pieces from the loaf I baked the other day; it melted away, along with a good portion of the thickness of the waterglass. Some of the waterglass also melted onto the top of one of the ceramic tile fragments, though I was able to dislodge it later.

The flames shooting out of the gaps in the oven were bright green, which might be from something in the ceramic (especially its glaze) or (less likely) the copper or waterglass.

During this process there were some acid fumes and some that smelled a bit like some kind of burning plastic, so I probably ought to throw out this vermiculite bed. Now I have both slightly irritated lungs and throat and a bit of a headache.

It occurs to me that if I can disperse a fine dust of a water-insoluble inert source of polyvalent cations in the waterglass, something that remains fairly inert at the low temperatures needed to soften and foam up the waterglass but then gives up its ions when the waterglass is facing real fire, it would probably work better than just applying oil of lime to the surface of the foam. Candidates include talc, aluminum trihydroxide, bentonite, copper oxides or hydroxide, finely ground enstatite, dehydrated borax, chalk, kaolin, rutile, or zincite. Amorphous silica such as diatomaceous earth might also work, but in a different way, by dissolving into the warm anhydrous sodium silicate melt and merely diluting the sodium rather than displacing it.

It occurred to me that an acid electrolyte like vinegar is not going to be useful for producing copper hydroxide, because it will convert essentially all the copper hydroxide into (soluble) copper acetate. So I resolved to get a vitriol electrolyte, but what I have handy is green vitriol, which might produce undesirable deposits of its own, maybe insoluble iron hydroxide, contaminating the copper hydroxide.

So I heated up some baking soda in tap water on the stove; it started bubbling as soon as I started applying heat, indicating that it was decomposing to soda ash. I added a little unheated baking soda to a near-transparent green solution of iron sulfate fertilizer in tap water, resulting in the formation of the usual nasty green muck with lots of bubbles. After the baking soda had boiled for a while, spoonfuls of the solution still bubbled enthusiastically when vinegar was dripped into the spoon. But I added the hot solution to the iron etc. mix and got only a little more bubbling, presumably due to the accelerated decomposition of the remaining baking soda. Bubbling slowed and the solution began to defecate after a few minutes.

Now I realize that because I didn't measure anything my solution contains an unknown mix of soda and sal mirabilis, plus the nasty green iron carbonate that's precipitating out. If I use that to electrolyze the copper I'll end up with verditer, which is the usual blue-green pigment and which decomposes to tenorite on heating, so I should be satisfied with that. But in that case I might as well just use the soda as the electrolyte directly without the sal mirabilis.

I turned off the flame for a while, allowing the soda to cool a bit and crystallize into a single large mass which broke up and partly dissolved readily upon the addition of more water. Reheating produced no further bubbles, indicating that the baking soda was essentially all decomposed. There was a small black speck floating in it, probably from the defective faucet. I spooned a bit of the menstruum into the defecated vitriol product, which precipitated no further visible solids, indicating an excess of soda. Then I decided to dehydrate the soda, for which purpose I added aluminum foil over the top of the container. After a few minutes of further heating, pops were heard, so I lowered the flame further. Pops continued; upon inspection the soda has crystallized into a mostly porous mass, which has a few dozen holes in it where steam explosion has blasted

fragments elsewhere.

The internet suggests that the green material may be iron(II,III) hydroxycarbonate, “carbonate green rust”. (On previous occasions, after filtration and drying, it turned into a very fine purple-brown powder.). After a few minutes it has defecated into a well-defined layer at the bottom of the jar (about 12 mm deep out of 60 mm, if the glass is 3 mm thick), which appears solid but is actually a fine suspension that offers no resistance to being stirred with a knife. The water above it is green and cloudy, but basically transparent.

I heated a piece of brass-wool dishwashing sponge (<500mg) with the brass torch as a control, to see if it behaved the same as the copper. It seemed to be, if anything, *harder* to melt, but it did emit a little white smoke, and did not ball up due to crossing sharply-defined melting points, but rather wilted. It did form the same kind of black oxide coating, though. However, a white surface deposit (presumably philosopher’s wool) was evident on the lower layers of the result. The burned sponge was crumbly rather than solid as the melted copper wire had been, perhaps because it was thin enough to oxidize most of the material rather than just a superficial coating.

After a few minutes more of heating, the pops and hissing from the soda ceased, and I was left with a bright white mass of soda, which had crept up the walls a bit. It had delaminated from both the floor and walls of the thin stainless steel pot, and indeed evidently detached from the pot in a single block. As it cooled, it crackled softly.

I flaked some soda off the wall and dripped some oil of lime on it in my hand, then let it to soak in my hand for a few seconds. No reaction was evident, not even warming, but afterwards the flakes took multiple washings over several minutes, clouding the water each time, to dissolve completely in water. By contrast, other similar flakes mostly dissolved in water in a few seconds and entirely in a minute or two. This suggests that fairly insoluble chalk was formed, though not enough in a few seconds to withstand multiple washings; chalk is not as strong as the phosphates of calcium, but still a viable building material. If you wanted to do this with an inkjet printer, calcium dinitrate might be a less corrosive source of calcium ions than oil of lime.

Using a cut-off end of construction I-beam as a handheld anvil, I was able to hammer one of the balls of copper flat, to perhaps 10× its original area and 1/10 of its original thickness, without any difficulty, although it cracked a bit around the edges.

96% ethanol is sufficient to remove residual adhesive from Emeth Línea Gourmet jam jar labels, though it requires several seconds of soaking and usually multiple tries.

I ground up the dried soda in a marble mortar and stored it in another Emeth jar.

Oh, I realize I never recorded the ingredients of the Cif dishwashing detergent, whose bottle looks like it’s made in the same factory as the Magistral, maybe with the same blow molds: the surfactants, both anionic, are linear sodium lauryl ether sulfate and sodium alkylbenzenesulfonate. The other ingredients are “pH regulating agent, coadjuvant, viscosant, preservatives, sequestrant, colorants, and perfume.” Pretty shitty labeling if you ask me.

Mercado Libre offers 5 kg of technical grade sodium lauryl sulfate for AR$7430 (US$41, US$8.30/kg), which is pretty close to the US$10 I was guessing I was paying at the supermarket, without having to dump the formaldehyde in it myself.

Because the Wikipedia article on green rust mentions that it normally oxidizes to iron oxyhydroxide, which is not green but more brown, and because the stuff I'd made previously turned purple-brown, I sucked a little of the green muck into a tube and put it in a bowl, then added some drops of 30-volumes hydrogen peroxide to it. It immediately turned brown-black and foamed up a lot, which is consistent with the identification as carbonate green rust. It would be better to use an oxidizer that isn't itself inclined to foam up, I guess, so I could be sure that it's really releasing the carbonate.

The weird thing is that the syntheses described in Wikipedia are a lot more involved than just dumping an excess of soda ash on some green vitriol in a jar, so I wonder if there are impurities in my fertilizer that predispose it to this. Or maybe this hydroxycarbonate isn't very pure.

This sounds more like the reaction described in the (nearly insoluble) iron carbonate article:

> Ferrous carbonate can be prepared also from solutions of an iron(II) salt, such as iron(II) perchlorate, with sodium bicarbonate, releasing carbon dioxide:
>
>
>
> $Fe(ClO_4)_2 + 2NaHCO_3 \rightarrow FeCO_3 + 2NaClO_4 + CO_2 + H_2O$
>
> Sel and others used this reaction (but with $FeCl_2$ instead of $Fe(ClO_4)_2$) at 0.2 M to prepare amorphous $FeCO_3$.
>
> Care must be taken to exclude oxygen O_2 from the solutions, because the Fe^{2+} ion is easily oxidized to Fe^{3+}, especially at pH above 6.0.

Which, I mean, that's pretty much what happened.

The hydroxycarbonate was a little stinky, same as last time. I wonder if that means the green vitriol is slightly contaminated with, say, iron sulfide.

02021-08-26

The hydroxycarbonate has entirely defecated, leaving what seems to be a perfectly clear menstruum, except that some of it is stuck to the walls of the jar. Siphoning this water out may usefully substitute for a stage of filtering if the objective is getting the solutes.

It occurred to me that in fact electro-etching of copper in soda ash solution may not work, precisely because copper carbonate is insoluble. If so, the sal mirabilis in that jar could be crucial.

02021-08-29

The cuboid of foam I put in to dehydrate a few days ago now weighs 700 mg, which I guess means that it wasn't fully dehydrated at 900 mg. The bowl seems to have remained reasonably well sealed with the foil; the flakes of muriate of lime that were slightly damp before have now become almost dry, just slightly shiny, and very firmly stuck to the bottom of the bowl, and slightly discolored with what looks like iron oxide rust. Upon immersing the cuboid in tared water the scale read 7.0, indicating that its extra-pore volume is 7.0 cc; upon removing it from the water, the scale read -1.1, indicating that 1.1 cc of water had entered its pores; upon weighing

the wet cube I get 1.8, which is consistent with 1.1 g of water plus 0.7 g of cube. So this foam cuboid, when dry, is actually about 8.1 cc and 86 mg/cc. Upon leaving it on the scale a little while, turning the scale back on, and picking it off, the scale reads −1.7 g, so about 100 mg of water seeped out of the foam; upon drying the scale with a paper towel (should have used a sample boat!) I get −1.8 g as expected.

Unfortunately I didn't weigh the bowl of muriate of lime before adding water to it to get rid of the muriate, so I can't heat up the muriate to see how much water I drive out of it (most of which was presumably obtained from the cuboid).

To get relief from gastric reflux, I drank a little dilute washing-soda solution earlier. It was bitter, left my tongue sort of tingling in a soap-like fashion, and left my throat more irritated than before, so I probably should stick to baking soda. (I took some baking soda a little bit later.) I was thinking it might be interesting to try to saponify some butter with that soda, but apparently this involves boiling it for 2–5 days, so I don't think I'll do that. Maybe I'll make some soap with lye, but not tonight.

Maybe I can make boric acid (61.83 g/mol). The usual borax is the decahydrate (381.33 g/mol) and decomposes to the anhydrous form (201.22 g/mol) at 75°, which then doesn't melt until 743°, a temperature with a deep-red glow. In water you can dissolve 3.17 g/100 mℓ, I guess at room temperature. Boric acid, by contrast, doesn't have a hydrated form and melts at 170.9° (well, dehydrates to the almost insoluble metaboric acid, which melts at 236°), but dissolves about 2.52 g at 0°, 4.72 g/100 mℓ at 20°, rising to 27.5 g/100 mℓ at 100°. I don't really understand pKa, but I think that because boric acid's (first) pKa is 9.24, acetic acid (60.052 g/mol, pKa=4.756) should be able to convert all but about 0.003% of the borate in borax into boric acid by stealing off its sodium for the highly soluble sodium acetate (82.034 g/mol, 119 g/100 mℓ at 0°, 123.3 g/100 mℓ at 20°, 162.9 g/100 mℓ at 100°).

The stoichiometry here is a little confusing. I need one acetate per sodium, and borax ($Na_2B_4O_7{\cdot}10H_2O$) has two borons per sodium, so I guess I have two borons per acetate. But that tetraborate group needs some extra hydrogen to break it up into loose boric acid molecules, namely three hydrogens per boron, plus some extra oxygens (5 oxygens per tetraborate) but acetic acid only provides one hydrogen per acetate and no oxygen. So I guess we have to get our other hydrogens and oxygens from the water: $Na_2B_4O_7{\cdot}10H_2O + 2CH_3COOH \rightarrow 5H_2O + 2NaCH_3COO + 4B(OH)_3$.

So if I get a reaction product that crystallizes at anything less than overwhelming concentrations, even at 0°, it's either borax or boric acid, and if the solution is still acidic, it's boric acid. And ideally I should be able to get about 90% of the boric acid to crystallize if I use a minimal amount of water to dissolve it. (Even if I were to crystallize it just by cooling down an excessive amount of solution, the yield ought to be above 50% as long as I have less than 100 mℓ of water per 5.04 g of boric acid.) Anhydrous sodium acetate's boiling point is listed as 881.4°, so as long as I don't boil the stuff dry (and then overheat it to red hot), it ought to be stable.

I don't have a simple way to find out how much acetic acid is in my vinegar, though, so I'd have to add an excess; acetic acid is entirely miscible with water and boils at only 118°.

I guess I could see roughly how much vinegar is needed to neutralize, say, a gram of baking soda (84.0066 g/mol). (This would be a lot easier with a better scale, but I guess I'll just be very approximate instead of carefully titrating.)

I added 1.0 g of baking soda to a tray (which initially weighed 2.2 g). I added vinegar to a cup that initially weighed 8.2 g, with a final weight of 53.7 g, so 45.5 g of vinegar. By the time the cup weighed 35.5 g (27.3 g of vinegar) the baking soda seemed to all be neutralized (by the 18.2 g of vinegar I added), but not much before that. Let's say somewhere between 13 g and 18.2 g.

In theory this is $NaHCO_3 + CH_3COOH \rightarrow CO_2 + H_2O + NaCH_3COO$, a 1:1 mole ratio between acetic acid and baking soda, so the 1.0 g of baking soda corresponds to 60.052/84.0066 = 0.715 g of acetic acid, but given the weighing imprecision of the scale, somewhere between 0.68 and 0.75 g. Continuing the interval arithmetic, that puts us between 3.7% and 5.8% acetic acid in this vinegar (*Alcazar vinagre de alcohol, acidez 5%*, UPC 7-790130-000294). I could probably take a second measurement with more careful titration to nail that down to something like "4.3% to 5.3%" but I'm not going to, because this is precise enough to allow me to ensure an excess of acetic acid with the borax without adding an absurd amount of extra water, and I can harmlessly boil off any extra water without losing any boric acid.

So suppose I want 100 g of boric acid, $B(OH)_3$. I'd need 364 g of water to dissolve it in. This bowl I used to dehydrate the glass foam weighs 42.5 g empty (it's probably the bowl that weighed 42.3 g before) and 562 g with an unreasonably large amount of water in it, an amount almost certain to spill. But looking at the equation above, every mole of borax (381.33 g) requires two moles of acetic acid (120.104 g) to neutralize it, producing four moles of boric acid (247.32 g) and two moles of sodium acetate (164.068 g). But 120.104 g of acetic acid at my lower-bound 3.7% concentration would be 3300 g of vinegar, and I have less than 500 g here, and at any rate if I used that much it would spill everywhere when it came to a boil.

So suppose I use a more reasonable 200 g of vinegar, providing at least 7.4 g of acetic acid. That should suffice to neutralize 23 g of borax, producing 15 g of boric acid and 10.1 g of sodium acetate. If I then just stick it in the freezer, the 193 g of water will only be able to dissolve 4.9 g of it, for a theoretical 67% yield, but if I boil it down to 100 g or 50 g or something, I should be able to improve that significantly.

I weighed out 20.0 g of borax into an improvised aluminum-foil sample boat, put it in the bowl, and weighed out 200 g of vinegar, then put it on a low flame. Some of the borax seemed to dissolve in the vinegar at room temperature, but, as predicted, not most of it. It finished dissolving at about 01:17 as the vinegar warmed up a bit, at a point where it was too hot to hold my finger in but did not burn my finger in a second or two, perhaps 40–45°. I continued heating it on low heat a while longer, then decided to go for broke and boil it

rapidly with an aluminum-foil lid. At 01:23 it weighed 244 g without the foil lid, 246 g with it. At 01:25 it weighed 232 g, and the foil seemed to be doing a reasonable job at preventing liquid drops from spraying out.

At 01:30 it had boiled down to 167 g (remember, 43 g is the bowl, and 1 g is the lid). I took off the lid to see how it looked, like if anything had precipitated, but it just smelled very acidic and looked clear. Unfortunately I tore the lid, which reduced the weight to 163 g, so about 3 g of condensation and spray went with the previous lid. I made a new lid (also about 1 g) and returned it to the fire.

7 minutes to drop from 244 g to 167 g is 11.0 g per minute, or more realistically 8–16 g per minute. If we assume that basically all the power is going into boiling water, at water's enthalpy of vaporization of 40 kJ/mol and 18 g/mol, this is about 400 watts.

At 01:36 it was starting to sound syrupy, so I turned it off. It weighed 98 g, 96 g without the lid, which stuck a little to the bowl as I tried to remove it. Some drops that had splashed onto the side of the bowl had crystallized into white spots, but they dissolved rapidly when I sloshed the hot syrup over them. I put the mix into the freezer. There was a strong smell of acid.

96 g should be about 53 g of solution, of which about 10.1 g should be sodium acetate, about 13 g boric acid, some undetermined but fairly significant amount should be acetic acid, some insignificant amount should be borax, and the rest (less than 29.9 g) should be water.

At 01:45 I looked in the freezer; there is no visible precipitation yet, except possibly some dust floating in the syrup. Its viscosity seems unchanged.

At 01:56 there was a lot of translucent white crystallization on the bottom and some more floating around in the syrup, which actually seems maybe a bit less syrupy now. I made a gravity filtration setup out of a couple of Monster cans and two coffee filters and stuck it in the freezer to cool. Empty, it weighed 27.6 g, but then I realized ⓐ it actually had a drop of water in it and ⓑ was too tall to fit in the freezer and ⓒ at any rate I was going to be filtering 40 mℓ of solution and not 400 mℓ so I could cut it down a bit. So I cut it down to 24.0 g.

At 02:06 the fluid is quite milky from crystallization and seems to be about the same viscosity as before, and it still smells intensely of vinegar. My plan is to dump the liquid into the filtration funnel, then rinse the crystals stuck to the bowl with some 0° water, passing that water also through the filtration funnel, and then set both the bowl and the filters out to dry under a loose covering for a few days.

At 02:18 I did the filtering, but didn't do a very good job of record keeping because I was trying to do it all before things warmed up and increased solubility. If I recorded this correctly, though, the filter funnel setup initially weighed 24.1 g, and I think the bowl initially weighed 77.8 g. After pouring the milky menstruum into the filter funnel, it weighed 38.9 g, but I didn't record the bowl's weight; it must have been about 63.0 g. I added some 0° water to the bowl, bringing its weight up to 86.0 g, which seems like a lot more water than I should have added; after I poured that into the filter funnel

too, the bowl was down to 72.7 g, and then I knocked over the filter funnel, spilling some of the menstruum onto the table and bringing its weight down to 45.2 g. I added more water to the bowl, bringing it up to 80.2 g, and dumped that into the funnel, reducing the bowl's weight to 69.5 g, and bringing the funnel's weight up to 58.4 g. I should have weighed the water bottle at the beginning, too!

Afterwards the water bottle weighed 514 g; it's a nominally 500 mℓ Coke bottle, of a type which weighs 24 g including the PCO1881 cap. On refilling it until only a small bubble is left, as before, it weighs 555 g. So the total amount of cold water I added was somewhere in the neighborhood of 30–45 g.

Now I realize I should add some extra rinse water to the filter funnel to wash water-solubles out of the filters. I first reduced the weight of the cold-water bottle so I could weigh it on the scale at 390.0 g, and on adding some of it to the funnel apparatus, its weight increased from 58.4 g to 105.2 g, leaving the bottle weighing 345.2 g, so I transferred 44.8 g or 46.8 g of water from one to the other. One of those numbers must be wrong, because I doubt I spilled 2 g of water or got it stuck to my hands, but on weighing the filter apparatus again it weighs 105.0 g, and the bottle still weighs 345.2 g. So, I don't know what went wrong, but it's probably safe to say I added on the order of 44–47 g of water to rinse the funnel.

105.0 g amounts to 66.1 g of rinse water, which will probably have stolen about 1.7 g of the boric-acid product from the funnel. It's a miracle there's anything solid left in that coffee filter, but there is.

To see if the wet product in the bowl is really boric acid, I spooned about 0.2 g of it with a chopstick onto a 3.9-g bottom of a Monster can and turned on the stove. At first it dissolved a bit as the water heated, then formed a white apparently amorphous foam as the water boiled. I scraped a little powder off the foam with a chopstick, which landed on the aluminum. No further changes were seen to the foam or the powder as I turned up the heat, though the epoxy inner liner of the Monster can started to smoke white, and my lungs started to criticize me. I blew a butane torch over the top of it to heat it further and hopefully burn off the smoke, but this mostly had the effect of heating parts of the foam to orange-hot and melting the aluminum underneath it, at which point I turned off the fires and opened the window.

This is uninspiring, more like what I had seen previously from heating borax (in a less stupid and dangerous way, using aluminum foil) than what I was hoping for from boric acid. There was never a point where it convincingly melted, even though it got to red heat. The only detectable difference is that it's black instead of white (from the smoke from the charring epoxy) and it's perhaps a bit harder (but maybe I reached borax's melting point and thus collapsed superficial bubbles, reducing porosity).

The remaining prediction to test, I guess, is that, after heating the "boric acid" enough to convert it to metaboric acid or boria, it should not dissolve in water; the solubility of the borax foams, by contrast, was one of their most striking properties. Accordingly, I added most of the blackened remains of the "boric acid" foam to a cup of warm (≈40°) water. In a minute or so, they dissolved into black sludge,

which breaks up when the water is stirred. So I think what I have here is still just borax, contaminated with the toxic decomposition byproducts of epoxy. I threw it away.

Other candidate tests suggested in some random lab handout include a flame test (which should be bright yellow for borax, transparent green for boric acid) and conductivity in distilled water (which I think ought to be very high for borax and very low for boric acid, though the lab worksheet doesn't actually say, instead being phrased as an exercise). The lab procedure used 1M vitriol rather than vinegar but was otherwise equivalent.

I tried the flame test and got a bright yellow flame that looked like sodium. However, I also got yellow flames from my iron wire and carbon welding electrode when they were supposedly clean, though maybe the yellow was a little less bright. I also contaminated the bowl of sample with the carbon welding electrode; when I dipped it in while hot, it scattered carbon particles around.

A much simpler test would be to heat dry "boric acid" past 200°, weighing it before and afterwards. If it's boric acid, which is 61.83 g/mol, heating to even a few hundred degrees will only drive off the hydrogens, reducing its weight by about 4%. If it's borax, heating it past 75° will dehydrate it from 381.38 g/mol to 201.22 g/mol, a 47% reduction in weight. But I cannot do that until it is dry. A differential scanning calorimetry/thermogravimetric analysis rig would be extremely helpful for this kind of thing.

It's 03:40. The filtering apparatus has mostly frozen; the funnel part has a little icicle hanging down, and the receptacle in the bottom is mostly a block of ice, with a little of some kind of syrupy liquid on top of it, which is plausibly a low-melting solution of water, acetic acid, borax, boric acid, and/or sodium acetate. I guess that means it's not doing any more filtering at 0° or -20° or whatever. So I took it out of the freezer and left it to thaw on the table, with a little aluminum foil over the top to keep dust from falling in. The idea is that when it melts, a little more filtration will happen, and then I can remove the filters from the funnel and lay them out to dry.

I found an old article about boric acid purification and a new CC-BY article about it. The current article mentions acidifying borax with vitriol, aqua fortis, oxalic acid, and propionic acid, but not acetic acid, and also using membrane electrolysis.

The *old* article, however, *does* mention acidifying with acetic acid — but followed by a stage of evaporating to dryness, then volatilizing the result with *methanol*! Methanol seems to be available for AR\$190/ℓ (US\$1.06/ℓ) as a gasoline performance additive for hot-rodding, so maybe I can find it at an auto parts store, but it seems like it's a more specialty product than in the US, and the HEET brand name (for its use as a gas-line antifreeze) is nonexistent.

However, Gooch seems to be concerned not with purifying boric acid but with measuring the amount present, so he ends up turning it into calcium borate to weigh it. On the other hand, he has a *lot* of helpful tips about this sort of work, mentioning for example "the exceedingly delicate test with turmeric" for detecting residual boric acid. This is mentioned in the ScienceMadness Wiki:

Boric acid reacts in alcoholic solution with two molecules of curcumin to form

rosocyanine, a dark green ionic solid that forms deep red solutions.

It seems like Gooch was forming (and distilling) trimethyl borate, which boils at 68°, but had no way to find that out.

It looks like boric acid is highly soluble in methanol and ethanol (even without reacting with them), while borax is only slightly soluble in ethanol. So that might be a thing to try.

I took out another sample of 0.2 g or so from the bowl and dripped 96% ethanol on it on a scrap of aluminum foil, letting the somewhat milky liquid run onto a different scrap. I set the second scrap on fire, and got what looked like a pretty normal yellow blackbody diffusion flame with a green aura around it, which at least vaguely suggests some boric acid might be present. Eventually the alcohol burned away, leaving a white residue. On heating this foil on the stove, the white residue turned black, even though there was no epoxy available to contaminate it. I suspect that this was sodium acetate being charred, which means that I didn't rinse the bowl very well. (Gooch mentions that sodium acetate is soluble in methanol; apparently ethanol also dissolves 5.3 g/100 mℓ of the trihydrate.)

02021-08-31

The borax/vinegar material in the bowl has solidified and smells strongly of vinegar, indicating again that it was inadequately washed. The filtration funnel still has sediment in the filters; the menstruum it filtered remains clear (but is hardly exposed to evaporation).

Yesterday I ate a can of mackerel and washed it out, but it retained a strong fish smell. I heated it full of water and sodium percarbonate on an electric burner, which produced froth. There was too much sodium percarbonate to dissolve, which restricted the flow of water near the bottom; this may have been a factor in the plastic can lining bubbling up along the bottom where it was over the heating element. It still smelled strongly of fish. I left it full of dilute household ammonia overnight, but today it still smells strongly of fish. Now I have put it full of dilute household bleach (taking care to rinse the ammonia out first).

To a cup cut from the bottom of a Monster bottle weighing 8.1 grams I added 40.4 g of the somewhat wet copperas fertilizer, removing a large crystal that somehow got in there (presumably of copperas, but possibly something else). 48.4 g was the final weight, so maybe that was more like 40.35 g. I added 151.2 g of tap water, which mostly dissolved the copperas immediately, though the solution was a bit murky. Applying gentle heating does not seem to remove the cloudiness, so it's probably insoluble particulate contamination, maybe the pyrites I suggested earlier might be aromatizing my iron precipitation products. Actually, gentle heating seems to have turned it an opaque green-brown! But there was no acidic smell, so the heat surely isn't decomposing sulfate ions.

Now I realize I should have tried heating it to dryness first to quantify how much water there was in it; this ought be the heptahydrate plus a bit.

To a yogurt cup weighing 7.1 g I added an egg white; afterwards it weighed 41.6 g, suggesting that I have 34.5 g of egg white here.

I turned another Monster can into a filter-funnel setup and filtered the warm copperas solution through it, spilling a little. Much to my surprise, there's a green translucent mud at the bottom, apparently copperas crystals. The filtrate seems to be just as opaque as the solution, so evidently this coffee filter is not doing a good job at filtering out the particles, which I now realize may be precipitated copperas.

To the can with the sparkling green sludge, now weighing 23.3 g (16.2 g sludge) I added 81.1 g of water. The crystals began to dissolve immediately, but were not completely dissolved after a few minutes when I spilled the whole thing on the floor. This, however, revealed that some of the crystals were stuck together in pale blue-green granular masses, presumably from when I was heating them; the sludge had hidden them previously. These masses broke into a few pieces when the can hit the ground. The can, unspilled green water, and crystals weigh 25.8 g (17.7 g sludge and water); I added another 21.3 g of tap water to try to dissolve them.

So evidently what happened was that I prepared a saturated solution of copperas with some minor particulate contamination, then evaporated it (failing to increase the solubility enough to desaturate) to precipitate lots of tiny copperas crystals, which are now clogging my coffee filter. But the 151.2 g of tap water I added was evidently capable of dissolving at least 40.4−16.2 = 24.4 g of the copperas crystals, and probably more, since some of that 16.2 g of sludge was water. This works out to at least 16.1 g/100 mℓ but less than 26.7 g/100 mℓ; WP says its solubility in water is 20.5 g/100 mℓ at 10° and 29.51 g/100 mℓ at 25°, ranging up to 51.35 g/100 mℓ at 54°. So I probably shouldn't have been so ambitious with the quantities.

The granular masses and crystals to which I added the 21.3 g of water mostly dissolved, so I poured the cloudy green solution to the funnel, leaving only a single crystalline mass stuck to the side (dropping the mass from 46.8 g, including the can, by 38.1 g, to 8.6 or 8.7 g) and added another 24.4 g of tap water, which rapidly diminished the size of the polycrystalline chunk, which dissolved in a minute or two. The pale green solution is still fairly cloudy, but I added it to the filter. The empty can weighs 8.2 g and has some green deposits on its walls.

I poured the 228.8 of filtered liquid back in to the 8.1 g (!) empty can and thence back through the filter funnel, except for some which I lost. I'm hoping that the particles clogging the filter will provide me with finer filtration this time around.

It looks slightly clearer maybe. I put 1.9 g of the egg white into the empty can and added 7.5 g of the slightly clearer filtrate, for a total weight of 17.6 g. Yellow-brown, slimy transparent masses started to coagulate in the solution; I was hoping for a precipitate but I was not expecting it to be either brown, transparent, or slimy, but rather hard and granular, with an octahedral crystal habit.

There is still considerably more particulate in the filtrate in the funnel setup than in the copperas I'd started with, although I've surely added much more water than evaporated, so I'm guessing that whatever particulate formed when I heated the copperas solution, it wasn't just precipitated copperas. It might conceivably be that I

oxidized some of the copperas, which would maybe explain the browner color (it has a browner green color) but that shouldn't precipitate it; rather the contrary, WP gives ferric sulfate's solubility as 25.6g/100 mℓ.

The particulate in the funnel is a yellow-brown, maybe even a baby-poop yellow, very different from the blue-green of the copperas itself. After 2½ filtrations through the same funnel the solution seems to be getting a little clearer.

After 3½ filtrations it is definitely getting clearer.

I added a spoonful of salt and nine spoonfuls of vinegar to half a cup of milk, which began to curdle into tiny curds. I think it started to thicken around six spoonfuls; at the end, it was very sour. I set up another Monster can as a filtration funnel with a coffee filter and used that to filter the curds from the whey.

I tried to remove the full coffee filter from the funnel, but it split, and I had to start again. I spilled a little of the cheese. The restarted filtration yielded a great deal of acid whey, somewhat curded at first but later almost perfectly clear, which I neutralized with baking soda and drank, despite its excessive saltiness and perhaps excess of bicarbonate, which I then tried to correct with a little lemon juice. Finally I diluted it with Diet Sprite. A white powder that looks like baking soda was visible at the bottom of the cloudy liquid, but the liquid was drinkable without difficulty. Indeed, on adding vinegar to the white powder, it fizzes! So it was at least partly undissolved baking soda.

I managed to get a little cheese off the first, broken filter, and scraped it off a plate. It was quite agreeable, despite the strong vinegar flavor. This is definitely a process to keep in mind if I ever have a fridge full of milk in a power outage. Presumably I can leach the vinegar out once I have it solid; I wonder if I could preserve the milk with acid and sugar (and, say, chocolate) rather than salt? Maybe some citrus terpenes?

Something has gone wrong with the water supply, so I suppose I will allow the various materials to repose until tomorrow; I no longer have a way to wash my hands after handling them. Hopefully the egg white will not putrefy. The little funnel full of cheese is gradually dripping at about 2 Hz.

After an hour or two, the cheese finished draining. It is still very soft.

A few hours later the water supply is back. After 4 filtrations the filtrate is clear, looking like fake apple juice — still not the blue-green color of copperas crystals, but definitely yellow-green rather than brown.

02021-09-01

The copperas solution looks like slightly cloudy urine.

To the can where I had combined egg white with copperas before, which looks like brown phlegm I coughed up once when I was sick, I added the remaining 29.0 g of egg white, then 100.0 g of the copperas solution. More ropy brown precipitate formed immediately.

Soaking the mackerel can in dilute bleach since yesterday seems to

have effectively deodorized it, which ammonia and (separately) sodium percarbonate failed to do. There *is* a little chlorine odor left.

Adding oil of lime to saturated clear solution of baking soda produces effervescence and a precipitate, just as with ferrous sulfate and baking soda; this despite WP claiming baking soda's solubility is 9.6 g/100 mℓ and calcium bicarbonate's is 16.6 g/100 mℓ. The precipitate appears white, being presumably chalk. I suppose that doing this in a dilute enough solution might produce large calcite crystals. See Fast electrolytic mineral accretion (seacrete) for digital fabrication? (p. 779). I diluted the results with water, and the chalk settled within a few hours, but remained impalpably fine. It adhered to the surface of the aluminum can a bit and did not immediately dissolve in vinegar, but did dissolve with mild heating of the vinegar.

By contrast, adding oil of lime to Diet 7-Up (*not* Sprite) produces immediate effervescence but no precipitate, just as you would expect: even though 7-Up is a saturated solution of CO_2, it does not react to form $CaCO_3$.

The large dark green crystal I fished out of the copperas yesterday has now dried to a sparkling light green; it looks like someone sprinkled sugar on wasabi.

02021-09-02

The cheese is no longer wet, just moist. It is delicious, similar to an acidic ricotta.

The label of Doreé [sic] Capilar 30-volumes hydrogen peroxide (UPC 7-794050-007050) peels off the polyolefin bottle without leaving an adhesive residue. 95% ethanol removes the lot number and expiration date. Now I have a 100mℓ hermetic polyolefin bottle.

The baby-poop brown precipitate from the copperas on the filter has now turned brown. Adding 30-volumes hydrogen peroxide to it does not change its color. This suggests it may have oxidized from ferrous to ferric when I was warming it up, thus reducing its solubility (?).

02021-09-03

The dried bowl of borax vinegar is actually sort of soft and slushy. I placed a little on a pebble of pumice, skewered on baling wire, and heated it with the brass torch; it melted rapidly down into the pores of the pumice, and with further heating converted the pumice surface into porous glass. There was a strong smell of vinegar. The vinegar charred into little balls of black on the surface of the pumice. Then the piece of pumice cracked, and a couple of flakes gradually hinged away from the main block and then fell. A third crack remained open by a couple of millimeters. Did the acetic acid attack the pumice at high temperature, opening or widening cracks? Did the borax?

I placed some oil of lime on the pumice, which soaked in. On further heating I saw an orange flame, and the pumice cracked again, right through the middle, held together only by the baling wire.

I took an additional piece of pumice and dunked it in tap water, then heated it in the same way, alternating between dunking and heating a few tims. It did not visibly crack, but on tapping it on a

ceramic plate afterwards, a piece fell off, evidently having been previously severed by an invisible crack.

The surface of the pumice reached orange and yellow heat during this procedure, but only melted where borax had been applied.

There was a bit of the "cockroach smell" I mentioned previously from heating the oil of lime, and also an acid smell, which might have been just more acetic acid.

The end of the baling wire was covered with borate glass from previously, which liquefied and smoothed out in the flame. It was still warm enough to sizzle when I added the oil of lime; thereafter it bubbled and had a pebbly surface in the flame, as if the glassy calcium borate (or whatever it was!) was only yielding up its water at orange heat. Calcium borate could be pretty interesting: colemanite (1:3 Ca:B mole ratio) is Mohs 4.5, piezoelectric, and fluorescent yellow, and nobleite (1:6 Ca:B) is Mohs 3. Sadly neither Wikipedia nor Mindat bothers to mention whether either of them is water-soluble, though colemanite mixed with ulexite is a popular pottery glaze ingredient known as Gerstley Borate. The US Borax MSDS for it describes it as "sparingly soluble".

02021-09-04

I poured some of the sodium silicate solution into some dry construction sand.

02021-09-06

I hammered flat some 60/40 Radio Shack lead/tin rosin-core solder (barcode 40293-11311, Radio Shack part number I think 64-007 E) and stuck it in bleach (*Ayudín agua lavandina común*, 25 g Cl/ℓ, UPC 7-793253-000400). Also, a piece of aluminum foil. After 30 minutes no tarnishing was visible on either but the aluminum foil has some small bubbles; I suspect it may be forming aluminum chloride. I don't know how much chlorine is left in this bleach; it's been sitting around the house for quite a while.

The can of dissolved copperas has been sitting open since 02021-08-31 and has developed a non-metallic yellow-brown deposit at the bottom, though the liquid is still clear.

I took a cut-off Speed can weighing 6.5 g and added 19.7 g of the copperas solution to it, which appeared perfectly transparent, in the sense of not being cloudy, but yellow. Then I added 7.4 g of 30-volumes hydrogen peroxide, which immediately began to effervesce and turned the solution apparently black and opaque. A little while later it was quite warm, red-brown, and perfectly transparent (in the same sense) and weighed 33.4 g, gradually decreasing to 33.2 g (unless that's an error in my scale). I conclude that I have made ferric sulfate (since none of the iron oxides, hydroxides, or oxyhydroxides are soluble) at a low enough concentration to not precipitate.

The mass loss so far is 400 mg, which is presumably a combination of lost oxygen and evaporated water.

I placed the Speed can out of the way with a loosely-fitting aluminum-foil hat to allow the solution to dry out without too much

dust falling in. NIH says ferric sulfate should appear as “a yellow crystalline solid or a grayish-white powder”. If it takes more than a week, I’ll try putting it in a sealed chamber with some muriate of lime and let them fight over the water in the air; this is probably going to be necessary because NIH also describes it as “deliquescent”, and “decomp in hot water”. A pH-1 solution of the stuff is apparently used in Pakistan as a dental hemostat: “Known as the “classic” hemostatic agent”! It seems to be the right color from all the pictures, too.

It’s been about 8 hours, and the bleach has eaten the aluminum foil I stuck in it. The lead-tin solder is still shiny, but there’s a white deposit on some of it; I suspect this might have resulted from galvanic contact with the aluminum foil. Oops.

The sodium silicate I poured into the sand two days ago soaked through about 20 mm of sand and formed a solid mass, which has not stuck to the polystyrene container it’s in at all. The mass looks a bit wet, and when I squeeze it between my fingers, it yields and fractures, revealing inner surfaces that look even wetter.

In theory I ought to be able to mix a fair bit of muriate of lime with baking soda without getting chalk — until the solution dries out. Chalk’s solubility is supposed to be 0.0013 g/100mℓ, but calcium bicarbonate’s is supposed to be 16.6 g/100mℓ (162.11464 g/mol), more than a thousand times higher. Baking soda’s is supposed to be 9.6 g/100mℓ (84.0066 g/mol), and muriate of lime’s 74.5 g/100mℓ (110.98 g/mol). All these at 20°, which is a little warmer than it is here at the moment.

I guess the saturated solution of calcium bicarb at 20° is 1.02 mol/ℓ. For that we’d need 2.04 mol/ℓ of baking soda, 17.1 g/100 mℓ, which is higher than its solubility (though the potassium or ammonium salts could maybe do it directly), suggesting that the appropriate amount of muriate of lime ought to *solvate* baking soda. We’d also need 1.02 mol/ℓ of muriate of lime, which would be only 11.3 g/100 mℓ, 6.6 times more dilute than the saturated solution.

Maybe to understand why the large excess of calcium in the oil of lime resulted in precipitation last time I need to understand solubility products a lot better. But I think this means that if I dilute the oil of lime 7:1 and then add saturated bicarb to it, it shouldn’t happen.

So I took a cut-off Monster can weighing 7.3–7.6 g (the scale can’t quite decide) which can hold 227 g of water, and I added 12.1 g of oil of lime to it, bringing the total to 19.7–19.8 g; this should contain about 5.17 g of muriate of lime and 6.93 g of water. Then I added tap water to bring the total to 103.3 g, thus 83.5 g of tap water. So in theory I have 5.17 g of muriate of lime, .0466 mol, dissolved in 90.43 g of tap water, thus about .515 mol/ℓ. I stirred it with a soda straw; a little dust settled to the center of the bottom. So now even a totally saturated baking-soda solution shouldn’t cause effervescence and chalk deposition.

.0466 mol of baking soda should be 3.91 g, which in theory would be contained in 44.6 g of the saturated solution (44.7 g water). So I prepared a saturated solution of baking soda. To the now 103.1 g can (?!) I added 48.4 g of the solution (oops!) and… it turned cloudy with what looked like chalk, but this time completely without

effervescence. That was neither the outcome I hoped for (a clear solution) nor the outcome I feared (effervescence and chalk).

So that should have been 135.1 g of water, 5.17 g of muriate of lime, and 3.91 g of baking soda. Oh, oops, that was half the amount of baking soda I was supposed to use...

And, after a minute or two, it started to effervesce. Ugh. I guess it just hadn't reached the solubility limit of choke-damp, and that *was* chalk after all. It continued to effervesce for many minutes.

I guess I should actually measure the ingredients and the temperature when I repeat this.

After a while the chalk had largely settled out, except for the abundant dust forming aggregates on the surface of the water; effervescence continued at a lower rate. To get a better look I dripped a drop of diluted dish detergent into the can, which immediately scattered the surface dust to the edges of the container. The water beneath was still pretty cloudy, but I can see down to where the chalk has settled on the bottom.

Since I now realize I added half as much baking soda as I'd intended, I added some more of the saturated baking soda solution. The effervescence increased dramatically again, suggesting that the baking soda was indeed the limiting reagent.

It's been a few hours, and a yellow-brown powder has precipitated in the "ferric sulfate", so maybe that's not what it was, because it certainly hasn't evaporated that much. Or maybe there was an excess of hydrogen peroxide and now it's producing insoluble iron oxide or oxyhydroxide from the soluble ferric sulfate in solution.

02021-09-07

The chalk has settled out, leaving clear water with a thin layer of chalk at the surface; maybe the chalk adsorbed the dish detergent. I took a 5.9-gram cut-off Speed can and added 10.0 g of the clear liquid to it with a syringe which I think had previously been used to fill inkwells. No India ink contamination was visible. Then I heated the can on the stove.

The idea here is that if the chalk precipitated, whatever is left over in the solution must be in equilibrium with the chalk. Maybe it's calcium bicarbonate (and salt), just at a lower concentration than I had thought would be soluble. If so, I should be able to boil it down to chalk (contaminated with salt), which should be recognizable by virtue of being a white precipitate that bubbles and dissolves in vinegar and does not dissolve in hot water. And hopefully there will be enough solids dissolved in 10 grams of the solution that I can weigh it.

Adding a few more drops of dish detergent to the can where the chalk was formed does not disperse the surface chalk, so I guess the chalk didn't adsorb the detergent, or the surface would have dramatically cleared again as before.

After heating the chalk (? etc.) deposit in the can for a few hours (mostly gently, because it was popping when I tried heating it intensely) the can weighs 4.8 grams. This suggests that the remaining solid deposit weighs -1.1 grams, which is obviously wrong; maybe I

had something like 1.1 grams of water on it previously when I first weighed it. Then a flying plastic fragment bounced off the ceiling and knocked it onto the floor, scattering an unknown amount of the precipitate, so I guess I need to start over.

Both “nail enamel diluyent” and “enamel remover” are capable of softening the pressure-sensitive adhesive that held the front polarizing film onto this discarded laptop screen, allowing me to scrape it into giant boogers with my fingernails, but neither actually dissolves it. Surprisingly, one or the other of them did attack the polarizing film itself, damaging it (though it still seems to polarize fine). 96% ethanol works dramatically better, softening the adhesive much more quickly, allowing me to rub it into eraser-crumb strings that peel off the plastic cleanly, and apparently not dissolving the plastic itself. Unfortunately, I ran out. Fortunately, some alcohol-based hand sanitizer gel was enough to finish the job. The plastic is still sort of foggy.

I diluted and tasted the citric acid from the health food store. It dissolves to perfect transparency in water and tastes like citric acid.

02021-09-08

The waterglass-cemented sand from 02021-09-04 is now rock-hard and survives being dropped on the ceramic floor without breaking.

02021-09-09

Although the waterglass-cemented sand from 02021-09-04 is rock-hard, I can still break it by hand by flexion. Heating it to orange-yellow with the brass torch for a few minutes turns it from brown to gray but doesn’t expand or soften it visibly, though it did settle a bit in the vermiculite bed, I suspect because the vermiculite had some waterglass or phosphoric acid sticking some of the grains together. However, it is now easier to break by hand, and seems to have some internal porosity (it’s gray all the way through), so maybe it did soften during heating; no such porosity was apparent upon breaking before heating. It gives the impression of poorly cured portland-cement mortar: the outer surface where the flame impinged remains intact when I rub my finger across it, but the inner porous mass instead releases some sand. It’s a bit more sparkly than portland-cement mortars usually are, though.

People have reported that mixing rust powder with waterglass affords a less intumescent refractory mix.

I have some litharge and glycerin here. Bain McKinnon’s 01933 dissertation at Oregon State reports (citing Harry A. Neville’s “Adsorption and Reaction II: the Setting of Litharge-Glycerine Cement”, J. Physical Chem., vol. 30, p. 1181, in 01926) that a 3:2 molar ratio of litharge (223.2 g/mol, 9.53 g/cc) to glycerin (92.094 g/mol, 1.261 g/cc) produces the highest temperature rise, exceeding 80° at times; but he finds that higher amounts of glycerin produce stronger results. I guess that means 670 g to 184 g glycerin, or about 3.6 g of litharge per gram of glycerin. Apparently tin can substitute for lead, and ethylene glycol for glycerin, and with heating to 110° you can get metal glycerolates of cobalt, zinc, manganese, and iron, from acetates, carbonates, oxalates, oxides, or hydroxides.

I heated a drop of the glycerin on the stove on a sheet of aluminum foil. At first it emitted a white smoke with a smell resembling burning sugar; this could be ignited, at which point it burned with a blurry yellow flame and emitted no further visible smoke. Small bubbles bubbled out of the glycerin as it burned, and at one point it popped and threw smoking drops of glycerin around. After a couple of minutes it finished burning and left no visible residue, but the aluminum foil had mostly melted.

To a 6.3-gram plastic cup (from Tregar yogurt) I added 7.3 g of buff-colored hardware-store litharge (Indalo *litargirio*, UPC 7-798123-981544) and 2.1 g of glycerin (Indalo, UPC 7-798123-981483), which I mixed for a couple of minutes with a q-tip to a stiff putty, which nevertheless behaves like a (perhaps non-Newtonian) viscous fluid. This should approximate the 3:2 molar ratio mentioned above: 33 millimoles of litharge to 23 millimoles of glycerin. No heating is evident.

I'd tried mixing up this litharge and glycerin before, but the solid mass initially formed fell apart into a powder after a day or two. There aren't a whole lot of things the ingredients could be other than litharge and glycerin, though; not many things are dense enough to be counterfeit litharge, though plenty are cheap enough.

The polarizing filter obtained from the front of the laptop screen is evidently a linear polarizer with the direction of polarization being the vertical dimension of the screen (the short dimension of the film). My own laptop's screen (a Lenovo Thinkpad) seems to be linearly polarized at about a 45° angle, and my cellphone screen horizontally.

02021-09-10

The litharge–glycerin cement has set up solid, but it's not very strong; poking it with a q-tip bent the bottom of the yogurt cup enough to crack it. It's still shiny as if it were wet; hopefully this means it is nonporous (as it should be) and it will remain solid this time and not crumble. Mina reports that it has a strong smell of paint.

The aluminum foil in bleach left a cloud of small dark particles but otherwise is gone. The solder in bleach has corroded significantly: brown-black, like iron rust, where it was submerged, and a fairly voluminous white that looks like fungus above the water line. If I had to guess, I'd guess the white was stannous oxyhydroxide, stannic oxide, or plumbous chloride, probably not the last since it only formed out of the water. WP says stannous oxyhydroxide is "easily oxidized to stannic oxide by air"; the chlorides of tin are white too, but they're highly water-soluble. No idea what the brown-black corrosion is.

In the evening, the litharge–glycerin cement is less shiny, with more of a matte look.

I put some litharge, without glycerin, in the bottom of the deodorized fish can and heated it on the stove. No change was visible, but the can started to smell of burning plastic. In order to be able to discard it safely, I added a layer of diammonium phosphate granules on top and continued heating; this produced a noticeable aroma of ammonia and a slight sizzling sound. An aspirator for gas scrubbing

would be really helpful for this kind of thing.

After heating it gently for half an hour or so, there was no visible change, but plenty of ammonia, so I decided to resort to more aggressive measures. I put some diammonium phosphate in a cut-off Speed can and heated it with the brass torch, which melted it and released a lot of ammonia vapors. Worse, though, it released a lot of burnt plastic fumes from the Speed can paint. The mass of bubbling phosphoric acid with phosphates of ammonia dissolved in it was black wherever the torch did not heat it to orange heat. After several minutes and a visible haze of white smoke in the living room, I gave up on this approach as well.

After cooling, the Speed can had a noticeable smell of acid, with a mass of what appeared to be carbon foam on top of a whiter porous mass of, presumably, a mix of ammonium phosphates and phosphoric acid. I rinsed it into the fish can with the litharge and fertilizer, moistening the fertilizer (some of which had stuck to the litharge) and added baking soda to the remainder, which fizzed enthusiastically.

The silicone flower Mina made on 02021-08-21 has faded from its original lilac color to a pale cyan. I think the magenta food coloring in its surface has faded from exposure to light; Mina points out that on the bottom, where it's been exposed to air but not light, it's still lilac.

02021-09-11

The litharge/phosphate mix is kind of syrupy with white chunks and smells like a rusty engine. The phosphate/baking soda mix smells acid, and on adding more water it resumes bubbling; maybe I didn't add enough baking soda to fully neutralize the acid, which I guess is less surprising considering that it's potentially triprotic. The litharge–glycerin cement still seems to be hard and is now even less shiny, like dried "satin" paint.

02021-09-14

The litharge–glycerin cement has mostly remained solid, rather than crumbling under its own power, but can now be crumbled like dried mud under the pressure of a q-tip, which is similar to its previous uninspiring performance and significantly less impressive than I was hoping for. Where it was exposed to light (with or without air) it has darkened from its buff color to a darker, grayer color. It pulls away from the plastic yogurt cup (recycling triangle 6, polystyrene) with only the lightest adhesion. Its "satin" paint luster remains unchanged.

I bit through one of the Oogoo samples I'd previously bitten on 02021-08-21. If there is an acetic-acid smell, it is faint enough that I can't be sure it's present. It's about 18mm thick in one dimension, so probably the points nearest the surface were something like 7mm away from the surface.

I rubbed the permanent marker spot off the surface of another of the Oogoo samples, this one about 28mm in diameter, and cut it in half with a razor knife; it too has no discernible acetic-acid smell. I'm not sure exactly when I made this one.

02021-09-15

The litharge-glycerin cement I unstuck from the yogurt cup yesterday has largely lost the bisque or buff color it previously had underneath, presumably due to being indirectly exposed to the sunlight through the window for a single day. I wonder if you could use this property to make photographs? It would be more useful if you could stabilize it, but possibly you can; plausibly the photoreaction product is something like lead dioxide or metallic lead, either of which would be much less easily dissolved by acids than litharge itself.

I added some more water and baking soda either yesterday or the day before to the ammonia-scented Speed can containing the product of decomposition of the fertilizer. Later, I added some hardware-store phosphoric acid (*Desoxidant fosfatizante TF3*, UPC 0-723540-593022, water-clear with unspecified additives). This induced effervescence, showing that the baking soda hadn't been entirely consumed.

02021-10-15

The various wet substances in open containers have dried up. I have thrown out some things whose identity I could no longer remember. The egg white with green vitriol is quite orange.

I tried heating up a flake of whitewash taken from some nearby political graffiti with the brass torch. I was able to heat it to orange-hot, glowing visibly even in direct sunlight. A smell of burning paint (like linseed oil) ensued, suggesting that it was contaminated with non-whitewash paint. The hottest part gradually crumbled away over several minutes.

02021-12-05

A laptop floppy disk casing seemed suspiciously light, leading me to suspect it of being magnesium. But filings cut from it with a hacksaw do not ignite with the brass torch and do not react with vinegar, so it's probably just aluminum. (At first the vinegar did not wet the filings; some dish detergent solved that problem.)

I have had some coupons of corrugated cardboard from a cardboard box floating in (originally) saturated aqueous muriate of lime for weeks now. The one I haven't turned over still isn't wet on top. I added another one to a bowl of just plain water a few days ago; it also isn't wet on top. Normally I would have expected capillary action to soak them through in a few minutes, so I guess they're treated with something to make them slightly hydrophobic. I was hoping that the calcium chloride would convert the sodium silicate adhesive into water-insoluble larnite, but in fact the layers of cardboard peeled apart in warm water just as easily as cardboard normally does, and a little rubbing got it thoroughly wet. I'm not sure if the *paper* is delaminating or if the glue is, but either way the result isn't a water-stable cardboard as I was hoping.

02021-12-13

No visible reaction at first between the hardware-store phosphoric

acid and aluminum foil in a plastic yogurt cup, but after an hour or two slow bubbling was detectable. After some 10 hours most of the foil had dissolved, leaving a slightly gray transparent liquid with some scraps of foil floating on the top of it, so I added more foil. I was expecting the foil to react, but not to dissolve — although both aluminum and phosphate are trivalent, I think of aluminum phosphate as a single material, one that's extremely water-insoluble. But in fact there also exists aluminum dihydrogenphosphate (aluminum phosphate monobasic, 磷酸二氢铝), which might be soluble; American Elements claims it is, and Alfa says they sell it dissolved 50/50 in water, so it might be difficult to redissolve after it precipitates, or perhaps disproportionate.

Also, I forgot to note this earlier, but the acid had no visible reaction with oil of lime, presumably because the pKa of muriatic acid is about −3.0 and the first pKa of phosphoric is +2.14. I suppose that if I partly neutralize it first with baking soda I should be able to precipitate various phosphates of calcium. Interestingly, https://en.wikipedia.org/wiki/Disodium_phosphate mentions a reaction going the other way as well, preparing disodium phosphate from "dicalcium phosphate" ($CaHPO_4$) and sodium bisulfate ($NaHSO_4$), which is sold in hardware stores to lower the pH of your swimming pool.

Sodium bisulfate itself is a GRAS intermediate byproduct of the Mannheim muriatic acid process, "an exothermic reaction that occurs at room temperature"; on heating it transforms to anhydrous (58°), pyrosulfate (280°), and finally sodium sulfate and sulfur trioxide (460°). It's a dry powder at room temperature that aggressively consumes azania. Pyrosulfuric acid is interesting as a legal, stronger alternative to sulfuric acid. The bisulfate ion is also available from thermal decomposition of azanium sulfate (250°), everybody's favorite GRAS thermally stable azanium compound — sold for US$1/kg as a fertilizer, and made from gypsum and hartshorn by throwing down chalk.

(Sodium *thio*sulfate, aka hyposulfite, $Na_2S_2O_3$, is a different material: it's a photographic fixer, reducing agent, and antidote to cyanide poisoning together with sodium nitrite.)

(Sodium azanium sulfate dihydrate is "a well known ferroelectric" and presumably the reaction product with azania. Presumably if you heat it you can expel the azanium and regenerate the azanium absorbent.)

Related to other useful materials to avoid witch-hunt persecution, $NaMnO_4$ (V-2 oxidant, Condy's Fluid) can reputedly be made from MnO_2, NaClO, and NaOH, with an analogous route available for $Ca(MnO_4)_2$. $MnSO_4$ is readily available as fertilizer (US$5/kg); the nitrate (obtained, for example, by reacting with calcium nitrate) can be decomposed at 400° to yield the oxide, which can be purified by anodic deposition. Dehydration of the lye using the oxide supposedly yields Na_2MnO_4, whose disproportionation is another possible route, the one originally used by Condy, but apparently a pentavalent manganese compound is produced additionally or instead.

Another super fun material would be potassium ferrioxalate; hardware stores sell "salt of lemon" to remove rust stains, and I guess

it works by complexing the insoluble ferric iron(III) ions, which I guess should produce $K_3(Fe(C_2O_4)_3)_2$. It's fluorescent green (literally fluorescent in solution) and decomposed by light or heat (reducing the iron back to ferric), a property that can be used to used to make blueprints as an alternative to ferric ammonium citrate (apparently Michael J. Ware invented this in 01994 and wrote a fascinating book covering this and every other aspect of cyanotyping); the ferric iron then reacted with the potassium ferricyanide to produce insoluble ferric ferrocyanide, Prussian blue.

02021-12-14

Local textile pricing:

- Calico doesn't exist.
- 180-thread-count *percal* at 260 cm width costs AR$7000/m on MercadoLibre; at AR$194/US$ that's US$36/m or US$14/m².
- 1.6-m-wide polyester *polar soft liso* (polar fleece with a nap on it to soften it further) is AR$850/m (US$4.40/m, US$2.70/m²). A roll is reported to be 22 kg and 50 meters, suggesting a weight of 280 g/m².
- 1.45-m-wide *camisero* cloth falsely advertised as linen (fine print: 55% linen, 45% rayon) is AR$700/m (US$3.60/m, US$2.50/m²).
- 1.5 m *fibrana* floral-print rayon cloth is AR$430/m (US$2.20/m, US$1.46/m²).
- 1.5-m-wide 120-gram-per-linear-meter polyester *gasa muselina*, muslin gauze, is AR$370/m (US$1.90/m, US$1.26/m²).
- 160-m-wide raw cotton 20/20 *lienzo* is AR$350/m (US$1.60/m, US$1.12/m²).
- 150-μm-thick 1.4-m-wide flexible clear PVC film is AR$290/m (US$1.49/m, US$1.07/m²).
- 1.40-m-wide nonwoven polyester *friselina*, 45-gram weight (/m²? though an unhappy buyer reports that it's actually 20-gram cloth), goes for AR$2244 per 50-meter-roll (AR$45/m, 23¢/m, 16.5¢/m²) in different brilliant colors.
- A week ago we went to town and found that burlap (*arpillera*) was AR$400/m² there (US$2.10/m²). The other prices above are all MercadoLibre. A MercadoLibre vendor offers 1-m-wide *arpillera yute* at AR$289/m (US$1.49/m, US$1.49/m²).

The *friselina* seems like probably the best material for anode bags and similar filtering: it's damned cheap, probably PET, safer and cheaper than asbestos, perhaps more inert than glass fiber, certainly more inert than any natural fiber, and PET is surpassed in its inertness among the organic polymers only by polyethylene, polypropylene and fluoropolymers, which are more difficult to find in the form of cloth, though coarse polypropylene cloth is available for carpeting and tarps. (Some vendors also offer nonwovens which purport to be both polypropylene and polyester, or blends of the two.) It may also be useful as a lint-free cloth for cleaning.

HEPA air filters are generally meltspun nonwoven polypropylene; this is a possible alternative source for polypropylene fabric for anode bags. Surgical scrubs and lab coats are another possible source.

The reduction of Prussian blue to Prussian white (produced by fading of Prussian blue by light! and reversible by wet air exposure, therefore inexhaustible) needs to steal an electron from somewhere, so

you can use it for photographic patterning of a surface by oxidizing other random things in the environment. I don't think anyone has done this.

02021-12-18

I heard the other day that you can electrodeposit aluminum from aluminum chloride dissolved in molten methylsulfonylmethane, which is easily available as a dietary supplement. (Aluminum chloride by itself won't work at atmospheric pressure; it sublimes at 180°). I can't wait to try it.

02021-12-21

Some time ago I deposited Elmer's Classic Glitter Glue Silver (UPC either 0-026000-185073 or 0-26000-18191, depending on which label you believe; ingredients: deionized water, PVAc emulsion, polyvinyl alcohol, and craft glitter (various sizes and colors); the US marketing label lists no ingredients but apparently Argentine law requires it) on a sheet of LDPE and let it dry. As deposited, the glitter flakes are dispersed almost isotropically, without any preferred orientation, but upon drying (I'm guessing to about a tenth of the original thickness, suggesting about 90% water content by volume) the glitter flakes were mostly parallel to the surface as indicated by their optical properties, ±10° or so. The dried PVA/PVAc mass peeled off the thin polyethylene sheet fairly easily, as you'd expect. (See 3-D printing in poly(vinyl alcohol) (p. 1080).) See Xerogel compacting (p. 1119) for the implications.

To get a slightly more precise read on the water content than "about 90% by volume", I took a polypropylene bottlecap weighing 1.7 g and deposited 3.0 g of the glitter glue in it, for a reported total of 5.3 g (!? fuck this scale) and allowed to dry in the sun all afternoon. At night it weighed 4.3 g and was still noticeably soft, though with a tough glittery skin, so I propped it up in front of the air conditioner condenser fan. I hope that will help it dry faster.

I also have a thin membrane of the dried glue (of mysterious origin), thin enough to be almost entirely transparent, which I sprayed with a saturated aqueous solution of borax. Upon drying, it was crinkly and "harder" than before; previously it was stretchy and "soft", but now it's stiff and brittle. I rinsed it again with water and it immediately became a soft gel, coming apart a little, and now it is drying again.

And I've deposited bits of the glue on some more LDPE, some aluminum foil, a PET bottle, some polystyrene, and some ABS. My expectation is that, when dry, it will have failed to adhere strongly to any of them and will be easily peeled off; "Elmer's" (Newell Office Brands) advertises that it's "ideal for paper, cloth, craft, etc."

From Podsiadlo, 9 other authors, and Kotov's work ("Ultrastrong and Stiff Layered Polymer Nanocomposites") in 02007, I learned that PVA can be strongly crosslinked with glutaraldehyde; yesterday I learned that glutaraldehyde can be easily bought from medical supply stores as a heavy-duty disinfectant (typically at concentrations of 2–2.5%, lower than the concentrations used for crosslinking). It's somewhat hazardous to the humans because it's great at crosslinking

proteins and they are made of proteins. Reportedly it does not reduce the solubility of pure starch because what it does is bond amine groups to hydroxyls, so it can crosslink proteins to starches but not starches to starches, but that can't be the whole story because PVA is nothing but a saturated carbon backbone with hydrogens and hydroxyls, and evidently glutaraldehyde is great at crosslinking it. 10% chitosan apparently is effective at making starch cross-linkable with glutaraldehyde.

02021-12-22

The glitter glue pulled itself free of the LDPE, polystyrene, and aluminum foil while drying. From the ABS and PET it didn't peel off spontaneously, but it easily peeled off by hand. In the case of the ABS this may be because it was spread thinly over a rougher surface, rather than because it had more adhesion. Also, the polystyrene may have a coating on it, and all the materials may be contaminated with skin oil.

02021-12-23

The glitter glue bottlecap brought in from outside after two days of drying in warm fan air weighs 2.6 g. Upon separating from the bottlecap the relatively dry glue weighs 0.9 g and the bottlecap a reassuring 1.7. This suggests that the 3.0 or 3.6 grams of glitter glue I originally deposited was at least 70% water, but unless it's still retaining some water inside, it's not close to 90%.

But honestly it probably is. I should break it up and put it in a desiccator with some muriate of lime.

Topics

- Materials (p. 1138) (59 notes)
- Pricing (p. 1147) (35 notes)
- Experiment report (p. 1162) (14 notes)
- Bootstrapping (p. 1171) (12 notes)
- Aluminum (p. 1180) (10 notes)
- Phosphates (p. 1184) (9 notes)
- Foam (p. 1185) (9 notes)
- Waterglass (p. 1189) (8 notes)
- Argentina (p. 1200) (7 notes)
- Refractory (p. 1225) (5 notes)
- Aluminum foil (p. 1237) (5 notes)
- Vermiculite (p. 1238) (4 notes)
- Poly(vinyl alcohol) (PVA) (p. 1245) (4 notes)
- Heating (p. 1253) (4 notes)
- Glass (p. 1254) (4 notes)
- Kingery, the father of modern ceramics (p. 1288) (3 notes)
- Alabaster (p. 1309) (3 notes)
- Oogoo (p. 1349) (2 notes)
- Pumice
- Borax

Dipropylene glycol

Kragen Javier Sitaker, 02021-08-01 (updated 02021-08-15)
(2 minutes)

I thought this stuff was just propylene glycol, but it isn't. It's relatively cheap (US$3/kg) and so I'm curious about its merits as a coolant. I'm guessing its solvent properties are similar to those of propylene glycol.

Of course it's miscible with water.

Dow says the pour point is -39°, it supercools, its viscosity drops from 75 centipoise at 25° to 10.9 centipoise at 60°, it boils ("distillation range") at 228–236°, its vapor pressure at 25° is 2.1 Pa, its flashpoint is 124°, and its specific heat is 2.18 J/g/K. The technical data sheet touts its "low toxicity". So far so good. Its heat of formation is -628 kJ/mol, but at 134.2 g/mol I'm not sure that means it's a bad fuel. Thermo Fisher gives its flashpoint as 138°, its autoignition temperature as 310°, and its orl-rat LD_{50} as *14850 mg/kg*; even more astonishing, its LC_{50} for freshwater fish is ">5000 mg/ℓ, 24h". So unless they're confusing it with propylene glycol it's about as toxic as water.

Perfumers describe it as "virtually odorless with a slight-ethereal-green solvent note." Monument says it's "far less hygroscopic than other common glycols" and mentions its use as a coolant. Shell says contact with copper, copper alloys, and zinc must be prevented and that it is not hazardous. The National Toxicology Program (paper) agrees, but notes that giving rats drinking water with 40000 ppm (4%) of it killed them with kidney disease within two years, but the 2% group was fine.

Topics

- Materials (p. 1138) (59 notes)
- Pricing (p. 1147) (35 notes)
- Toxicology (p. 1316) (2 notes)

The ayurvedic “fire mud” of Bhudeb Mookerji and modern castable refractories

Kragen Javier Sitaker, 02021-08-05 (updated 02021-08-15) (22 minutes)

Some comments in a book on ayurvedic alchemy led me to think about possible formulas for castable refractory.

The Rasajalanidhi of Bhudeb Mookerji and its “fire mud”

The Rasa-jala-nidhi describes part of the preparation of a nabhi-yantra or jala-yantra as follows (p. 272, 307 of 406, Bhudeb Mookerji, M.A., Vol. 1, published in 01926):

> ...plaster the joint by means of the following paste, which serves as a good waterproof:—oxidised iron, finely powdered,1 molasses, and lime, rubbed together with a highly condensed decoction of the bark of babbula [Acacia nilotica, now Vachellia nilotica, the source of gum arabic]. This plaster is called “water mud”, and water cannot pass through it. Similarly, chalk, salt, and oxidised iron, rubbed together with buffalo’s milk, gives rise to a plaster, called, “fire mud”. This plaster is a strong fire proof, The joint of the basin and the crucible is to be plastered with this fire mud, ...

It doesn’t really say what the proportions of the chalk, salt, oxidized iron, and buffalo’s milk are; WP says buffalo milk is a bit higher in everything (except water, I suppose) than cow’s milk. I suspect this mixture will form hematite cemented with hydroxyapatite and contaminated with salt and carbon when heated — that the milk serves as a source of phosphates, which combine with the calcium from the chalk to form hydroxyapatite, as well as protein and sugar which act as low-temperature binders; then, at higher temperatures, the protein cross-links and forms a thermoset, holding the fire mud in shape until all the organics have been converted to carbon with a little nitrogen. I suspect the salt is just present as a catalyst and the iron oxide as an inert aggregate that is stable at high temperatures.

In other parts of the book (e.g., Vol. 1, p. 41 (76/406), or on p. 10) he does give precise proportions of ingredients by weight.

I do not regard the author of this “translation” as entirely reliable; much of his book is concerned with procedures for transforming base metals into gold, silver, and copper, a process he says he has had demonstrated to him by his preceptor and is confident he could carry out himself given the appropriate equipment; moreover he claims that the human race is 1.6 billion years old. The book he “translates” from Sanskrit says it is written by one Bhudeba of the Mukhopadhyaya family, descendant of the sage Bharadwaja who, nine hundred thousand years ago, brought to India the science of medicine; his parents are Harilala Deba and Nistarini Debi; and on the next page he explains that Mukhopadhyaya is “generally written in English for the sake of brevity” as “Mookerji”.

That is, the book is written in Sanskrit by Bhudeba Mookerji and translated into English by Bhudeb Mookerji. He says the book is "based on many of those which are still extant", but is written in "year 5026 of the perverted Kalijuga". The Kali Yuga started on February 17/18 in 03102 BCE, so year 5026 is most of 01925 CE and a little of 01926. So unfortunately it is impossible to know whether this "fire mud" recipe dates from 01926 or a thousand years earlier, or all the way back to Nagarjuna 1700 years earlier, to whom rasāyana (रसायन) and rasaśāstra are traditionally attributed. (On p. 20 he claims that a particular prescription, one tola of purified sulfur mixed with butter, "was prescribed more than 5000 years ago. Present-day human beings can stand only one fourth to half of the dose prescribed in those days." units(1) and Wikipedia agree that a tola is an 80th of a seer, (though Dr. Mookerji says it's a 64th of a seer) and a seer is 14400 grains by the British Colonial Standard, so a tola is some 11.7 g if the standards haven't shifted too much in 5000 years; elemental sulfur's oral LD_{50} is unknown but is over 5000 mg/kg, so if Dr. Mookerji was getting toxic symptoms from feeding a "beautiful lady" 11.7 g of sulfur (<300 mg/kg) every day for three weeks, maybe his sulfur was contaminated, probably with arsenic.)

On p. 301 (345/406) Dr. Mookerji gives the chain of weights java (barley seed)x6 = gunja; ×3 = balla; ×2 = masha; ×2 = dharana; ×2 = niska; ×2 = kola; ×2 = tola. This would make the tola 1152 barley seeds. Normally a barley seed is about 40 mg, which would make the tola some 46 g rather than 12 g, but perhaps Indian barley was smaller a thousand years ago. He also says a java weighs 6 mustard seeds (sarshapas), and mustard seeds are typically about 3–6 mg, which would put a java at 18–36 mg, unless modern mustard seeds are similarly enlarged.

Still, though, I think Dr. Mookerji's explanation of how to make a phallus out of mercury or "rasalingam" (Vol. 1, p. 10, 45/406) by dissolving one third of its weight of gold leaf into it, then rubbing it with vegetable sour juice, putting it inside a lemon, and boiling it in gruel, is much more convincing than Sadhguru Jaggi Vasudev's explanation that consecration or energization with a divine reverberation is what solidifies the 99.2%-pure mercury or 99.8%-pure mercury in the mercury lingas in the temple he consecrated called Dhyanalinga, in the subterranean water tank called Theerthakund.

He explains that the term "pārada" comes from "para", "end", and "da", "give", both of which words amusingly enough have the same meaning in Spanish, and in the case of "da" is in fact the same word. Spanish "parar" "stop" comes from Latin "parāre" "prepare" or "make ready", which comes from Proto-Indo-European "*pere-" "produce, procure", which I think very likely isn't the same as Sanskrit "end".

Getting back to the refractory recipe, Sigma says tribasic calcium phosphate melts at 1100°, which doesn't count as "a strong fire proof" in my book, but it's already hot enough for a pretty wide range of uses; and the carbon would tend to support it up to much higher temperatures. WP says tribasic calcium phosphate $Ca_3(PO_4)_2$ melts at 1670° so maybe Sigma was talking about hydroxyapatite $Ca_5(PO_4)_3OH$; WP says, "Most commercial samples of 'tricalcium

phosphate' are in fact hydroxyapatite."

I think this phosphate-formation reaction probably has the stoichiometry to actually work. Human milk has about a 1.4:1 to 1.7:1 Ca:P ratio, while cow's milk is 1.24:1, while TCP is (by moles) 1.5:1 and hydroxyapatite is 1.67:1, so if you wanted to make calcium phosphate cement out of cow's milk, you'd have to add about 0.25–0.5 moles of calcium for each mole of phosphate in the milk. Buffalo's milk is reported to have higher calcium, except in Argentina :

> Buffalo milk is characterized by high calcium content (about 1.5-fold Ca than in cow's milk) as was apparent from several studies, except in BM from Argentina (Patino et al. 2007) which had Ca content comparable to that of CM.

And buffalo-milk phosphate content seems to be similar to that of cow's milk. So it's possible that buffalo milk wouldn't be calcium-deficient in the same way — but I really have no idea whether we're talking about Indian buffalo from 01926, 01826, 01626, 01226, 00426, or what, so no way to even guess at the calcium content of their milk.

Alternative routes to heat-setting castable refractory mortars

Even if cow's milk doesn't work, though, different sources of phosphate and calcium (or possibly aluminum or boron) would surely work to form calcium phosphate (or aluminum phosphate or boron phosphate) at high temperatures.

The 01950 MIT dissertation of Wm. David Kingery ("the father of modern ceramics"), "Phosphate Bonding in Refractories" seems very much worth reading here. He outlines a wide variety of recipes then in use or disclosed in patents, and in particular mentions:

> Aluminum, chromium, magnesium, and zirconium oxides react chemically with phosphoric acid at 200°C to form a bonded material. The metal phosphate reaction products are refractory and stable. Rather than using the oxides, the halides of magnesium, tin, thorium, calcium, barium, aluminum, zirconium, or titanium may be used with phosphoric acid to form a bonded refractory. After mixing the constituents to a pasty constituency, the plastic mass is formed and heated to approximately 1000°C to effect the bonding reactions and form the final product. Using aluminous refractory materials, phosphoric acid reacts to form a film of aluminum phosphate around each particle, which acts as a bond. ...
>
> Aluminum hydrate [i.e., hydroxide] may be used with refractory clay, filler and phosphoric acid to form a bond which becomes permanent when heated to 100–300°C. The addition of aluminum hydrate to refractory compositions of zircon, silicon, etc., and phosphoric acid is also advantageous. Use of this material allows final heat hardening at temperatures of about 600°F [i.e., 315°] rather than the 1200°F [i.e., 650°] which must otherwise be applied. ...
>
> With aluminous materials, alkaline earth acid phosphates or ammonium acid phosphates may be used in place of phosphoric acid. On heating to 200–300°C bonding action is obtained...
>
> Acid phosphates may also be formed by addition of triphosphate [i.e., tribasic phosphate] and an acid which readily reacts with it forming mono- or bi-phosphates [i.e., hydrogen or dihydrogen phosphates]. This process may be used with alkaline earth phosphates, preferably calcium which is less expensive than other materials. ...

And that's exactly what I speculated above is going on with the buffalo milk: it contains some amount of calcium phosphates which are not yet tribasic. Also, though, this sort of points out that the acids

formed by burning the protein in the milk might help to acidify the phosphates further, perhaps replacing some of its hydroxyls with hydrogens. He also mentions a very exciting possibility of refractory inorganic elastomers:

> A study of the use of metaphosphates in refractory mortars by Herold and Bust indicated that *rubber-like metaphosphate polymers* form a clay-gel cement with clay. However, a large amount of the phosphate was required to form an adequately plastic mass.

He goes on to mention some other possibilities:

> In all, the oxides and/or hydrates of thirty-four cations were tested. ...Oxides of a highly basic nature react so violently with phosphoric acid that a porous friable structure results (MgO, ZnO, CaO, La_2O_3, BaO, SrO). ...
>
> A large number of weakly basic and amphoteric oxides did react with phosphoric acid to produce a bonded cement-like product. Included in this group are BeO, CuO, Cu_2O, CdO, Fe_2O_3, Fe_3O_4, SnO, Pb_3O_4, $Al(OH)_3$, $Ti(OH)_4$, $Zn(OH)_4$, ThO_2 and V_2O_5 in addition to calcined ZnO, MgO, La_2O_3 and CaO [with partially neutralized acid].

This suggests that the iron oxide of Dr. Mookerji's recipe might *not* be an inert filler as I thought.

On p. 17 (23/94) of his dissertation he has a chart of "modulus of rupture" (flexural strength, the same as the tensile strength for a homogeneous material) of an alumina mix with 10% kaolin bonded with various amounts of various phosphate cement mixes; aluminum is best at some 1300 psi (9 MPa), followed by beryllium (1200 psi, 8 MPa), magnesium (almost the same), and, surprisingly, iron (1100 psi, 8 MPa). Just bonding the mix with phosphoric acid alone yielded 750 psi, same as with barium, while calcium and thorium did substantially worse (as he puts it, "Calcium, barium, and thorium additions to phosphoric acid decrease its effectiveness [as a cement for alumina and kaolin]."). Moreover, all the good performers showed strength continuing to grow linearly with the cement fraction — he only tried up to 7% or 8% cement by weight, and while the weaker bonds like Ca and Ba were leveling off at that point, the stronger bonds weren't, suggesting this was only a small fraction of the strength they could potentially develop.

Later on (p. 33, Fig. 9) he measures a strength of 2500 psi (17 MPa) with 13% monoaluminum phosphate ($Al(H_2PO_4)_3$) cement. He makes the monoaluminum phosphate sound super tempting: it gradually increases in viscosity as the water content decreases, and then you can fire it to a berlinite gel, much like waterglass. But these solutions "precipitate at pH values greater than about 2.8" (p. 34), so maybe you can do the same kind of instant hardening trick you can do with waterglass, only with bases rather than acids.

Another interesting thing he points out, when he tried mixing up some different phosphate mortars, is that at 1500° the magnesium phosphate mortar is stronger than the aluminum phosphate mortar, because the aluminum phosphate mortar hasn't started fusing yet!

Ultimately, he says, the monoaluminum phosphate ends up as aluminum metaphosphate, $Al(PO_3)_3$ (p. 25), which is different from the $AlPO_4$ orthophosphate of which berlinite consists. But above 830°, in the absence of oxygen, as in a gasworks, the metaphosphate gradually becomes the orthophosphate! And above 1220°, again in the absence of oxygen, even that berlinite gradually calcines to sapphire.

This all happens in a few hours when it's in a very thin layer.

Oh, fantastic! His Appendix G on p. 79 has cost data. US$1.10 per 100 lb. (45 kg) of 40° sodium silicate, US$2 for $MgCl_2$, US$5.50 for 85% H_3PO_4, US$17 for $Al(H_2PO_4)_3$.

Ooh, interesting, "Ceramic properties of kaolinitic clay with monoaluminum phosphate ($Al(H_2PO_4)_3$) addition" at UNLP.

In isolation the phosphates that dissociate sometimes produce phosphoric acid, and sometimes pyrophosphates or polyphosphates. A "metaphosphate" is a polyphosphate chain that's either cyclic (as in the trimetaphosphate and hexametaphosphate of sodium, rings of respectively three and six phosphate radicals) or extremely long.

Phosphates, ordered by the decomposition or melting point of their crystalline structure as a rough guide to how stable it is:

- Magnesium ammonium phosphate, Mohs 1.5–2, mineral struvite
- Diammonium phosphate, decomposes at 155°
- Nickel phosphate, decomposes at 158°
- Monosodium phosphate, decomposes at 169° to disodium pyrophosphate or to sodium trimetaphosphate at 500° over 5 hours
- Ferric phosphate, decomposes at 250°
- Monopotassium phosphate, melts at 252°, decomposes at 400°
- Cupric phosphate, Mohs 4–6, decomposes somewhere over 300°
- Monopotassium phosphate, decomposes to potassium metaphosphate KPO_3 at 400°
- Dipotassium phosphate, decomposes at 465°
- Disodium pyrophosphate, melts over 600°
- Sodium tripolyphosphate, melts at 622°, very water-soluble
- Sodium trimetaphosphate, melts at 627.6° (?), can be synthesized from orthophosphoric acid and table salt at 600° in 2 hours
- Sodium hexametaphosphate, melts at 628° but hydrolyzes spontaneously
- Zinc phosphate, melts at 900°, common corrosion inhibitor, dental cement
- Cobalt phosphate, melts at 1160°
- Trimagnesium (di)phosphate, melts at 1184°
- Tripotassium phosphate, melts at 1380°, strongly basic like TSP, deliquescent
- Boron phosphate, sublimes at 1450°, glassy but crystallizable reaction product of boric acid and phosphoric acid, or at 1000° of diammonium phosphate and borax
- Trisodium phosphate, melts at 1583°, strongly basic and water-soluble
- And of course aluminum phosphate, melts at 1800°

So much for phosphates; what about the cations — boron, calcium, magnesium, and aluminum?

Boron is the simplest, and borates (not to be confused with Borat) have also been used to make ordinary methyl vinyl silicone into a "ceramizable composite". It has the advantage of having a very amphoteric oxide, so you can get your boron as a borate. Indeed, nearly all the non-borate compounds of boron are extremely exotic, highly toxic, very unstable, or all three.

It's worth mentioning that, if I'm remembering correctly, borates

are commonly used to *slow down* the setting of magnesium phosphate cements.

- Borazane, melts at 104°. Just kidding.
- Boric acid, melts at 170.9°, water-soluble; tends to evaporate rapidly over 500°.
- Ammonium pentaborate, decomposes over the range 428.3°–458.4° after several dehydration steps, water-soluble; this also makes the stuff one of the highest-temperature ammonium compounds ([a 01994 paper says
- Cesium dodecaborate. Just kidding.
- Borax, melts at 743° when anhydrous, but the more normal decahydrate melts at 75°; water-soluble.
- Zinc borate, melts at 980°.
- Calcium borates in the 986°–1479° range; see below.

Calcium of course has many compounds, but most of them are dismayingly stable for this sort of purpose, like larnite (melts at 2130°) and quicklime (melts at 2613°), though Kingery's experiments quoted above suggest that quicklime reacts with phosphoric acid with great violence even at room temperature. Less heat-stable compounds include:

- Chloride of lime, which melts at 100° and decomposes at 175°, is a strong oxidizer, is highly water-soluble, and gradually decomposes to quicklime in air.
- Oxalate of lime, which decomposes at 200°.
- Calcium formate, which decomposes at 300° to chalk and formaldehyde.
- Calcium nitrate, which decomposes starting at 500°.
- Chalk, which of course calcines (thus the name) to quicklime at temperatures over 550°, and rapidly over 898°.
- Slaked lime, which calcines to quicklime at 580°.
- Calcium sulfite, melts at 600°, oxidizes in air to alabaster; used in food as a sparingly soluble oxygen-scavenger antioxidant.
- Muriate of lime, melts at 775°, deliquescent at room temperature.
- Calcium tetraborate, melts at 986°, CaB_4O_7, tends to be glassy, but can be devitrified in a few minutes — but there are several calcium borates, many of which can be understood as continuously varying mixtures of boria B_2O_3 and quicklime CaO, with melting points varying from 986° to 1479° (or 2613° if you count quicklime itself).
- Calcium metaborate, melts at 1154° apparently, $Ca(BO_2)_2$

How about magnesium? Magnesium phosphate is an awesome refractory cement. As with calcium, [the oxide] is absurdly stable, melting at 2852°, though it's quite soft. But there are some convenient magnesium compounds that could provide magnesium to react with phosphate if heated:

- Magnesium nitrate melts at 129° and decomposes at 330°. It's very water-soluble.
- Magnesium formate decomposes to magnesium carbonate around 200° and magnesia around 400°.
- Magnesium carbonate decomposes to magnesia around 350°–900°.
- Magnesium oxalate decomposes to magnesia around 420°–620°.

• Magnesium chloride melts at 714°. It's very water-soluble.
• Magnesium sulfite melts at 1124° (?).
• There exist magnesium borates too, including "admontite", "mcallisterite", and "boracite", with uses including microwave dielectrics (?), and apparently they melt over 1000°. Often these are produced by reacting magnesia with boric acid at low temperatures. However, it is remarkably hard to find good information about their physical properties, such as their melting point, hardness, and strength.

If anything, most aluminum compounds are even more obnoxiously stable than those of calcium and magnesium, but there are exceptions. The obvious candidate sources for aluminum cations are aluminum trihydroxide and metallic aluminum; also, the highly water-soluble (and violently acidic) aluminum chloride sublimes at 180°.

(Aluminum hydroxide flour might also be an interesting material for an Oogoo made with hardware-store silicone, as an alternative to cornstarch; it might contribute strength, but also possibly contribute hydroxyls to speed the uniform setting of the silicone, and then, after setting, might make it possible to calcine the silicone into a silicoaluminate or carborundum-alumina composite.)

Topics

• Materials (p. 1138) (59 notes)
• Contrivances (p. 1143) (45 notes)
• Pricing (p. 1147) (35 notes)
• History (p. 1153) (24 notes)
• Filled systems (p. 1161) (16 notes)
• Strength of materials (p. 1164) (13 notes)
• Clay (p. 1179) (10 notes)
• Phosphates (p. 1184) (9 notes)
• Ceramic (p. 1193) (8 notes)
• Refractory (p. 1225) (5 notes)
• Weighing (p. 1267) (3 notes)
• Kingery, the father of modern ceramics (p. 1288) (3 notes)
• Units
• Ayurveda

Arc maker

Kragen Javier Sitaker, 02021-08-07 (updated 02021-12-30)
(11 minutes)

Stick welding is a pain partly because of the necessity to strike the arc and the danger of sticking the electrode; with a traditional buzzbox it takes some skill to learn to avoid these. And an arc furnace is clearly the easiest way to reach temperatures over 1000° or so.

Basic circuit parameters

In theory, though, an intelligent control circuit could make this a lot easier. A high-frequency start mode, as found on some TIG welders, could activate when open circuit is detected, initiating a plasma; and when you stick the electrode and a short circuit is detected, the high-power circuit can be turned off entirely until a low current detects that the short is cleared. Perhaps by monitoring the arc voltage it could even warn you you're getting too close.

I think about the lowest an arc can go usefully is about 50 volts and 10 milliamps, but for welding as such you need much higher currents: 10 amps at least, 100 more typically. But the initial high-frequency start, according to conventional wisdom anyway, requires something on the order of a kilovolt per millimeter, so, say, 3kV at 100kHz, but then probably only up to a few microamps.

The normal welding current can run at a much lower frequency than the high-frequency start; 100–120 Hz is traditional, but even 500 Hz would be reasonable.

A table of values from Deringer-Ney cites minimal voltage and current conditions for maintaining an arc, however short, for different electrode materials; silver is said to require 400–900 mA and 11–12.5 volts, while carbon only 10–30 mA, but 15–22 volts. Sadly, no values are given for the materials I'm most interested in, like copper, steel, brass, and tungsten carbide. Still, these values are much lower than I had expected!

Can you just use a single flyback? No!

In theory a flyback converter can smoothly switch between any output voltage and frequency, but in practice I feel like this factor of 60 in voltage and 200 in frequency is probably pushing it pretty far. Like, suppose your flyback transformer has a turns ratio of 80:1, so one volt on the input produces 80 volts on the output. So in theory you can get 50 volts output with 5-volt input pulses at a duty cycle of 1:8 or 1/9, so the pulses are on for 11% of each 2-ms cycle (thus 220 μs) and off for the other 89%. And to get 3000 V out, the pulses use a duty cycle of 8:1 or 8/9, giving 3200 V. But now we're talking about 8.9 μs on and 1.1 μs off.

So what's the magnetizing inductance of our flyback core? 10 amps on the output is 800 A (oof! car jumper cables!) on the input, but that's divided by our duty cycle to get 7200 A, which is the average value our current must reach in its 220 μs, so a current slew rate of 7200 A / 220 μs = 32 MA/s, under an influence of only 5 V. So the

magnetizing inductance must be 153 nH, and our switching MOSFET bank needs to handle ten thousand amps. ½LI² at 14400 A is 15.9 J, which is going to be a pretty huge inductor.

Is that inductance a minimum or a maximum? If the magnetizing inductance is higher, the current will rise lower, and less energy will be stored in the core, thus producing lower output and limiting our welding power to a too-low value. If the inductance is lower, then instead we will produce more output power, if we can, unless we saturate the core. (Hmm, what does "if we can" look like?)

Then what happens with the high-frequency case? With 153 nH, in 8.9μs our 5 V can "only" raise the primary current from 0 to 291 A, which limits out output high-frequency start current to 3.6 A, which is grossly overkill.

Using two supplies should work

It would surely be better to use two separate flyback converters to produce the high-voltage, high-frequency, low-current starter signal and the low-voltage, low-frequency, high-current arc sustaining supply. They can be usefully separated by passive means to take different circuit paths. An inductor reaching 100 ohms (=ωL) at 100kHz would be 160 μH, and a capacitor reaching 100 ohms (=1/(ωC)) at 500 Hz would be 3.2 μF, so a "bias tee" that routed low-frequency "DC" stuff to one flyback and high-frequency stuff to the other would be very easy to build. 160 μH at 500 Hz is only 0.503 ohms, and 3.2 μF at 100 kHz is similarly 0.497 ohms.

The low-voltage, high-current system probably ought to be powered from mains power rather than from a 5-volt supply, using a voltage step*down* flyback transformer. 50 volts at 10 amps is necessarily 500 watts, and delivering 500 watts from a 5-volt power supply is just gonna suck. Delivering 500 watts from a 240-VAC power supply is a little tricky but highly doable. Stepping it down to 50 volts or less could be done simply with a stepdown transformer, but 50Hz transformers are heavy; I think an H-bridge from the mains wires across the primary of a high-frequency transformer would be a more elegant approach that would easily permit the kinds of power and polarity control I'm talking about here.

This can be done without the kinds of large energy-storage capacitors needed by conventional switching power supplies, because it doesn't need to produce a constant ripple-free voltage; the H-bridge can also operate at frequencies of around 100 kHz, with a fuse and a little bit of filtering on the inputs and outputs, at just the cost of having an annoying 100-Hz buzz in the arc. A 500-watt power supply that has to store 10 milliseconds' worth of energy needs 5 joules of energy storage; one that only has to store 10 microseconds' worth needs 5 millijoules.

The high-voltage high-frequency starter supply could use a stepup flyback as usual, or maybe a chain of two 16:1 stepups. The flyback part of it would operate at a 7:3 duty cycle: 7 μs on at 5 V, 3 μs off at 0 V, which would produce 11.7 volts referred to the input, or 187 V on the output. The second stage 16:1 stepup transformer would then raise this to 2995 V. If this output is to be capable of delivering 10 mA, which may be rather a lot but is surely adequate, the first input

winding needs to handle 2.56 A average, 3.66 A average during the 7 us, 7.3 A peak, which is an eminently straightforward thing to achieve. Magnetizing inductance can't be more than 5 V 7 μs / 7.3 A = 4.79 μH. $\frac{1}{2}LI^2$ is 0.13 μJ.

Microcontrollers

This seems like a waveform that would be easy to generate with an Arduino or an ATTiny9. 3 μs is 48 cycles or 24–48 instructions at 16MHz even if you're doing it in software; the ATTiny9 runs at 10MHz and has a 16-bit PWM counter. (Also the 61¢ ATTiny4: and the 40¢ ATTiny5: 12MHz, 256 instructions of Flash, 32 bytes of RAM. The difference is that the ATTiny5 has an ADC and the ATTiny9 has 512 instructions of Flash.) The transformers seem like they'd be easy to wind; 2 or 3 turns on each primary, 32 or 48 turns on the secondary. Putting an ac-coupling cap in between the MCU's GPIO (or rather its external buffer transistor or transistors) and the transformer would keep the transformer from saturating.

Looking at AVRs, there's a new (from 02018) "0-series" 46¢ ATTiny202 with 20MHz, 1024 instructions of Flash, 128 bytes of RAM, and a bunch of onboard peripherals including I^2C ("TWI"), master/slave SPI, two PWM channels, a 10-bit 115-ksps 6-input ADC, 6 GPIOs, and an FPGA-style 3-LUT logic cell (or two?). Also it maps the Flash into the data address space so you don't need the special aptly named LPM instruction to read it.

Winding transformers

This coil inductance calculator suggests that this low inductance should be easily reachable with an air-core coil; for example, 3 turns packed into 2 mm with a radius of 16 mm and a relative permeability of 1 should be about 4.55 μH. But I think this may be inaptly using a long-coil approximation formula $\mu N^2 A/l$ which assumes Nagaoka's coefficient is 1. WP says that for cases like this `L\approx {\frac {\mu _{0}}{2\pi}}N^{2}\pi D\left[\ln \left({\frac {D}{d}}\right)+\left(\ln 8-2\right)\right]+{\sqrt {\frac {\mu _{0}}{2\pi }}}\;{\frac {ND}{d}}{\sqrt {\frac {\mu _{\text{r}}}{2f\sigma }}}` but I have no idea what to make of that. Low-frequency ferrites have permeabilities in the 350–20000 range so any ferrite would instantly rocket us out of the microhenry range. But without a core, how can we guide the flux through 48 windings of a secondary?

10 amps in a transformer winding or something like that requires at least 12-gauge wire, 2.1 mm in diameter, even though 16-gauge (1.3 mm) would be fine for single conductors.

Switching elements

Our peak wall voltages here are 340 volts. If we're H-bridging that across the primary of our transformer, our switching elements need to be able to handle that much voltage, and in either direction. Common high-voltage signal MOSFETs like the 57¢ BSS131 won't cut it, and not just because of the body diode; we need much larger switches like the 82¢ STS1NK60Z (600V 0.25A), the obsolete 185¢ STF5N52K3 (525V 4.4A), the 89¢ STD2LN60K3 (600V 2A), the 63¢ AOD1N60 (600V 1.3A), the obsolete 116¢ STGB14NC60KT4 (600V 25A), the obsolete RoHS-non-compliant 122¢ STGD3NB60FT4

(600V 6A), or the 847¢ C3M0120065K (650V 22A, SiC). I think I need 8 of these for a full H-bridge to deal with the body diodes, which suddenly makes this seem a lot less appealing… maybe a center-tapped winding with diodes down to both live and neutral, and then I can get by with just two MOSFETs on the ends? Or I could just bridge-rectify the input, of course.

To get 500 watts out of 240 Vrms, I need only slightly over 2 A, and that's spread across two transistors.

Hmm, I wonder if this is some version of the "totem-pole PFC topology", though I'm not using it for PFC. Or maybe the "phase-shift full-bridge and LLC circuit"?

Topics

- Contrivances (p. 1143) (45 notes)
- Electronics (p. 1145) (39 notes)
- Pricing (p. 1147) (35 notes)
- Power supplies (p. 1176) (10 notes)
- Welding (p. 1181) (9 notes)
- Microcontrollers (p. 1211) (6 notes)

Argentine pricing of PEX pipe and alternatives for phase-change fluids

Kragen Javier Sitaker, 02021-08-07 (updated 02021-12-30) (2 minutes)

I've been thinking about some experiments with phase-change fluids and temperature control, and basically a big thing I need is a way to pump fluids around (sometimes salty fluids that could damage metals).

I found that people are selling PEX tubing for building radiant-heating floors. PEX is great because it's super inert and sturdy. Unfortunately only a couple of sizes seem to be available:

- 16 mm PEX for AR$154/m
- 20 mm PEX for AR$93/m
- 20 mm PEX for AR$110/m
- 20 mm PEX for AR$160/m in red

Other alternatives:

- 4mm transparent vinyl tube for AR$30/m
- 4mm transparent vinyl tube for AR$22/m for aquariums
- 16mm transparent vinyl tube for AR$117/m
- 3/4 inch (18mm) PVC pipe for AR$53/m
- 1/2 inch (12mm) black polyethylene tube for AR$26/m
- 3/4 inch (18.4 mm) black polyethylene tube for AR$40/m
- 1/2 inch (12mm) fiber-reinforced garden hose for AR$141/m
- 3/4 inch (18mm) corrugated polypropylene tube for AR$24/m
- 1.5 inch (38 mm) corrugated polypropylene tube for AR$123/m
- 150 mm corrugated aluminum air duct for AR$249/m for growing pot

Pure polypropylene, polyethylene, and PVC are also super inert, though these versions may not be. *Black* polyethylene has the advantage that it can absorb sunlight easily.

The thin aquarium tubing may be interesting not only for when girth doesn't matter (because it's 10% cheaper than anything else by length) but also because girth can actively be a drawback; it takes up space, it requires more coolant, it lengthens the transit time for coolant at a given mass flow rate, and, especially when full of water, it adds weight.

Topics

- Pricing (p. 1147) (35 notes)
- Argentina (p. 1200) (7 notes)

Power transistors

Kragen Javier Sitaker, 02021-08-07 (updated 02021-12-30)
(12 minutes)

I was looking at power transistors to use to control a load of some tens of watts (a tiny arc furnace) from a microcontroller, driving a couple of flybacks with something like 10 amps at 5 volts at a few kHz. But obviously the microcontroller can't drive 10 amps, so you need a high-power buffer, and it should probably be a transistor.

What buffer transistors would you want to use?

MOSFETs

In MOSFET-land, a pricey power MOSFET like an IRF540N is one option when you need low on-resistance; (0.044 ohms, 33 A, Vds up to 100, 145¢); another would be a parallel pair of IRLML6344s (0.029 ohms, 5 A each, 30 V, 36¢).

An interesting figure here is the expected load impedance. An IRF540N can control a 3300-watt load, but if it's resistive, that load has to be 3 ohms. If the load is 10 ohms, at 100 volts it can only draw 10 A, and so it's only 1000 watts. Similarly, if the load is 1 ohm, we can only use voltages up to 33 V, and so it can only be 1089 watts. So even though the actual impedance of the IRF540N is only 0.044 ohms, from the point of view of efficient power transfer, it's kind of like a power supply with a 3-ohm internal impedance. I'll call this the "virtual impedance".

In general if the virtual impedance is too low, you can stack up multiple switches in series to get the voltage you want. It may be inconvenient but it's probably not that bad. But if it's too *high* you may be in for trouble; BJTs need a lot of ballast to avoid current hogging.

Because you can parallel MOSFETs, too high a virtual impedance figure is less of a concern; paralleling them drops your virtual impedance just like it drops real impedance. So one IRLML6344 has "6 ohms" of virtual impedance, but two in parallel have "1.5 ohms", and four have "0.75 ohms".

The modern choice would probably be a GaN FET like the EPC2036. Damn, those things are sweet. 100 V, 1 A, 0.065 ohms, and only 0.91 nanocoulombs of Qg, compared to the IRF540N's 71 nC and the IRLML6344's 6.8. And its threshold voltage is lower, too. So you can switch it on or off much faster and with less energy. The 245¢ EPC2016C is 100 V, 18 A.

TIP120s are too voltage-greedy

You can't use a TIP120 darlington for this kind of thing; though it can deal with 60 V, 5 A continuous, 8 A peak, its saturation Vce is 4 volts at 5 amps, so it would eat basically all the power you were trying to feed the transformer.

60 V / 5 A is a virtual impedance of 12 ohms.

Other bipolars

So are there bipolars that would work better?

Digi-Key suggests the 55¢ 2STN1550, the 141¢ MJB41CT4G, the 232¢ MD2001FX, the obsolete 21¢ 2SD23210RA, or the obsolete 32¢ 2SD250400A.

21¢ 2SD2321

Starting with the cheapest, the 21¢ Panasonic 2SD2321 can switch 5 A (8 A peak) at up to 20 V with a beta of at least 150 at 2 A and a typical saturation Vce of 0.28 V when you're running it at 3 A, zooming up to 1 V at 8 A or so. It has a 150MHz "transition frequency", which I think means its beta is guaranteed to be not more than 15 if you're running it at 10 MHz.

20 V / 5 A is a virtual impedance of 4 ohms.

So, you could feed it 40 mA through a 120-ohm base resistor from a microcontroller GPIO pin, grounding the emitter, and running the flyback primary winding through it from 5V. The current through the coil starts to climb, and keeps climbing until we turn off the transistor, at which point the current leaps to the secondary and energizes the arc. If we don't buffer it further we probably won't get more than 6 amps out of it. But then it's dropping a whole volt, so it's dissipating 6 watts, briefly exceeding its 0.4 watt rating 15 times over. And it probably spends a fair bit of time dissipating more than half that. So it's probably going to overheat in its little bitty NS-B1 TO-92-like package.

But, even before that, quite likely at 5 V 2 A we drive the poor little transistor into second breakdown.

The max you could theoretically switch with this transistor, if second breakdown wasn't a consideration, is 100 W. In a flyback setup you won't get more than 40 W; the flyback waveform is an interrupted sawtooth, so its RMS value is half of its mean value, a quarter of its peak value of 160 W. With a 5 V supply, though, you'll be lucky to control 10 W with it. And because of its lousy power dissipation you can only control a tiny fraction of that continuously.

32¢ 2SD2504

This is slightly more promising, specced to switch 5 A (9 A peak) at up to 10 volts, and dissipate 750 mW from its TO-92-B1. But its saturated Vce crosses 1 V at only 4 A; at 8 A it's up to 2 V (and thus 16 W). So it's just going to dissipate way too much power for this, even if it doesn't hit second breakdown (Panasonic forgot to include the safe-operating-region plot in the datasheet this time).

10 V / 5 A is a virtual impedance of 2 ohms.

55¢ 2STN1550

This is a little bitty surface-mount SOT-223, which entitles it to dissipate 1.6 watts, and it's rated to switch up to 5 A (10 A peak) at up to 50 V; at 5 A 5 V it says its (non-small-signal) beta is typically 95, so you'd need 53 mA to avoid saturating, which is a bit much to ask from a microcontroller. It turns on in 90 ns and off in 700 ns, so you can switch efficiently at near-MHz rates.

50 V / 5 A is a virtual impedance of 10 ohms.

ST omits the performance curve plots entirely, as it turns out, only specifying a 0.26 V saturated Vce at 3 A.

I feel like this transistor would probably work in a 12V system! But in a 5V system we're just asking too much current from it. Say we can drive its base with another transistor so we don't have to worry about the 40 mA limit on AVR pins. 5 amps at 5 volts is a peak of 25 watts; in a flyback setup we can never get more than half of that, since if the *mean* current of the sawtooth is 5 amps, its *RMS* current is only half of that. So we're talking about 12.5 watts, which is not much of an arc furnace. If we were using 12 volts, though, we could do 30 watts with a switch like this.

I don't know, I think we'd need to go for something a lot beefier to get hundreds of watts of power into our stepdown flyback arc power supply.

141¢ MJB41CT4G

This is a TO-263-3 surface-mount version of the TIP41 power transistor, specced to switch 6 A (10 A peak) at *100* volts, but with a beta of only 15 and a transition frequency of only 3 MHz (which the onsemi datasheet helpfully explains *is* the gain-bandwidth product), which limitations would be fine for this application. You'd have to use some kind of a driver circuit to drive its base: another transistor, a step-down pulse transformer, something.

100 V / 6 A is a virtual impedance of 17 ohms.

It's rated both for 2 watts and 65 watts dissipation, depending on whether you're holding the case or the ambient air at 25°. The junction temperature max is 150°, junction-to-case thermal resistance is 1.92°/W, and junction-to-ambient thermal resistance of 62.5°/W (or 50°/W "when surface-mounted to an FR-4 board using the minimum recommended pad size"). If you divide 150°-25° by 1.92°/W you get the 65-watt number, while 62.5°/W gives you the 2-watt number. So if you heatsink this guy well enough, you could dissipate tens of watts.

It says its saturated Vce is 1.5 V at 6 A, which would be 9 watts, so that's really all the heatsinking it needs if you're using it as a switch. At 5 volts it would be grossly inefficient, controlling a 3.5 V 6 A (21 watt) load at a cost of 9 watts. But if it were controlling, say, a 70-volt load, it might be fine.

232¢ MD2001FX

This is a monster 700-volt bipolar 12-amp (18-amp peak) 58-watt NPN BJT, marketed as "High voltage NPN power transistor for standard definition CRT display"! It's a transistor specifically designed for the horizontal deflection output for a CRT, but astonishingly it's not marked as "obsolete" and it was only introduced in 02007. Its beta is only 4.5, and it's sloow, 2.6 microseconds storage time.

700 V / 12 A is a virtual impedance of 58 ohms.

Triacs

Suppose you connect the flyback primary between the positive power rail and a capacitor to ground and put a triac (or just an SCR)

across the capacitor. Initially when you plug it in there will be a flyback pulse as the capacitor charges up, to twice the input power rail, I think, but then when the cap starts to discharge, the flyback secondary's diode will go forward-biased and rapidly drain most of the energy out of the circuit, so fairly rapidly the cap will be charged to the power rail voltage, and everything will be quiescent. But if you tickle the SCR gate, the cap dumps to ground and the whole cycle starts again. You can control the amount of power delivered to the output by doing this more or less often. A triggered spark gap could maybe substitute for the triac in a pinch. I mean basically this is just a Tesla coil.

This doesn't seem like it is going to be very efficient, but it would definitely work.

I am going to assume that the 46¢ Ween (formerly NXP, formerly Philips) Z0109MN0 is a typical triac. It's an SOT223 four-quadrant triac that starts passing 1 A at 600 V (dropping 1.3 V) if you tickle its gate with 1 V and 10 mA, until the current drops below 10 mA (or maybe 30 mA?).

600 V / 1 A is a virtual impedance of 600 ohms.

Topics

- Electronics (p. 1145) (39 notes)
- Pricing (p. 1147) (35 notes)
- Power supplies (p. 1176) (10 notes)

Pocket kiln

Kragen Javier Sitaker, 02021-08-09 (updated 02021-08-15)
(7 minutes)

As mentioned in Material observations (p. 633), my experiments on materials are limited by needing a kiln that can maintain a reproducible temperature somewhere in the 500°–1500° range for probably considerably longer than 20 minutes.

The thermal aspect of this seems like an eminently feasible thing to do with the intumescent refractory recipes explored therein: I can mix up some intumescent mix, maybe with a bunch of vermiculite filler, mold it to a rough shape, and heat it with the torch enough to get it to bubble up; then I can cut and grind it to shape with saws, knives, bricks, bits of granite, and so on. Maybe a little hardfacing with waterglass-bonded sand or alumina would be useful to improve durability, or maybe a bed of vermiculite would be adequate. That would already be a significant improvement over just an open bowl of vermiculite or even the Monster-can forge I've been using.

(If I didn't have a butane torch, a stove burner with a metal bowl, or a wood fire, would probably work too. We're getting into serious bootstrapping territory here! Also worth mentioning is using a pile of loose particulate, supported by the thing being heated up and possibly fuel, instead of a solid refractory.)

Temperature control involves at a minimum some kind of electrical circuit, probably a temperature sensor, and probably one or both of airflow control (say, with muffin fans well upstream of any heating) and electrical heating, say with a resistive heating element. Without electrical heating, blowing a variable amount of air through ignited charcoal, ignited yerba mate, or some other burner (maybe an oil burner like the one I made with the porous magnesium silicate) could add a variable amount of heat to the chamber; mixing that air with a second airstream could control the temperature of the air being introduced to the kiln as well as reducing unburnt fuel contamination and making the atmosphere oxidizing again if desired.

Once I can do that, making further components from fired-clay ceramics should be easy!

Although this involves contaminating the kiln atmosphere with the fuel, and it has a relatively low maximum temperature, this sort of thing has the advantage that very high powers can be easily attained without imposing a significant load on the electrical system. A typical US$10 computer case fan might use 200 mA at 12 V and be able to provide 30 "cfm", cubic feet per minute, which is 14 liters per second, which works out to 3.6 grams per second of oxygen (21% of 1.2 g/ℓ). (Maybe that's too pessimistic; the first fan I checked on Amazon claims 56.3 cfm at 0.96 W, but a US$2 40mm fan claims 6.7 cfm at 1.2W.) Suppose you're burning charcoal, and that it's basically the same as graphite; one mole of O_2 (31.998 g/mol) produces one mole of CO_2 (44.009 g/mol), which has a standard enthalpy of formation of -393.5 kJ/mol. So crudely you get about 12.30 kJ/gO_2, so at 3.6 g/s you get **44 kilowatts** of heating power, while the fan motor is

only using 2.4 watts. The charcoal amplifies the fan's heat output by a factor of 18000. Or 86000 if we believe the Amazon large fan number, or only 8200 if we believe the Amazon small fan number.

(In a sense this design involves two combustion chambers: one with fuel in it, which will increase the rate of fuel volatilization when airflow through it increases, and a second one downstream from it, which serves to ensure complete fuel combustion and perhaps restore oxidizing conditions by adding enough oxygen to the exhaust from the first combustion chamber, and which does not transmit heat to the still-unburned fuel.)

Fuel burning is of course much more difficult to control.

Using a resistive heating element upstream from the kiln chamber itself would enable me to maintain a reducing atmosphere in the kiln (with a little sacrificial carbon or something, similar to the ignited-charcoal-heater approach) without needing exotic heating elements (graphite, carborundum, zirconia, platinum, fused-quartz-encapsulated tungsten) that can withstand reducing atmospheres at high temperatures.

Running the flue gases through a scrubber would be pretty useful to avoid poisoning the neighborhood, creating noxious smells, and setting things on fire with hot gases. A bubbler, a spray-nozzle column, or a sewage-treatment-style trickling-filter sort of arrangement (see file `liquid-packed-beds.md`)would enable a large contact surface area between the flue gases and the water. Driving the whole thing with a suction pump at the output of the scrubber would keep the pump itself from being exposed to hot gases and melting, as long as the scrubber itself is cooling the gases adequately.

Even if the kiln isn't heated by combustion, flue gases are a concern, because reactions inside the kiln can produce gases which can be stinky, poisonous, or both, and of course the output will be hot. But it may be possible to have less of them!

Carbon monoxide (perhaps from incomplete combustion of organics inside the kiln) is a particular concern because it's hard to remove; you pretty much just have to burn it.

Alternatives to combustion for heating include direct resistive joule heating, indirect resistive heating by way of an induction coil, dielectric heating, microwave heating (which provides indirect resistive and dielectric heating), direct arc heating with consumable electrodes (typically graphite), indirect arc heating with microwaves, self-propagating high-temperature synthesis, and solar furnaces like Lavoisier's. Of these, the last two have natural temperature limits, and the others basically don't.

Once I have a refractory ceramic automated fabrication capability working, it should be possible to make fan blades and even Parsons turbines out of refractory ceramics, which straightforwardly would seem to make it relatively easy to pump hot gases directly. But I'm not sure how to transmit the rotation to the blades: through a long shaft that traverses refractory gland packing? (This is a "bifurcated fan" in industrial fan lingo.)

Topics

- Contrivances (p. 1143) (45 notes)
- Thermodynamics (p. 1219) (5 notes)
- Refractory (p. 1225) (5 notes)
- Heating (p. 1253) (4 notes)
- Insulation (p. 1290) (3 notes)

Cola flavor

Kragen Javier Sitaker, 02021-08-10 (updated 02021-08-15)
(2 minutes)

I accidentally inhaled a little acid gas from heating diammonium phosphate (see Material observations (p. 633).) It was just a whiff, but it set me to thinking.

It occurs to me that, because it polymerizes above 200° instead of simply boiling like a decent liquid should, hot phosphoric acid with some impurities might outgas a bunch of muriatic (−114.22°) or vitriol (over 300°), which is a potential inhalation hazard from heating it (something I hadn't thought to consider hazardous); electrolysis of ammonium compounds could suffer the same problem. (Brown & Whitt 1952 say the aqueous azeotrope boils at 840° at 92.7%.)

I wonder if the Potash Corp.'s helpful booklet mentions this? Well, it says you should use rubber-lined (latex or chlorobutyl) steel, PE, polyester, or (if you can keep Cl content below 15 ppm, so don't dilute it with chlorinated water!) 316L for the stuff. They especially warn you away from carbon steel, and they warn that over 3000° F (1650°) it decomposes into phosphorus oxides. They have reassuring words about toxicity: LD_{50} of 1530 mg/kg, slight dermal toxicity, but it can cause acid burns, especially in eyes. Also on p. 24 they have a reassuring graph of "vapor composition over boiling solutions", with <1% H_3PO_4 up to 300° ("at which point the acid strength is about 103%", i.e., some of it has become pyrophosphoric acid), crossing 10% around 500°, and continuing up to about 57% at 800°. It never mentions the risk of protonating other substances to create volatile acid gases.

Topics

• Materials (p. 1138) (59 notes)
• Phosphates (p. 1184) (9 notes)

Constant current buck

Kragen Javier Sitaker, 02021-08-10 (updated 02021-08-15)
(4 minutes)

I was talking about flashlights and mentioned that I'd tried designing a constant-current buck converter without success. But now I don't see what was so hard about it. Maybe when I submit this design to simulation I'll find out.

The basic constant-voltage buck converter is GND->|-a-R1-b-L1-c-C1-GND. The input power waveform is applied between GND and a; the output voltage appears across the output capacitor C1. Subject to enough output load to keep it in continuous conduction mode (CCM), the output voltage is the average input voltage, so if the input voltage is a fixed voltage with a fixed duty cycle, then the output voltage is inherently regulated and will change very little with load. The current shunt resistor R1 allows you to measure the average load current, filtered through the LC low-pass filter that forms the output; when this is not necessary it can be 0Ω.

To drive this circuit, though, you need some kind of square-wave oscillator that switches point a between between a constant input voltage and open circuit (or, in CCM, between the voltage rails, in which case you don't need the freewheel diode ->|-.). You can use an opamp configured as a relaxation oscillator, for example, or a comparator between a sawtooth or triangle oscillator and a reference voltage level which determines its duty cycle.

My thought was that you should be able to use a single opamp as a differential amplifier across points a and b to set the reference voltage level for that comparator; for example, hook up point b to its noninverting input and point a through a 10k resistor to its inverting input, then a 100k resistor between its inverting input and its output. So then if point a swings up 0.1V relative to point b, the output must swing down by 1V to compensate.

I think you can probably make it simpler, though: by adding an error-integrating capacitor and positive feedback for a Schmitt-trigger effect, you should be able to integrate the square-wave generation into the same opamp rather than needing a separate ramp generator and comparator. I'm kind of fuzzy on how to actually do this, so I'm not sure you can do it all with a single opamp, but I suspect you can.

The capacitor across the output of the buck converter low-pass filters the current signal you're measuring and can briefly source or sink immense amounts of current itself, so if you want a *really* constant-current supply (perhaps because overshooting your current limit will burn up your expensive laser diode) you might want to make it very small or remove it entirely. Such extreme measures would probably require clamping the output with a TVS or at least a regular zener.

This feedback circuit is pretty close to being a class-D amplifier — I think you can use the same approach of modulating a square-wave

duty cycle with the integrated difference between a reference signal and the variable-duty-cycle square wave itself to get a class D audio amplifier. The only difference is that the "audio" amplifier input here is tied to the low end of the current sensing shunt resistor. It would be pretty cool if you could make a class D amplifier out of a single op-amp.

Topics

- Electronics (p. 1145) (39 notes)
- Frrickin' lasers! (p. 1168) (12 notes)
- Power supplies (p. 1176) (10 notes)

Methane bag

Kragen Javier Sitaker, 02021-08-10 (updated 02021-08-15)
(8 minutes)

As explained in Material observations (p. 633), I filled a small plastic shopping bag with methane, but to my disappointment it did not rise into the air. Calculations suggest that methane has $(12+4)/(.79\times2\times14+.21\times2\times16)\approx55\%$ of the density of air, so should provide some 500 mg/ℓ of lift. I didn't weigh this bag beforehand or measure its volume, but it surely only contained a liter or two of air. Average plastic shopping bags weigh about 5 g, but lightweight ones weigh as little as 100 mg. None of the ones I have here weigh less than 800 mg; a household garbage bag (550 mm × 480 mm) weighs 10.9 g, and another bag nearly as big (470 mm × 460 mm) only weighs 4.5 g. A 1.1 g pharmacy bag (170 mm × 240 mm) weighed 1.1 g. Several bags of intermediate size weigh around 3 g. The sturdy 8-liter bags from the sandwich shop weigh a little over 9 g.

If we approximate the volume of the 4.5-g bag as a 470-mm-long cylinder wih 460-mm half-circumference, it would hold almost 32 ℓ, enough to lift about 16 g, so it should definitely fly, but only by a factor of about 3. (And so should the garbage bag.) If we scaled it down by a factor of 3 to a 160-mm-long cylinder with 150-mm half-circumference and 500 mg of mass, it would only hold 1.1 ℓ and thus be just on the edge of buoyancy. Most of my bags seem to be a lot thicker than that, so they'll only fly if they hold several liters.

4.5 g over twice 470 mm × 460 mm is 10.4 g/m², so if the plastic is close to 1 g/cc (it is — it's high-density polyethylene, not lead or something) it's about 10 μm thick, same as the aluminum foil we get here. So this is probably about as good as it gets without going to gold leaf or exotic composites. Note that this suggests that ordinary balloons (and bladder and intestine tissue) stretch even thinner than this.

Lifting this body into the air would require 115 kg of lift, a sphere of 2.8 m radius if filled with vacuum, or with methane (45% of the lift of vacuum) 3.7 m, or 170 m² of surface area. Hydrogen would probably be a less dangerous lifting gas for humans suspended below it, with more tendency to rise upwards when there's a rupture and less tendency to radiate heat downwards if it catches on fire. There's still the problem of how to stop falling before the ground kills you, though.

Unfortunately Mina doesn't want me to fill a bag with 32 liters of inflammable gas in the kitchen, which I have to say is an entirely reasonable preference. And either compressing the gas or running a hose to the park seems daunting. And, although CNG cars are common here, we don't have a friend who has one.

A hydrogen generator would be a much more portable solution, and has the advantage that hydrogen is only 7% of the density of air, so you get a little over twice as much lift. A 1-ℓ sphere 124 mm in diameter has a surface area of 0.048 m² and would thus require 480 mg of 10-μm polyethylene to enclose it, so the critical limiting

scale with those materials would be about 50 mm in diameter: 65 mℓ, 0.0079 m², 79 mg of plastic, 79 mg of air displacement. 7% of 1.2 g/ℓ is 80 mg/ℓ, so you'd need 80 mg of hydrogen to fill a 1-ℓ sphere. Let's pick 2 ℓ as a reasonable size, with a good bit of safety margin for non-idealities. If we use a tubular bag that's 120 mm in diameter, it will be 177 mm long and, when stretched out flat, 188 mm wide. At 10 μm two such rectangles are 666 μℓ and thus about 666 mg of HDPE. The requisite 160 mg of hydrogen is 0.079 mol, which the universal gas law tells us is 1.9 ℓ — off by 5%.

To make it we need to electrolyze 0.079 mol of water, about 1.4 g, which seems like an eminently practical amount of water to carry to the park. If our Faraday efficiency were 100%, each electron would liberate a hydrogen ion, so we would need 0.158 moles of electrons, about 9.5×10^{22} electrons, which turns out to be 15 kilocoulombs, 4.2 amp hours; at 2.4 volts that's 37 kJ; at 200 watts that's 3 minutes. And in practice I think the Faraday efficiency will be more like 60%, so it would be more like 6 minutes.

NREL Conference Paper NREL/CP-550-47302 explains that electrolysis requires 237.2 kJ/mol of electricity and 48.6 kJ/mol heat. The number I came up with above is 234 kJ/mol (37/0.158 = 234), so I guess I did the ideal-Faraday-efficiency calculation right, but it didn't occur to me that the reaction was endothermic! Although this must be purely by chance, since they're saying the actual voltage is 1.229 volts, not 2.4 volts. “Whereas the practical fuel cell operates well below 1.23 volts (in the range of 0.750 to 0.900 volts), the practical electrolysis cell operates above this voltage in the range of 1.60 to 2.00 volts.”

Well, 285.6 kJ/mol * 1.8 V / 1.23 V = 420 kJ/mol, or 66 kJ for 0.158 mol. So basically 100 kJ.

100 kJ of batteries is about US$4 of lead-acid batteries; the US$8.30 2-kg 7-amp-hour Risttone battery mentioned in Energy autonomous computing (p. 143) holds 300 kJ, 150 kJ/kg. A lithium-ion version would be about ⅙ the weight for the same energy capacity, or ⅓ for a high-power battery, but those cost ten times as much per joule or more, and the limiting factor becomes power rather than energy. So we're probably talking about a battery that's practical, but a bit cumbersome, to carry to the park.

A different alternative would be to fill the bag with azane, which has even less lifting power (17.031 g/mol compared to methane's 16.043 or air's 28.9) but can be dissolved 30% by weight in water at 25°, forming a strongly basic 26°Bé (1.22 g/cc) solution; at 0° this rises to 47% by weight, and at 60° to something like 10%. 20 ℓ of ideal gas would be 0.831 mol, 24.0 g of air or 14.2 g of azane, thus lifting 9.8 g. 14.2 g of azane could be dissolved in 47.3 g of aqueous azane solution occupying about 38 mℓ, then liberated by heating it, as in an absorption refrigerator. (And you'd need a “water separator” or “moisture separator” or “mist eliminator”, also as in an absorption refrigerator.) A disadvantage of this is that it has a strong smell that the vulgar may associate with its use in witchcraft.

At pure azane's boiling point of -33.4°, its $\Delta_{vap}H\Theta$ is 23.35 kJ/mol, so perhaps at a higher temperature from aqueous solution it would be a bit lower; maybe this would require 15 kJ of heating. This doesn't

sound like much of an advantage over 66 kJ, but you could supply it with a candle, or with thermochemical energy storage like muriate of lime (combined with water in a separate chamber).

The other traditional way to make hydrogen on demand is with lye and aluminum foil; zinc and muriate of lime reportedly also work, and hydride of calcium has been used for inflating weather balloons for a long time. I guess I should figure out the stoichiometry of this; a couple of liters seems like it ought to be doable...

Topics

- Materials (p. 1138) (59 notes)
- Pricing (p. 1147) (35 notes)
- Physics (p. 1157) (18 notes)
- Experiment report (p. 1162) (14 notes)
- Flying (p. 1296) (3 notes)
- Batteries (p. 1302) (3 notes)
- Azane (p. 1386) (2 notes)

Iodine patterning

Kragen Javier Sitaker, 02021-08-11 (updated 02021-08-15) (1 minute)

Reading some random encyclopedia article, I came across a reference to the dissociation energy of I_2 being (the photon energy of light of wavelength) 578 nm, so that irradiating a mixture of H_2 and I_2 with this light (kind of yellow-green apparently) accelerates the formation of HI.

It occurred to me that monatomic halogens can react with a remarkably large number of things, so doping a thin layer of something with I_2 might be a straightforward way to make it photosensitive, so that you can pattern it with a yellow, green, blue, violet, or UV laser. Where the laser strikes, free iodine radicals are released, kicking off some kind of polymerization or depolymerization or something.

Dunno, I guess iodine is not a very convenient element.

Topics

- Materials (p. 1138) (59 notes)
- Digital fabrication (p. 1149) (31 notes)
- Frrickin' lasers! (p. 1168) (12 notes)
- Patterning (p. 1282) (3 notes)

Heating a shower tank with portable TCES?

Kragen Javier Sitaker, 02021-08-11 (updated 02021-08-15)
(6 minutes)

The shower tank in Mina’s bathroom holds 25 ℓ, though a typical shower is more like 20 ℓ, and the water is preheated by a gas on-demand heater. Unfortunately the plumbing was apparently done by the same idiot or would-be murderer who installed the deadly fake electrical system, because the water arrives at the shower tank at a lukewarm temperature of about 30°. Raising it to a luxurious 41° would be desirable; currently I do this by pouring boiling water into the top from a pot.

If we disregard the variation of water’s specific heat over the relevant range, $v_0 t_0 + v_1 t_1 = 41°\ 20\ \ell$, $t_0 = 30°$, $t_1 = 100°$, and $v_0 + v_1 = 20\ \ell$, so $v_1 = 20\ \ell - v_0$ and we have $30° v_0 + 100°(20\ \ell - v_0) = 41°\ 20\ \ell = (30° - 100°)v_0 + 100° 20\ \ell$, so $v_0 = (41° - 100°)20\ \mathrm{l} / (30° - 100°) = (41° - 100°)/(30° - 100°)20\ \ell = (59/70)20\ \ell = 16.9\ \ell$, so 3.1 ℓ is the amount of boiling water to add, and indeed lerp(30°, 100°, 3.1ℓ/20ℓ) is 40.85°. If it’s just a matter of adding energy to 20 ℓ of 30° water, though, it’s 921 kJ.

But maybe we could use a much smaller heating unit using TCES, for example with muriate of lime (see file `muriate-thermal-mass.md` in Derctuo) sealed in a sturdy plastic bag. The idea is that you have some kind of “heat pack” that you dunk in the tank, then activate, and it adds 921 kJ of heat to the water, and you have a nice shower and then fish it out of the tank and go recharge the heat pack.

How small can 921 kJ be? Fully hydrating the anhydrous salt to the hexahydrate gives you (2608.01 - 795.42 = 1812.59) kJ/mol (at 110.983 g/mol, that’s 16.3321 kJ/g), but I think you can get further heat by dissolving the salt and diluting the solution; I’m not sure. 6 mol of water weighs 108.09 g, 2% less than the mass of the anhydrous salt it would fully hydrate, dropping the energy density of the heat pack to about 8.1 kJ/g.

This would give us a 114-gram heat pack, which is a *lot* less than the 3.1 kg of boiling water.

It wouldn’t quite work that well, though: the hexahydrate dehydrates to the tetrahydrate above 30°, and the tetrahydrate only has an enthalpy of formation of -2009.99 kJ/mol, leaving only 1214.57 kJ/mol — a third of the stored heat is inaccessible at temperatures above 30°. We need less water, though, only 72.06 g/mol. So you get 1214.57 kJ per 183 g of heat pack, or 6.6 kJ/g, pushing the heat pack mass up to 140 g.

This same phenomenon limits the temperatures the heat pack will expose its envelope to. The various hydration forms have these characteristics, according to wikipedia and my calculations:

H_2O	decomposition	enthalpy of formation	o

0	2.15 g/cc	772-775° (melts)	-795.42 kJ/mol	110.98 g/mol	0	8.274 kJ/g	72.89 J/mol/K	0.6568 J/g/K
1	2.24 g/cc	260°	-1110.98 kJ/mol	129.00 g/mol	2.446 kJ/g	5.828 kJ/g	106.23 J/mol/K	0.8235 J/g/K
2	1.85 g/cc	175°	-1403.98 kJ/mol	147.01 g/mol	4.140 kJ/g	4.134 kJ/g	172.92 J/mol/K	1.176 J/g/K
4	1.83 g/cc	45.5°	-2009.99 kJ/mol	183.04 g/mol	6.636 kJ/g	1.638 kJ/g	251.17 J/mol/K	1.372 J/g/K
6	1.71 g/cc	30°	-2608.01 kJ/mol	219.07 g/mol	8.274 kJ/g	0	300.7 J/mol/K	1.373 J/g/K

The two reversed columns of energy density say that, for example, the tetrahydrate (plus the water to hydrate it the rest of the way) is an energy store of 1.638 kJ/g, but in hydrating the anhydrous salt to the tetrahydrate we produced 6.636 kJ/g of heat. If that heat couldn't go anywhere else, it would have raised the temperature of the resulting material by 4837°; the reason this salt doesn't explode is that it stops absorbing water once it gets warm enough.

In particular, the corresponding calculation for the monohydrate produces a result of 2970°, so if water is limited enough, even in a small region, you should be able to reach 260° by hydrating the muriate of lime.

For a plastic bag, this might be bad; nylon 6 melts at 220° and nylon 6,6 melts at 264°, and those are the likely plastics available in a supermarket oven bag. It would also be a problem if one part of the material reached 100° while nearby some water was still pure — the water might boil, float the bag, and burst the bag with steam pressure. So it might be best to compromise on energy density in order to limit the maximum possible temperature; while in theory you could do this by diluting anhydrous muriate of lime with glass beads or something, I think it's probably better to mix together two hydration levels to set the maximum temperature they can reach when fully hydrated to below 100°.

Drying the muriate for reuse can be tricky; a 01964 US patent 3,339,618A describes the situation at the time. The guy had come up with a spray-drying process using 370° air to produce anhydrous calcium chloride.

A different source gives a much lower enthalpy for hydrating muriate of lime. A third source mentions that through repeated cycling, some amount of the salt will hydrolyze into marine acid air; I'd think that a small amount of chalk in the mix would be adequate to prevent this, but they seem to think it's a harder problem.

Topics

- Materials (p. 1138) (59 notes)
- Energy (p. 1170) (12 notes)
- Thermodynamics (p. 1219) (5 notes)
- Heating (p. 1253) (4 notes)
- Enthalpy (p. 1369) (2 notes)

- TCES energy storage

Subset of C for the simplest self-compiling compiler

Kragen Javier Sitaker, 02021-08-12 (updated 02021-12-30) (6 minutes)

What's the smallest straightforward self-compiling compiler, targeting a conventional assembly language, you could write in a subset of C?

That is, not the smallest subset of C; implementing a very small subset of C means the compiler doesn't have to do as much, but being written in that same very small subset means that everything is more difficult to do. Also, though, I'm thinking of something I could reimplement straightforwardly in assembly.

You need comments.

You surely need subroutines --- syntactically C needs them at least for main(), and almost certainly there will be things you want to do in more than one place. There's a question of whether or not to implement recursion. Without recursion you could statically allocate all variables and only have a single way to compile lvalue variables and a single way to compile rvalue variables, but you need `while`. With recursion you can use a recursive-descent parser, which you probably should, but you probably need to store some variables in global space and others in stack frames.

Arguments and return values can be omitted by storing them in global variables, but I think that will probably obscure the data flow a lot. If there are arguments, you don't need an arbitrarily large number of them.

Alternatives to recursive-descent parsing with local backtracking (PEG parsing) might be more compact but are unlikely to be as straightforward.

You need some form of data structuring, either structs or arrays or both, which means at least one additional kind of lvalue and rvalue. You don't need separate struct namespaces, and if you don't have them you can avoid having types for expressions at all, treating characters, ints, and pointers interchangeably as words, as putchar() and getchar() already do. Structs might be a big improvement, but they probably mean you need data of varying sizes.

You could reasonably support such composite data structures only at global scope, so local variables are only scalars; and, if you do that, shallow binding might be an alternative to using separate indexing schemes for globals and locals. Locals would just be globals whose value is saved and later restored.

Structs are more appealing if you have dynamic allocation, so you can build trees out of them using pointers.

In terms of arithmetic, you almost certainly need addition, subtraction, and integer constants. You don't need pointer arithmetic, and you probably don't need bitwise operations, multiplication, division, and modulo. You surely do need ==, !=, and at least one

kind of ordering comparison.

Boolean operations `&&`, `||` might turn out to be painful to do without. Similarly for augmented assignment `+=`, `-=` and pre- and/or post-increment `--`, `++`.

In terms of control flow, you surely need `if`, and if you don't have `else`, you probably need early `return` (and thus `return`). I don't think there's any advantage to not having nested blocks for `if`, and little advantage for not requiring them. Early `return` is more complicated with shallow binding (you'd probably want to jump to a shared epilogue to restore the proper set of variables).

I think that's a sufficient set of statement types: either expression statements or assignment statements and function calls; `if`; and possibly `while`. If we have return values, we need `return`. Local variable declarations are needed for local variables, and although C doesn't treat them as statements (in particular, you can't precede them with a label) I think that's probably wrong. Still, it might turn out to be simpler to require them to all be declared at the top of the function, as in Smalltalk.

Modern machines have enough registers that you could reasonably statically allocate one register for a frame pointer, three or four registers for arguments and returns, and, say, three or four other registers as temporaries. Nested function calls would still require storing the temporary result in someplace that isn't clobbered by calls, either in the stack frame or a callee-saved register, so having a few callee-saved registers would be handy.

All of that is far more complicated than just using a runtime stack for expression evaluation and passing parameters and return values, though. Variadic C functions sort of require the caller to pop the arguments, but the subset doesn't have to include variadic functions. Or, as I said, arguments at all.

At least ignoring `#include` is probably necessary in practice. Lacking either `#define` or `enum` would be a pretty big impediment to readability, though, comparable to lacking arguments.

Writing a C compiler without string literals would be pretty hard. Doing it without data initializers, like basically every Wirth compiler, wouldn't be particularly hard, but I think you do need string literals. However, I think C-compatible string literals more or less require some kind of pointer support; I don't think we can pass them off as offsets into a table of all constant strings, because we need to be able to build up new strings at runtime. I think this pretty much forces on us the ability to take a pointer into the middle of an array, and thus C's equivalence of `a[b]` with `*(a+b)`, though, and thus `*`. Maybe we can still get away without expression types (which we would need for the implicit multiplication by sizeof) by shifting the pointers left by 2 or 3 bits before dereferencing them, but of course that would break ABI compatibility with everything else. I'd sure like to find a way to avoid this mess.

For writing the tokenizer you probably also need character literals.

If you didn't need C compatibility, for many purposes, you could in fact use indices into a global string table as your string type, with maybe a couple of character buffers elsewhere used by other functions

to build up strings incrementally before interning them.

Topics

- Bootstrapping (p. 1171) (12 notes)
- Compilers (p. 1178) (10 notes)
- Small is beautiful (p. 1190) (8 notes)
- C (p. 1194) (8 notes)
- Control flow (p. 1299) (3 notes)

A compact bytecode sketch that should average about 3 bytes per line of C

Kragen Javier Sitaker, 02021-08-17 (updated 02021-09-13) (66 minutes)

From Table I we see that the assignment, IF, CALL, RETURN and FOR statements together account for 96 percent of the source statements. Therefore we will design an instruction set to handle the object code from these statements efficiently. To push local variables (including parameters) onto the stack, we propose 12 distinct 1-byte (format 1) opcodes, one each for offsets 0–11. Twelve instructions allow access to all the locals (and parameters) in 94.6 percent of the procedures, and to more than 50 percent of the locals in the remaining procedures. For example, opcodes 114–125 might be used for PUSH LOCAL 0, PUSH LOCAL 1..., PUSH LOCAL 11.

(Tanenbaum 01978, §5, p. 243)

What would a compact stack bytecode for C look like? I think you could usually manage to compile to about 3 bytes per line of C code, which would enable you to run C programs of about 10kloc on an Arduino or about 300kloc on an Ambiq Apollo3, in exchange for nearly an order of magnitude worse power usage and runtime for the interpreted code. This could expand the uses of such small computers dramatically.

Why?

There are two major reasons to do this: to pack more functionality into less memory and to improve in-application programmability.

To pack more functionality into less memory

There are lots of microcontrollers around now, but for traditional personal computing purposes they have more speed than needed and less memory. The 3¢ Padauk microcontrollers (the PMS150 family) as described in file `minimal-cost-computer.md` in Derctuo has “512–4096 words of program memory and 64–256 bytes of RAM, all running at up to 16 MHz (normally 8 MIPS) ... running on 2.2–5 volts at 750 μA/MHz,” though they’re out of stock everywhere last I saw. As mentioned in A bargain-basement Holter monitor with a BOM under US$2.50 (p. 323), the ATTiny1614 costs 82¢ from Digi-Key; it has 16KiB of program memory (8192 words) and 2048 bytes of RAM and runs at 20 MHz (≈20 MIPS). (The formerly cheaper ATTiny25V-15MT is now out of stock due to the Chipocalypse and its price has shot up enormously to US$1.82 in quantity 1000.) The STM32F051C8T6 is still out of stock at Digi-Key with a standard lead time of 52 weeks, and now the CKS32F051C8T6 is out of stock at LCSC too, but at least they have the GD32F130C8T6: 64KiB program memory, 8KiB RAM, 72 MHz Cortex-M3 (80 DMIPS?), US$3.25. And the Ambiq Apollo3 mentioned in Energy autonomous computing (p. 143) has 1MB Flash and 384KiB SRAM and is a 48MHz Cortex-M4F, running at about 10 μA/MHz at 3.3 volts.

In How do you fit a high-level language into a microcontroller?

Let's look at BBN Lisp (p. 160) and Energy autonomous computing (p. 143) I've written about how to effectively use off-chip memory in order to kind of bridge this gap somewhat. Swapping in code or data from off-chip Flash is a great deal faster than swapping from a spinning-rust disk drive or loading overlays from a floppy, so it can be a lot more transparent.

However, an alternative approach is to make more frugal use of the internal memory, and in How do you fit a high-level language into a microcontroller? Let's look at BBN Lisp (p. 160) (and in file `tiny-interpreters-for-microcontrollers` in Dercuano) I explored compact bytecode formats a bit.

The Microsoft Excel team wrote their own C compiler in the 01980s for this purpose; as I understand it, to fit more functionality into PCs of the time, such as the Macintosh, their compiler allowed them to choose for each function whether to compile it to bytecode or to native code.

The MakeCode VM reportedly consists of about 500 bytes of AVR machine code:

> MicroPython and similar environments cannot run on the [Arduino] Uno due to flash [32 KiB] and RAM [2 KiB] size limitations. We also ran into these limitations, and as a result, developed two compilation modes for AVR. One compiles STS [Static TypeScript, a JS-like language] to AVR machine code, and the other (MakeCode VM) generates density-optimized byte code for a tiny (~500 bytes of code) interpreter. The native strategy achieves code density of about 60.8 bytes per statement, which translates into space for 150 lines of STS user code. The VM achieves 12.3 bytes per statement allowing for about 800 lines. For comparison, the ARM Thumb code generator used in other targets achieves 37.5 bytes per statement, but due to the larger flash sizes we did not run into space issues.

In exchange for this 4.9× code compression, they accepted about a 6.4× interpretive slowdown.

The Aztec C compiler for the Apple (available for free download)included a bytecode mode to reduce space, and you could compile only part of your program with it:

> As an alternative, the pseudo-code C compiler, CCI, produces machine language for a theoretical machine with 8, 16 and 32 bit capabilities. This machine language is interpreted by an assembly language program that is about 3000 bytes in size.
>
> The effects of using CCI are twofold. First, since one instruction can manipulate a 16 or 32 bit quantity, the size of the compiled program is generally more than fifty percent smaller than the same program compiled with C65 [the Aztec C native code compiler for the 6502]. However, interpreting the pseudo-code incurs an overhead which causes the execution speed to be anywhere from five to twenty times slower.

Chuck McManis famously reverse-engineered the Parallax BASIC Stamp, based on a PIC16C56, and found that it used a variable-bit-length "bytecode" system to take maximum advantage of the 2048 bytes of its serial EEPROM. Most of the "tokens" in the "bytecode" corresponded straightforwardly to BASIC tokens, but some of them were significantly swizzled around; `NEXT B0` compiles to `10111 CCCVVV VVVVVV F CCCVVV CCCVVV AAAAAAAAAAA` where CCCVVV encodes the increment 1, VVVVVV encodes B0, F encodes +, the second CCCVVV encodes the ending counter value (255, say), the third CCCVVV encodes the starting counter value (0, say), and the final AAAAAAAAAAA encodes the address of the statement to jump to (the first statement inside the loop). This is 41 bits, plus 16 in the

FOR token, for a total of 57 bits per loop.

How big is, uh, functionality? The original MacOS Finder was 46KiB and fit on a 400KiB floppy with MacOS ("System") and an application and a few documents, making it possible to use the Macintosh with a single floppy, though drawing on the 64KiB of ROM including LisaGraf/QuickDraw (the 128KiB ROM didn't arrive until the Mac Plus?); MacPaint was under 50 KB and consisted of 5804 lines of Pascal and 2738 lines of assembly, but that includes blank lines and comments and excludes MyHeapAsm.a and MyTools.a. A more reliable figure is 4688 lines of Pascal and 2176 lines of assembly, which is according to David A. Wheeler's "SLOCCount". If we figure that a line of Pascal is about 5 lines of assembly, according to the folklore.org figures, that's the equivalent of 6351 lines of Pascal, and under 7.9 bytes of compiled code per line; or, using SLOCCount figures, 5123 Pascal-equivalent lines and under 9.8 bytes of compiled code per line.

If you could get 4 bytecode bytes per Pascal-level line of code, which seems plausible but difficult, you could fit MacPaint into about 20 KB. The same 2.5× compression would reduce the Mac ROM to 26 KB.

There's a nonlinear benefit to this kind of thing, too: the functionality of software comes more from interactions among the components than from individual components. If you can fit 12000 lines of code into your Arduino, there are nine times as many potential pairwise interactions as if you can only fit 4000 lines of code in. At least potentially, it's an order-of-magnitude increase in system functionality.

To facilitate programmability

So, the primary objective is to fit more functionality into less memory. But a secondary objective is to improve "in-application programmability" (that is, reprogrammability without popping the microcontroller out of its ciruit and into a PROM burner) and thus flexibility.

Such a bytecode can help overcome several different obstacles to in-application programmability.

One-time-programmable devices like the PMS150C can only have code burned into them once; any further changes requires either making that code do different things based on what's in RAM, or replacing the chip with a fresh, unburned chip (3.18¢ according to file `minimal-cost-computer.md` in Derctuo — but taking advantage of that potentially involves desoldering and/or waiting for a package in the mail). It has 1024 words of PROM but only 64 bytes of RAM, but like the AVR and the GD32VF, it's a Harvard-architecture device, so it can't execute native code from RAM. Only some kind of interpreter can make it programmable.

Moreover, RAM is often small, but loading code into RAM is easier than loading code into program memory, when that is possible at all. The PMS150C's 64 bytes of RAM is enough for about 15 or 20 lines of C using a compact bytecode, but would be only about 5 lines of C using native code. (And in practice you probably only have about 48 of those bytes, since you need a few for stack and variables.)

The 82¢ ATTiny1614 I mentioned earlier has 16384 bytes of Flash but only 2048 bytes of RAM; 2048 bytes is enough for 500—800 lines of C.

Finally, compiling to bytecode can be considerably easier than compiling to native code, which makes it more feasible to include a compiler in the device itself. Its input format doesn't necessarily have to be C; it might be something like ladder logic, scientific-calculator formulas, keyboard macros, Lisp, or Forth.

How?

Probably the right thing to do, to be able to run existing microcontroller-friendly code, which is mostly written in C, C++, or the Arduino dialect of C++, is to write a C compiler to some kind of very compact bytecode, along with a bytecode interpreter for it.

A virtual machine designed for C probably needs a frame-pointer register, but otherwise a registerless stack architecture will probably result in more compact code. A lot of the usual C compiler optimizations aren't useful on a stack machine, though dead-code elimination, constant folding, and common subexpression elimination may be valuable.

There's a tradeoff between interpreter size and code size. In the MakeCode case, they evidently had about 9–10 KiB left for STS user code, and by using 500 bytes of it for the interpreter they were effectively able to increase the functionality that fit in memory by almost a factor of 5. But this is probably far from the optimum; if by doubling the size of the interpreter they were able to increase the code density from 12.3 to 10 bytes per statement, then instead of 800 statements, they would be able to fit 930 statements into the same space. How much further could you take that approach?

So it's quite plausible that the right tradeoff for minimum space is to use the *majority* of program memory for a really elaborate interpreter. But we can't do this simply by adding common subroutines as built-in interpreter instructions written in native code; we can certainly vector some built-in interpreter instructions to common subroutines, but if we want to use minimum space, we should still encode that common subroutine in bytecode, not native code. (For the most part, anyway — for example, while you could implement multiplication in terms of addition and bit tests, for example, invoking a multiply instruction is going to be a lot simpler, and similarly for most functionality commonly provided as part of the CPU instruction set.) So what should we put in the interpreter proper? Other than optimizations, of course.

Probably a big part of the answer is "weird addressing modes".

For really tiny chips like the PMS150C with its 1024 instructions of ROM, probably only 512-768 instructions or so of interpreter are affordable; obviously 1024 instructions would not be. So I think it's worthwhile to think about what kind of kernel bytecode interpreter you could fit inside that kind of constraint, even on an 8-bit machine, before decorating it with richer functionality for machines with larger memory.

Elisp and Smalltalk bytecodes commonly have an operand field

within the byte, 3–5 bits of operand. Often these operands are indexes into tables (of constants, selectors, or functions) that are associated with the method or function, and which can contain more data than the bytecode itself.

I tend to think that variable-bit-length instructions like the BASIC Stamp are hard to justify, particularly on machines like the AVR where you apparently need to write a 5-iteration loop to shift a 16-bit word by 5 bits. But variable-byte-length instructions are probably easy to justify, where perhaps you have 3 bits of operand within the instruction byte, perhaps another operand byte following, and perhaps, with a different opcode, two operand bytes following.

Making a global table of all subroutines is probably a win. Talking to solrize about the subject, they made the observation that if a subroutine is called zero times, it can be omitted from the compiled program; if it is called once, it can be inlined; and if it is called twice or more, then it is quite plausibly a savings to put its address into a global table of subroutines, and refer to it elsewhere only with an index into this subroutine table. For example, on the ATMega328 in the Arduino Uno, a word-aligned address anywhere in the 32-KiB program memory requires 14 bits, but if you can only fit 800 lines of user code in there, you can't plausibly have more than 160 user subroutines, so you only need 8 bits for a subroutine-table index. Or 7.32 bits, really. So by putting a 16-bit address into the subroutine table, you can invoke the subroutine with an 8-bit operand instead of a 14-bit operand, which probably saves you the 2 bytes that it cost to put it into the subroutine table. If there are more than 2 calls to it, you're in the black.

Exceptions might include where there was one reference to the subroutine, but you were passing its address rather than invoking it, so it can't be inlined; or where the alternative to a global table of subroutines is a local table (like the ones Smalltalk and Elisp use) that can be indexed with fewer bits, like 3–5 instead of 7.32.

If a subroutine table is too large for a compact operand field to index into it, the subroutines can be sorted so that the ones with the most callsites come first, and the losers who drew the long addresses can't be invoked without a 2-byte calling sequence.

We can improve in density over Elisp bytecode to the extent that we can exclude K&R C's generic variadic calling convention, where the caller knows how many parameters are passed but the callee doesn't, and every function returns a value — because we don't have to encode the number of arguments at the callsite, and we don't have to explicitly discard void return values. A function header can specify how many arguments there are, automatically moving them from the operand stack into the function's stack frame.

C is pretty thorough about having byte-oriented memory. IIRC you can do pointer arithmetic between nested struct members. You're not supposed to do pointer arithmetic between different things in the same stack frame. On the other hand, you could very reasonably allocate word-sized local variables in a separate area, even if you take their addresses.

A sketch with a low-level bytecode

Let's look at some examples of what bytecode-compiled subroutines might look like. My examples so far, with a strawman bytecode I made up as I went along, are respectively 3.4, 3.1, 7.5, 3.4, 2.4, and 2.3 bytes per source line of code. I think this is a compelling argument that 4 bytes per line of code is usually achievable and 3 bytes might be.

DrawTxScrap

Here's a Pascal subroutine from MacPaint; C and this Pascal are almost identical.

```
{$S       }
PROCEDURE DrawTxScrap(dstRect: Rect);
VAR i:          INTEGER;
    myProcs:    QDProcs;
BEGIN
  KillMask;   { not enough room in heap for mask and font }
  ClipRect(dstRect);
  EraseRect(dstRect);
  InsetRect(dstRect,2,0);
  dstRect.right := Max(dstRect.right,dstRect.left + txMinWidth);
  dstRect.bottom := Max(dstRect.bottom,dstRect.top + txMinHeight);
  SetStdProcs(myProcs);
  thePort^.grafProcs := @myProcs;
  myProcs.textProc := @PatchText;
  i := CharWidth('A');  { swap in font }
  HLock(txScrapHndl);
  TextBox(txScrapHndl^,txScrapSize,dstRect,textJust);
  HUnlock(txScrapHndl);
  ClipRect(pageRect);
  thePort^.grafProcs := Nil;   { restore to normal }
END;
```

What would this ideally look like in bytecode?

This only has three local variables, but two of them are structs (records in Pascal). `dstRect` evidently has `.top`, `.bottom`, `.left`, and `.right` members, which are probably integers, and it's evidently being modified by `InsetRect`, which I guess must take it as a var parameter. The `@` operator, an Apple extension, takes the address of a variable like `&` in C, just as passing a var parameter does implicitly. `thePort` is a global variable imported from elsewhere; `txMinHeight`, `txMinWidth`, `txScrapHndl`, `textJust`, and `pageRect` are five of the 128 global variables in this file; and `PatchText` is a procedure defined just above.

The repeated use of the same `txScrapHndl` and `thePort` globals makes me think that maybe a function could have an associated vector of referenced global variable indices so that such repeated references only cost one byte.

Maybe in idealized bytecode assembly this would look something like this:

```
; This next line compiles to a procedure header structure with bit
; fields.  This procedure takes 8 bytes of "blob arguments" and
; has another 18 bytes of local blobs immediately after them; both
; of these are in the blob part of the activation record.  It also
```

```
; takes 1 word of word (register-sized) arguments, which go into
; the word part of the activation record.
PROCEDURE DrawTxScrap argblob=8 localblob=18 localwords=1
globals txScrapHndl, thePort
    call KillMask
    lea_blob 0       ; load address of offset 0 into stack frame blobs
    call ClipRect
    lea_blob 0
    call EraseRect
    ;      InsetRect(dstRect,2,0);
    lea_blob 0
    tinylit 2        ; push constant 2 (coded in a 3-bit field in the bytecode)
    tinylit 0
    call InsetRect
    ;       dstRect.right := Max(dstRect.right,dstRect.left + txMinWidth);
    loadword_blob 1  ; load word from stack frame blobs at offset 1 word (.right)
    loadword_blob 0  ; load .left
    loadglobal txMinWidth
    add
    max
    storeword_blob 1 ; store max result into .right
    ;       dstRect.bottom := Max(dstRect.bottom,dstRect.top + txMinHeight);
    loadword_blob 3  ; load .bottom
    loadword_blob 2  ; load .top
    loadglobal txMinHeight
    add
    max
    storeword_blob 3
    lea_blob 8          ; load @myProcs (@myProcs)
    call SetStdProcs
    lea_blob 8          ; load @myProcs (@myProcs)
    loadglobal thePort

    storeword_offset 5  ; using thePort value from top of stack, offset 5 words and store result
    ;       myProcs.textProc := @PatchText;
    loadfuncptr PatchText
    lea_blob 8
    storeword_offset 3  ; let's say .textProc is the fourth word of myProcs

    ;      i := CharWidth('A');  { swap in font } (oddly this return value i is not used, but let's compile it anyway)
    lit8 65             ; 'A'
    call CharWidth

    storeword 0         ; store into i, in the stack frame word-sized variables (rather than blobs)
    loadglobal txScrapHndl
    call HLock
    ;      TextBox(txScrapHndl^,txScrapSize,dstRect,textJust);
    loadglobal txScrapHndl  ; not sure if this is a var parameter, I'll assume so
    loadglobal_long txScrapSize  ; this variable is 32 bits
    lea_blob 0
    loadglobal textJust
    call TextBox
```

```
    loadglobal txScrapHndl
    call HUnlock

    lea_global pageRect     ; again, assuming this is a var parameter; if not we
must memcpy
    call ClipRect
    ;       thePort^.grafProcs := Nil;   { restore to normal }
    tinylit 0
    loadglobal thePort
    storeword_offset 0
    ret
```

That's 47 bytecode instructions, so 47 opcode bytes; how many operand bytes? All but 5 of them have immediate operands, but probably none of those operands need to be more than 1 byte, so we have at most 39 operand bytes. 10 of them are call instructions (or 12 if we count the `max` instances); there are 211 functions and procedures declared in this file, including EXTERNAL procedures, and each of them is only called once in this function, so plausibly those 10 call operations do need an operand byte to index into a global table of subroutines. Another 10 operands are global variables; of these half (thePort twice and and txScrapHndl three times) are referenced more than once and could thus be usefully listed in the function's global-references vector so they could be referenced in a single byte; the other 5 would require a separate operand byte. loadfuncptr requires a separate operand byte, too. Of the other 21 operands, most are small integers between -3 and 3 or between 0 and 6, so they could be packed into a 3-bit immediate field; the only exceptions are 8, 8, 8, and 65, so there would be 4 more bytes of operands, and 17 bytecodes with embedded operands.

So that's 4 numeric operand bytes, 5 global-index operand bytes, 10 function-call operand bytes, and one loadfuncptr operand byte, for 20 operand bytes, and 44+20 = 64 bytes of bytecode. The procedure header is probably 2 bytes, the procedure's address in the global subroutine table is another 2 bytes, and then you have two global-variable offset bytes, so all in all the subroutine is probably about 64 + 2 + 2 + 2 = 70 bytes. This corresponds to 20 non-comment non-blank source lines of Pascal code, or 3.5 bytes per line, which is about 2.8 times the density of the original MacOS compiled program.

All this is assuming that there are few enough bytecode operations to fit into a single byte. The above code already uses `call`, `lea_blob`, `tinylit`, `loadword_blob`, `loadglobal`, `add`, `max`, `storeword_blob`, `lit8`, `storeword`, `storeword_offset`, and `ret`, which is 12, to 9 of which we are imputing this 3-bit operand field (let's call these heavyweight opcodes "baryonic"); furthermore supersymmetry implies the existence of such undiscovered massive particles as `storeglobal`, `loadbyte_blob`, `storebyte_blob`, `loadword_offset`, `storebyte_offset`, and `loadbyte_offset`, which bring us to 15 of the 32 major-opcode slots filled, plus leptons such as `subtract`, `multiply`, `divide`, `mod`, `bitand`, `bitor`, `xor`. So I think it's pretty plausible that we'll have plenty of opcode space, but it's something to watch.

(If we wanted the bytecodes to be *printable ASCII* things might get

more difficult: from 32 to 128 we only have room for 11 baryons and 8 leptons, or 10 baryons and 16 leptons, with one of the baryons missing the DEL character. But that's probably too strict a restriction; text in ISO-8859-1 (24 baryons instead of 12) or Windows CP1252 (most of 26, though maybe the A0 control characters would be best as leptons due to the holes) would be more reasonable, and conceivably UTF-8 would work well if operand bytes were trailing 10xx xxxx bytes. Cyrillic and Greek codepages have many more homoglyphs.)

It's a little bit unclear how blob parameters are supposed to get passed here. Do we pass them in word-sized chunks on the operand stack, or is there a separate blob stack or something? If we assume that the bytecode interpreter is equipped to go look at the function header in memory when it's interpreting a call, it might be reasonable to put the `call` bytecode *before* the arguments so that the interpreter can allocate the callee's stack frame, allowing arguments to be poked directly into it in the appropriate places instead of needing to be copied there on subroutine entry.

strlcpy

Here's `strlcpy`, originally from OpenBSD, but this version is the copy from avr-libc, for which SLOCCount reports 26 lines of code, condensed down to 14 lines for ease of reading:

```
size_t strlcpy (char *dst, const char *src, size_t siz) {
        register char *d = dst;
        register const char *s = src;
        register size_t n = siz;

        if (n != 0 && --n != 0) {
                do { if ((*d++ = *s++) == 0) break; } while (--n != 0);
        }

        if (n == 0) {
                if (siz != 0) *d = '\0';
                while (*s++)
                        ;
        }

        return(s - src - 1);
}
```

Unlike the straight-line Pascal code above, this has a bunch of control flow, seven conditional jumps. (Normally a `while` would also involve an unconditional jump, but in this case the body is empty.)

This has six local variables, all word-sized, but only five of them are live at once; `dst` passes the torch to `d` early on, which can be eliminated by the compiler.

If we try encoding it in the same bytecode as before with a fairly traditional compilation strategy, I think it looks like this:

```
PROCEDURE strlcpy argwords=3 localwords=2
    loadword 1        ; load argument 1, src
    storeword 3       ; store into word variable 3, s
    loadword 2        ; load siz (argument 2)
```

```
    storeword 5      ; n
    jz 5, 1f         ; jump to label 1 if word variable 5 is 0
    decrword 5       ; decrement word variable 5
    jz 5, 1f
 2: loadword 0       ; d, preparation for *d++
    incrword 0       ; d++
    loadbyte_offset 0 ; dereference pre-incremented pointer

    dup              ; this will be used in an assignment whose result value is u
sed
    loadword 3       ; s
    incrword 3
    storebyte_offset 0
    jztos 1f
    decrword 5       ; --n
    jnz 5, 2b        ; repeat do loop if word variable 5 still isn't 0
 1: jnz 5, 1f        ; if (n == 0)
    jz 2, 3f         ; if (siz != 0) using word variable 2
    tinylit 0        ; '\0'
    loadword 0       ; d
    storebyte_offset 0 ; *d =
 3: loadword 3       ; s
    incrword 3       ; ++
    loadbyte_offset 0
    jnztos 3b
 1: loadword 3       ; s
    loadword 1       ; src
    subtract
    tinylit 1
    subtract
    ret
```

That's 32 bytecode instructions. 4 of these are zero-operand leptons (dup, subtract, subtract, ret), 7 are conditional jumps (6 with 2 arguments), and the other 21 are the same kind of one-operand one-byte operations that dominated DrawTxScrap. Maybe `jnz 5, 2b` gets encoded with 5 in the 3-bit immediate field, indicating that it's looking at local register-sized variable 5, and jumping to label 2, looking backwards, if it is Not Zero. The byte offset to label 2 is encoded in a following operand byte, within the range ±127; if it's -128 then two more operand bytes follow giving the real jump offset. `jnztos` jumps if the top of stack is 0 instead of if a local variable is zero; I think the way to do this is that if the 3-bit immediate field is 0 through 6, it tests that variable, but if it's 7, it tests the top of stack (and pops it). By contrast, in the other baryonic operations, I was thinking 7 would indicate that the immediate parameter is in the following byte, but if you want to test a local variable that's higher than 6, then you could just `loadword` it and then `jztos`.

So, in addition to demonstrating the previously speculative `loadbyte_offset` and `storebyte_offset` operations, this imposes new baryonic opcodes on us: `incrword`, `decrword`, `jz`, `jnz`, and probably `js`, `jns`, and `jmp`, bringing the total from 15 to 22 out of 32.

So with the above-suggested encodings, we have 7 jump-offset argument bytes, 2 bytes of procedure header, and 2 bytes of global

subroutine table entry, for a total of 32+7+2+2 = 43 bytes. That's 1.7 bytes of bytecode per SLOCCount line or 3.1 bytes per physical non-blank line in the above.

You could very plausibly squeeze this down a bit more. The 8086's LOOP instruction provides the decrword/jnz functionality, its LODSB instruction provides the `loadword/incrword/loadbyte_offset` functionality, and its STOSB instruction provides `loadword/incrword/storebyte_offset`. And of course the final expression being returned admits all sorts of optimizations, including a decrement-top-of-stack operation or an increment or decrement to one of the variables being subtracted.

In an alternative direction, note that this code has 7 jump instructions, comprising 14 bytes, but only 4 labels; following Henry Baker's COMFY-65, if we could use a 1-byte instruction to set the destination to jump to on failure or success, maybe we'd only need 4 bytes of jump destinations instead of 7.

`proc_get_cam_register_3`

Let's look at something a little less nice and pretty: `proc_get_cam_register_3` from `linux-2.6/drivers/staging/rtl8192e/rtl_debug.c`:

```
static int proc_get_cam_register_3(char *page, char **start,
                                   off_t offset, int count,
                                   int *eof, void *data)
{
        struct net_device *dev = data;
        u32 target_command = 0;
        u32 target_content = 0;
        u8 entry_i = 0;
        u32 ulStatus;
        int len = 0;
        int i = 100, j = 0;

        /* This dump the current register page */
        len += snprintf(page + len, count - len,
                        "\n#################### SECURITY CAM (22-31) ######"
                        "############\n ");
        for (j = 22; j < TOTAL_CAM_ENTRY; j++) {
                len += snprintf(page + len, count - len, "\nD:  %2x > ", j);
                for (entry_i = 0; entry_i < CAM_CONTENT_COUNT; entry_i++) {
                        target_command = entry_i + CAM_CONTENT_COUNT * j;
                        target_command = target_command | BIT31;

                        while ((i--) >= 0) {
                                ulStatus = read_nic_dword(dev, RWCAM);
                                if (ulStatus & BIT31)
                                        continue;
                                else
                                        break;
                        }
                        write_nic_dword(dev, RWCAM, target_command);
                        target_content = read_nic_dword(dev, RCAM0);
                        len += snprintf(page + len, count - len, "%8.8x ",
                                        target_content);
```

```
                }
        }

        len += snprintf(page + len, count - len, "\n");
        *eof = 1;
        return len;
}
```

This is 36 lines of code, which I think is maybe a little too much for a sketch, so I'll try just the first half of it, plus enough cleanup to get it to compile, which brings it to 22 lines:

```
static int proc_get_cam_register_3(char *page, char **start,
                          off_t offset, int count,
                          int *eof, void *data)
{
        struct net_device *dev = data;
        u32 target_command = 0;
        u32 target_content = 0;
        u8 entry_i = 0;
        u32 ulStatus;
        int len = 0;
        int i = 100, j = 0;
        len += snprintf(page + len, count - len,
                        "\n#################### SECURITY CAM (22-31) ######"
                        "############\n ");
        for (j = 22; j < TOTAL_CAM_ENTRY; j++) {
                len += snprintf(page + len, count - len, "\nD:  %2x > ", j);
                for (entry_i = 0; entry_i < CAM_CONTENT_COUNT; entry_i++) {
                        target_command = entry_i + CAM_CONTENT_COUNT * j;
                }
        }
        return len;
}
```

Let's say our C is a 16-bit-pointer platform, like Arduino, so the `u32` items all go into blobland instead of wordland. As before, I'll elide the copy from `data` to `dev`. And I'll assume that the interpreter by default initializes all local variables to zero, which is a reasonable thing for a C implementation to do.

```
PROCEDURE proc_get_cam_register_3 argwords=6 localwords=4 localblob=16
const "\n#################### SECURITY CAM (22-31) ##################\n "
const "\nD:  %2x > "
    lit8 100

    storeword 8  ; i.  note that this implies a separate operand byte with 3-bit
immediate fields
    loadword 0   ; page *
    loadword 7   ; len
    add
    loadword 3   ; count *
    loadword 7
    subtract
    loadconst 0  ; the string *
```

```
    call snprintf
    loadword 7
    add         ; len +=
    storeword 7
    ; for (j = 22; j < TOTAL_CAM_ENTRY; j++) {
    lit8 22
    storeword 9
1:  loadword 9
    lit8 32      ; TOTAL_CAM_ENTRY
    subtract
    jstos 1f     ; if result negative, skip loop
    incrword 9
    loadword 0   ; *
    loadword 7
    add
    loadword 3   ; *
    loadword 7
    subtract
    loadconst 1  ; the other string *
    call snprintf
    loadword 7
    add
    storeword 7
    tinylit 0    ; *
    storebyte_blob 0  ; entry_i = 0.  In the blob because u8 *
2:  loadbyte_blob 0   ; *
    lit8 8            ; CAM_CONTENT_COUNT
    subtract
    jstos 1f
    loadbyte_blob 0   ; *
    tinylit 1         ; *
    add
    storebyte_blob 0  ; *
    ; target_command = entry_i + CAM_CONTENT_COUNT * j;
    loadbyte_blob 0   ; *
    lit8 8            ; CAM_CONTENT_COUNT
    loadword 9        ; j
    multiply
    storelong_blob 0  ; target_command = *
    jmp 2b
1:  loadword 7
    ret
```

This is a mess! But a workable mess. 49 bytecode instructions. Of these, 11 have no operands at all; another 14, marked with * above, have operands that can be packed into a 3-bit field; the remaining 24 each need an operand byte. 73 bytes of bytecode! But by itself that would be pretty okay for 21 lines of code (3.8 bytes per line). What really kills us here is the 64-byte literal string and the other 12-byte literal string, plus a constant vector (probably two 16-bit pointers). All that, plus the 2-byte procedure header and 2-byte entry in the global subroutine table, adds up to $73 + 64 + 12 + 4 + 2 + 2 = 157$ bytes. That's 7.5 bytes per line of code! Those two string literals *doubled* the space this truncated subroutine needs.

I think that if I were to add in the missing 15 lines of code, this would get slightly less ugly, maybe another 64 bytes, which would get the total down to about 4.4 bytes per line of code. But it would be easy for someone to write code that really does have that many quoted # signs in it.

Of course, this *particular* code probably has no business running on a microcontroller, even if I hadn't snipped out the `read_nic_dword` call that does the real work; it's part of a Linux device driver for a Wi-Fi card that I think requires a PCI bus to operate at all. But I think it's a good representative of workaday C code.

File_pipe2file

This is from `php5-5.4.4/ext/fileinfo/libmagic/compress.c`:

```
protected int
file_pipe2file(struct magic_set *ms, int fd, const void *startbuf,
    size_t nbytes)
{
        char buf[4096];
        ssize_t r;
        int tfd;
#ifdef HAVE_MKSTEMP
        int te;
#endif

        (void)strlcpy(buf, "/tmp/file.XXXXXX", sizeof buf);
#ifndef HAVE_MKSTEMP
        {
                char *ptr = mktemp(buf);
                tfd = open(ptr, O_RDWR|O_TRUNC|O_EXCL|O_CREAT, 0600);
                r = errno;
                (void)unlink(ptr);
                errno = r;
        }
#else
        tfd = mkstemp(buf);
        te = errno;
        (void)unlink(buf);
        errno = te;
#endif
        if (tfd == -1) {
                file_error(ms, errno,
                    "cannot create temporary file for pipe copy");
                return -1;
        }
```

It... goes on for another 36 lines from there; that's 30 lines. It's not super plausible that we could fit the PHP interpreter into an Arduino, but we could surely fit it into an Ambiq Apollo3. The `protected` suggests that this is not actually C at all but C++, but it's pretty close. Let's see what this would look like in the strawman bytecode assembly. Let's imagine we *do* `HAVE_MKSTEMP`.

```
PROCEDURE file_pipe2file argwords=4 localwords=3 localblob=4096
```

```
const "/tmp/file.XXXXXX"
const "cannot create temporary file for pipe copy"
const 4096
globals errno
    lea_blob 0      ; buf
    loadconst 0     ; filename template
    loadconst 2     ; 4096, sizeof buf
    call strlcpy

    drop            ; discard result, because of course that makes sense when call⚬
⚬ing strlcpy
    ; #ifndef HAVE_MKSTEMP drops the next N lines
    lea_blob 0      ; tfd = mkstemp(buf);
    call mkstemp
    storeword 6
    loadglobal errno ; te = errno
    storeword 7
    lea_blob 0
    call unlink
    drop
    loadword 7        ; errno = te
    storeglobal errno
    loadword 6
    tinylit -1
    subtract
    jnz 1f
    loadword 1
    loadglobal errno
    loadconst 1
    call file_error
    tinylit -1
    ret
1:
```

So that's 25 bytecodes, of which the four `calls` and the two references to `te` require an operand byte. The others can all reasonably be 1 byte each, so we have 31 bytes of bytecode, plus 43 + 17 = 60 bytes of literal strings, 6 bytes of constant table, 1 byte of global vector, 2 bytes of procedure header, and 2 bytes of global subroutine table entry, 31 + 60 + 6 + 1 + 2 + 2 = 102 bytes, nearly ⅔ in those two stupid strings. Still, that's 102 bytes for 30 lines of code: 3.4 bytes per line. But only because 12 of those 30 lines were discarded by the preprocessor!

Like most of the PHP interpreter, this is a good example of really pretty shitty C code that people nevertheless want to run, and run correctly.

Insertion sort

Here's an excerpt from the latest .c file I wrote in my dev3 directory, which does an insertion sort on an array of ints (obviously a programming exercise):

```
static inline void
swap (int *x, int *y)
```

```
{
  int tmp = *x;
  *x = *y;
  *y = tmp;
}

void
isort(int *a, size_t n)
{
  for (size_t i = 1; i < n; i++) {
    for (size_t j = i; j > 0; j--) {
      if (a[j-1] > a[j]) swap(&a[j], &a[j-1]);
    }
  }
}
```

This counts as 16 lines of source code, although that's pretty generous! Let's suppose our compiler does in fact inline `swap` and cancel out the resulting `*&`:

```
void
isort(int *a, size_t n)
{
  for (size_t i = 1; i < n; i++) {
    for (size_t j = i; j > 0; j--) {
      if (a[j-1] > a[j]) {
         int tmp = a[j];
         a[j] = a[j-1];
         a[j-1] = tmp;
      }
    }
  }
}
```

This is the first example we've seen that does real pointer arithmetic, the kind where you have to multiply by the size of the pointer. Using just the strawman bytecode above and no CSE I think it looks something like this:

```
PROCEDURE isort argwords=2 localwords=3
    tinylit 1
    storeword 2    ; i
1:  loadword 2
    loadword 1     ; n
    subtract
    jstop 2f
    loadword 2
    storeword 3    ; j
3:  jz 3, 4f       ; if j == 0, exit loop
    loadword 0     ; a for a[j-1]
    loadword 3     ; j-1
    tinylit 1
    subtract
    tinylit 2      ; sizeof int
    multiply
```

```
    add
    loadword_offset 0
    loadword 0      ; a[j]
    loadword 3
    tinylit 2
    multiply
    add
    loadword_offset 0
    subtract          ; if >
    js 5f
    loadword 0        ; tmp = a[j]
    loadword 3
    tinylit 2
    multiply
    add
    loadword_offset 0
    storeword 4       ; tmp
    loadword 0        ; a[j] = a[j-1]
    loadword 3
    tinylit 1
    subtract
    tinylit 2
    multiply
    add
    loadword_offset 0
    loadword 0
    loadword 3
    tinylit 2
    multiply
    add
    storeword_offset 0
    loadword 4        ; a[j-1] = tmp
    loadword 0
    loadword 3
    tinylit 1
    subtract
    tinylit 2
    multiply
    add
    storeword_offset 0
5:  decrword 3      ; j--
    jmp 3b
4:  incrword 2
    jmp 1b
2:  ret
```

This is 61 bytecode instructions, which I think is pretty horrific. 5 of them are jumps which take an operand byte, so 66 bytecode bytes in all, plus the usual 4 bytes of per-subroutine overhead, for 70 bytes. That's 4.4 bytes per line of code.

However, there are a couple of directions to go to improve this. One is that, despite the examples above, neither `a[j-1]` nor `for (...; i < expr; i++)` is actually at all unusual in C. We could support the first one with leptonic bytecodes to decrement top of stack and to do

base+index addressing:

```
loadword 0       ; a[
loadword 4       ; j
decrtos          ; -1
loadword_indexed ; ]
```

You'd also have `storeword_indexed`, `loadbyte_indexed`, `storebyte_indexed`, `leaword_indexed`, and maybe `leabyte_indexed`, though that last one is really just `add`.

And we could support the loop with a baryonic bytecode similar to the 8086 LOOP instruction mentioned earlier, but for up-counting loops; if at the bottom of the loop:

```
loadword 1       ; n
countup 2, 1b    ; i < n? if so increment and jump back to label 1
```

For the full C semantics, which loops zero times when n is 0, you'd need to initialize the counter to one *less than* the initial value and then unconditionally jump to that loop trailer; 5 bytes per loop in total. Alternatively you could reverse the sense of the conditional jump, put it at the top of the loop, and put the unconditional jump at the end of the loop.

The Forth way to handle counting loops is different; it stores the loop counters, limits, and increments on the return stack instead. This would require more analysis from the compiler (it has to verify that the loop limit is a constant expression and that the loop counter is only read from, insert an UNLOOP instruction at breaks) but it would allow you to write the outer loop with two new leptonic instructions as follows:

```
tinylit 1        ; start value
loadword 1       ; n, loop limit
fortoloop        ; stash loop counters and loop address on the loop stack
...
continue         ; update loop counter, maybe jump to address after fortoloop
```

And the inner loop as follows:

```
i                ; get outer loop counter as start value
0                ; loop limit
tinylit -1       ; step
fortosteploop
...
continue
```

These are respectively 4 bytes (32 bits) and 5 bytes (40 bits), which are significantly less than the BASIC Stamp's 57 bits despite using a byte-oriented encoding.

Instead of having Forth-like i, j, and k instructions, you could maybe stick the loop counter in a regular word-sized local variable, using a baryonic for-loop instruction. But suppose we go with the leptonic approach above:

```
PROCEDURE isort argwords=2 localwords=1
    tinylit 1
    loadword 1
    fortoloop
    i               ; outer loop counter, currently topmost
    tinylit 0
    tinylit -1
    fortosteploop
    loadword 0
    i               ; inner loop counter, called j in the code, now topmost
    decrtos
    loadword_indexed
    loadword 0
    i
    loadword_indexed
    subtract        ; if not <
    jns 1f          ; skip body
    loadword 0      ; a[j] → tmp
    i
    loadword_indexed
    storeword 2
    loadword 0      ; a[j-1] → a[j]
    i
    decrtos
    loadword 0
    i
    storeword_indexed
    loadword 2      ; tmp → a[j-1]
    loadword 0
    i
    decrtos
    storeword_indexed
1:  continue
    continue
    ret
```

That gets us down to 34 bytecode instructions and 35 bytes of bytecode, 39 bytes in all, 2.4 bytes per line of code.

Finally, you could actually do common subexpression elimination and stack-allocate the value of `tmp`:

```
PROCEDURE isort argwords=2 localwords=2
    tinylit 1
    loadword 1
    fortoloop
    i
    0
    tinylit -1
    fortosteploop
    loadword 0      ; &a[j-1]
    i
    decrtos
    leaword_indexed
    storeword 2
```

```
    loadword 0
    i
    leaword_indexed
    storeword 3     ; &a[j]
    loadword 2
    loadword_offset 0
    loadword 3
    loadword_offset 0
    subtract
    jns 1f          ; if
    loadword 3      ; swap
    loadword_offset 0
    loadword 2
    loadword_offset 0
    loadword 3
    storeword_offset 0
    loadword 2
    storeword_offset 0
1:  continue
    continue
    ret
```

That gets it down to 33 bytecode instructions, 34 bytes of bytecode, 38 bytes in all, still 2.4 bytes per line. It's surprising, though, how little space those complicated compiler optimizations actually bought us: one measly byte!

However, we did need *some* complicated compiler optimizations to use the above version.

XXX what if we use the dumber loop operations? how about if we don't inline?

Rob Kendrick mentions that DSP loop instructions are often implemented as Intercal's "COMEFROM", which seems like another approach that might be worth investigating — a for-loop instruction that encodes the length of the loop, so that instead of

```
    tinylit 1
    loadword 1
    fortoloop
    ...
    continue
    ret
```

you would write

```
    tinylit 1
    loadword 1
    fortoloop 2f
    ...
2:  ret
```

so that interpreting each instruction would require first checking to see if you had hit the address of the end of the innermost for-loop. This would improve compression over having a `continue` bytecode only

if the loop extent could be packed into the fortoloop byte, which limits us to fairly short loops.

Also, it might be worthwhile to have a special short form for for (int i = 0; i < x; i++) where x and the loop end are the only varying parameters, so the starting index doesn't eat up a bytecode. Here's a random sampling of 16 for loops in C codebases I have handy, divided into loops that fit that pattern and loops that don't:

```
9base_6.orig/rc/trap.c: while(ntrap) for(i = 0;i!=NSIG;i++) while(trap[i]){
cmusphinx/multisphinx/sphinxbase/feat.c:        for (i = 0; i < start_pad; ++i)
emacs24_24.5+1.orig/src/coding.c:  for (reg = 0; reg < 4; reg++)
exopc/lib/libtermlib/mkinfo.c:    for (i = 0; i < numcaps; i++)
flashlight-firmware/ToyKeeper/crescendo/crescendo.c:    for(i=0;i<32;i++) {
gmp-5.0.5+dfsg/mpn/generic/mu_divappr_q.c:      for (i = 0; i <= err_rec; i++)
linux-2.6/drivers/watchdog/sbc8360.c:           for (i = 0; i != count; i++) {
linux-2.6/sound/soc/soc-dapm.c:     for (i = 0; i < ARRAY_SIZE(dapm_up_seq); i++)
puzzles-20200513.c9b3c38/bridges.c:        for (x = 0; x < params->w; x++) {

exopc/bin/less/charset.c:   for (p = charsets;  p->name != NULL;  p++)
exopc/bin/less/opttbl.c:    for (o = option;  o->oletter != '\0';  o++)
exopc/bin/sh/show.c:    for (p = arg->narg.text ; *p ; p++) {
exopc/sys/ubb/dependency.c: for(e = l_first(a->out); e; e = l_next(e, out_next))
gawk_4.2.1+dfsg.orig/cint_array.c:  for (i = NHAT; i < INT32_BIT; i++) {
linux-2.6/drivers/atm/eni.c:             for (*pre = 3; *pre >= 0; (*pre)--)
maru-2.4/eval.c:    for (;;) {
```

9 of these, the majority, though barely, fit that pattern, more or less. Saving a tinylit 0 byte on half of all for loops seems like it would be worthwhile. There's substantial diversity in the termination condition — <, <=, != — though in many cases <= err_rec is equivalent to < err_rec + 1 or != err_rec + 1.

Anduril config_state_base

Let's look at some BudgetLightForum flashlight firmware, because Anduril now compiles to over 8 KiB and so won't fit in some of the smaller AVRs. Here's a random 27-line function from flashlight-firmware/ToyKeeper/spaghetti-monster/anduril/anduril.c, with internal comments removed:

```
// ask the user for a sequence of numbers, then save them and return to caller
uint8_t config_state_base(Event event, uint16_t arg,
                          uint8_t num_config_steps,
                          void (*savefunc)()) {
    static uint8_t config_step;
    if (event == EV_enter_state) {
        config_step = 0;
        set_level(0);
        return MISCHIEF_MANAGED;
    }
    else if (event == EV_tick) {
        if (config_step < num_config_steps) {
            push_state(number_entry_state, config_step + 1);
        }
        else {
```

```
            savefunc();
            save_config();
            pop_state();
        }
        return MISCHIEF_MANAGED;
    }
    else if (event == EV_reenter_state) {
        config_state_values[config_step] = number_entry_value;
        config_step ++;
        return MISCHIEF_MANAGED;
    }
    return EVENT_HANDLED;
}
```

Event is uint8_t. config_step is a "global variable" that no other functions use. EV_enter_state is (B_SYSTEM|0b00001000). B_SYSTEM is just 0, so EV_enter_state is just 8. MISCHIEF_MANAGED is EVENT_HANDLED which is 0. EV_tick is (B_SYSTEM|0b00000001) or 1. EV_reenter_state is (B_SYSTEM|0b00001010), or 10. config_state_values is an array of 3 uint8_t. number_entry_value is a volatile global uint8_t. So maybe that function would look something like this:

```
; event and num_config_steps are two argument blob bytes; arg and
; savefunc are two word args.
PROCEDURE config_state_base argwords=2 argblob=2
globals config_step
    loadbyte_blob 0      ; event
    lit8 8               ; EV_enter_state
    subtract
    jnz 1f               ; if ==
    tinylit 0
    storeglobal_byte config_step
    tinylit 0
    call set_level
    tinylit 0
    ret
1:  loadbyte_blob 0
    tinylit 1            ; EV_tick
    subtract
    jnz 3f
    loadglobal_byte config_step
    loadbyte_blob 1      ; num_config_steps
    subtract
    jns 1f               ; if <
    loadfuncptr number_entry_state
    loadglobal_byte config_step
    incrtos
    call push_state
    jmp 2f
1:  call savefunc
    call save_config
    call pop_state
2:  tinylit 0
    ret
3:  loadbyte_blob 0
```

```
    lit8 10             ; EV_reenter_state
    subtract
    jnz 1f

    loadglobal_byte number_entry_value  ; config_state_values[config_step] = numb
er_entry_value
    leaglobal config_state_values
    loadglobal_byte config_step
    add
    storeword_offset 0
    loadglobal_byte config_step
    incrtos
    storeglobal_byte config_step
    tinylit 0           ; return MISCHIEF_MANAGED
    ret
1:  tinylit 0           ; return EVENT_HANDLED
    ret
```

That's 44 bytecode instructions. 5 of these are calls, 2 are lit8s, 5 are jumps, and 1 is loadfuncptr, and those 13 instructions will need a separate operand byte. I think all the others can be single-byte instructions, so this would be 57 bytes of bytecode. There would additionally be 2 bytes of procedure header, 2 bytes of subroutine table entry, and 1 or 2 bytes of global vector (to enable the function to refer to `config_step` with a 3-bit immediate index field), for a total of 63 bytes, which is about 2.3 bytes per line of code.

(There's questions here of how this kind of thing affects the instruction set, and whether there might be opcode space contention, but I feel like there was enough headroom previously that I shouldn't worry about that.)

I asked what this function looks like in AVR machine code; solrize generously provided the following assembler output listing for a slightly different version of the code, which I've edited down for readability:

```
 4121                     config_state_base:

 4134 0fc0 8830               cpi r24,lo8(8)                 ;      config_st
ep = 0;
 4135 0fc2 01F4               brne .L348

 4138 0fc4 1092 0000          sts config_step.2502,__zero_reg__ ;    set_level
(0);
 4141 0fc8 80E0               ldi r24,0
 4142 0fca 0E94 0000          call set_level

 4145 0fce 8091 0000          lds r24,button_last_state          ;        confi
g_step ++;
 4146 0fd2 8111               cpse r24,__zero_reg__
 4147 0fd4 00C0               rjmp .L350
 4150 0fd6 81E0               ldi r24,lo8(1)

 4151 0fd8 8093 0000          sts config_step.2502,r24           ;         push_
state(number_entry_state, 0);
```

```
 4152 0fdc 00C0                 rjmp .L362
 4154 0fde 982F          .L348:  mov r25,r24
 4155 0fe0 907F                  andi r25,lo8(-16)

 4158 0fe2 903B                  cpi r25,lo8(-80)                ;       if (confi
g_step <= num_config_steps) {
 4159 0fe4 01F4                  brne .L351

 4162 0fe6 2091 0000             lds r18,config_step.2502        ;           if (2
 == (arg % (TICKS_PER_SECOND*3/2))) {
 4163 0fea 4217                  cp r20,r18
 4164 0fec 00F0                  brlo .L352

 4167 0fee CB01                  movw r24,r22                    ;               c
onfig_step ++;
 4168 0ff0 6DE5                  ldi r22,lo8(93)
 4169 0ff2 70E0                  ldi r23,0
 4170 0ff4 0E94 0000             call __udivmodhi4
 4171 0ff8 0297                  sbiw r24,2
 4172 0ffa 01F4                  brne .L353
 4175 0ffc 2F5F                  subi r18,lo8(-(1))
 4176 0ffe 2093 0000             sts config_step.2502,r18

 4179 1002 4217                  cp r20,r18                      ;
   set_level(RAMP_SIZE * 3 / 8);
 4180 1004 00F0                  brlo .L350
 4183 1006 88E3                  ldi r24,lo8(56)                 ;           }
 4184 1008 00C0                  rjmp .L360
 4188 100a 82E1          .L353:  ldi r24,lo8(18)                 ;           }
 4189 100c 00C0                  rjmp .L360
 4193 100e 80E0          .L352:  ldi r24,0                       ;       }
 4195 1010 0E94 0000     .L360:  call set_level
 4196 1014 00C0                  rjmp .L350
 4200 1016 903E          .L351:  cpi r25,lo8(-32)
 4201 1018 01F4                  brne .L355

 4204 101a 8091 0000             lds r24,config_step.2502        ;           push_
state(number_entry_state, 0);
 4205 101e 8823                  tst r24
 4206 1020 01F0                  breq .L361

 4209 1022 4817                  cp r20,r24                      ;           push_
state(number_entry_state, 0);
 4210 1024 00F0                  brlo .L361
 4214 1026 60E0          .L362:  ldi r22,0                       ;       }
 4215 1028 70E0                  ldi r23,0
 4216 102a 80E0                  ldi r24,lo8(gs(number_entry_state))
 4217 102c 90E0                  ldi r25,hi8(gs(number_entry_state))
 4218 102e 0E94 0000             call push_state
 4219 1032 00C0                  rjmp .L350
 4223 1034 8A30          .L355:  cpi r24,lo8(10)
 4224 1036 01F4                  brne .L350
 4227 1038 6091 0000             lds r22,number_entry_value
 4228 103c 8091 0000             lds r24,config_step.2502
 4229 1040 F901                  movw r30,r18
```

```
 4230 1042 0995                icall

 4233 1044 0E94 0000           call save_config              ;     pop_state⚬
⚬();
 4237 1048 0E94 0000    .L361: call pop_state                ; }
 4241 104c 80E0         .L350: ldi r24,0
 4242 104e 0895                ret
```

That's 58 instructions, and with the 16-bit immediate address arguments for instructions like `sts` and `call`, it turns out to be 144 bytes of AVR machine code, 5.3 bytes per line of code, 2.3 times the size of as the strawman bytecode above.

The Anduril repo is fairly large; it draws from the 1703 SLOCCount lines of code in its parent directory and contains 2814 SLOCCount lines itself. But the majority of these lines are `#ifdef`fed out in most configurations or are declarations or preprocessor directives. A typical configuration might have 1200 logical lines of code (with semicolons) that actually make it through the preprocessor, a quarter of which are declarations. But that still compiles to over 8 KiB of AVR code. So, although I could be mistaken, I tentatively think that this sort of bytecode trick could be useful even for existing embedded projects that have gone to substantial trouble to fit into the limitations of tiny microcontrollers.

A sketch with an AST bytecode

The above design corresponds pretty closely to machine operations, although in some cases several of them. But, for example, `loadbyte_blob 0` pushes a byte onto the stack from the beginning of blob space.

A disadvantage of this is that, in a way, the type of every variable is encoded redundantly all over the program. Every time `config_step` is loaded or stored, the `loadglobal_byte` or `storeglobal_byte` instruction repeats the fact that it's a byte. This is fast but uses up a lot of the 32 possible baryonic opcodes: you need the cross product of {lea, load, store}, {global, local}, and {word, byte}, which is 12, and at some point you need to handle longs too, which I've given short shrift to above, and maybe long longs, at which point you'd be at 24. (The load and store instructions that use computed pointers on the operand stack could be leptonic, although in the above sketch they contain a rarely-used offset in their operand field, but the global and local instructions need an operand field to specify which global or local variable they're referring to. Potentially you could supply the long (or short, on a 32-bit system) and long long versions only in leptonic form so they always take a pointer from the stack.)

But you could imagine storing the fact that `config_step` is a byte just once, in the `config_step` object, and just having `loadglobal` consult that information. Moreover, you could even eliminate the distinct types of `storeword_offset` etc. if the pointers on the stack carried a data type with them (which could be overridden by an explicit cast operation).

In a sense, your virtual machine ends up being dynamically typed, purely in order to reduce the number of distinct opcodes; it mirrors the structure of the C source code, where type information is confined to variable declarations rather than replicated across all uses

of the variables. Aside from potentially making more efficient use of opcode space (and thus getting a more compact bytecode) such dynamic typing might be beneficial in two other ways: it might be friendlier for interactive use, and it might allow library functions to be polymorphic, so you could just invoke a linear-search function rather than writing a linear-search loop.

Self took this approach further; because its bytecode was intended for JIT-compiling rather than direct interpretation, it was very nearly just an abstract syntax tree of the Self program. Could this provide a more compact representation? Consider my isort example above:

```
void
isort(int *a, size_t n)
{
  for (size_t i = 1; i < n; i++) {
    for (size_t j = i; j > 0; j--) {
      if (a[j-1] > a[j]) {
         int tmp = a[j];
         a[j] = a[j-1];
         a[j-1] = tmp;
      }
    }
  }
}
```

As an S-expression, its abstract syntax tree might look like this:

```
(function isort ((pointer int) unsigned) ; argument types, words 0 and 1
                (unsigned unsigned int)  ; local types, words 2, 3, 4
   (forto (word 2) (const 1) (word 1)    ; i is word 2, n is word 1
      (fortostep (word 3) (word 2) (const 0) (const -1)
         (if (> (aref (word 0) (- (word 3) (const 1))) (aref (word 0) (word 3)))
            (progn
               (setf (word 4) (aref (word 0) (word 3)))

               (setf (aref (word 0) (word 3)) (aref (word 0) (- (word 3) (const ↩
↪1))))
               (setf (aref (word 0) (- (word 3) (const 1))) (word 4)))))))
```

Most of the node types here have a fixed arity; the exceptions are `progn` and the function arguments and locals. So if we wanted to build up this tree in RPN, we could use a setf operation that pops two expressions off the stack and pushes a setf node, and so on, but for `progn` we'd need some kind of marker. A dumb way of writing this down might be:

```
isort:
    [         ; begin list
    int
    pointer   ; make pointer of int
    unsigned
    ]         ; end argument type list
    [         ; begin local variable type list
    unsigned
```

```
unsigned
int
]
word 2   ; (for) i =
const 1  ; 1 to
word 1   ; n
word 3   ; (for) j =
word 2   ; i to
const 0  ; 0 step
const -1 ; -1
word 0   ; a[
word 3
const 1
-
aref     ; ]
word 0   ; a[
word 3
aref     ; ]
>
[        ; progn
word 4   ; tmp =
word 0   ; a[
word 3   ; j
aref     ; ]
setf
word 0
word 3
aref
word 0
word 3
const 1
-
aref
setf
word 0
word 3
const 1
-
aref
word 4
setf
]progn
if
fortostep
forto
```

By my count, this is 52 AST-node building operations, each quite plausibly encodable in a byte; `const 1` corresponds rather closely to `tinylit 1`, `word 3` to `loadword 3`, etc. It's pretty hard to read, as stack-machine code often is, because the context that gives meaning to the for-loop stuff at the beginning isn't provided until half a page later. The type-building operations could conceivably use a separate encoding from the expression-building operations like `aref` and `-`, which could conceivably use a separate encoding from the

statement-building operations like `if`, `forto`, and `]progn`, but that would probably also require a REBOL-like or LOGO-like non-reverse Polish notation approach. That might look like this. The `.` token represents a byte that terminates a variable-length thing like a progn; `word.2` represents a byte with `2` packed into the low 3 bits.

```
function isort pointer int unsigned .
               unsigned unsigned int .
   forto word.2 const.1 word.1
      fortostep word.3 word.2 const.0 const.-1
         if > aref word.0 - word.3 const.1  aref word.0 word.3
            progn
                setf word.4 aref word.0 word.3
                setf aref word.0 word.3  aref word.0 - word.3 const.1
                setf aref word.0 - word.3 const.1  word.4
            .
```

(I'm not suggesting that writing in this format would be a good way to program, just trying to write out the contents of the sketched bytecode in a way that's easy to run through `wc` and easy to see if I left something out of.)

These 52 tokens are almost the same tokens as before, but in a different order. My earlier lower-level sketch of this using fortoloop and fortosteploop and no CSE was more compact for the following reasons:

- It doesn't have 8 or more bytes of procedure header.
- It doesn't specify where to store the loop counters; instead they are stored on a loop counter stack, saving 2 bytes.
- Instead of saying `- word.3 const.1` it uses `decrtos`, saving 3 bytes.
- Instead of `if >` it uses `subtract; jns`, but that's a wash.
- Instead of `setf aref`, it uses `storeword_indexed`, and instead of `setf word`, it uses `storeword`, saving 3 bytes.
- Instead of using `progn ... .` to delimit the innermost block, it uses `continue; continue; ret`, which ought to use one *more* byte.

So it was 34 bytes, and 34 + 8 + 2 + 3 + 3 - 1 = 51, which I think is everything but the "function isort" at the beginning.

If we apply the applicable improvements from the above, we get:

```
function isort pointer int unsigned .
               int .
   forto const.1 word.1
      fortostep i const.0 const.-1
         if > aref word.0 1- i  aref word.0 i
            progn
                setword.4         aref word.0 i
                aset word.0 i     aref word.0 1- i
                aset word.0 1- i  word.4
            .
```

This gets us down to, say, 40 bytes, which is still worse than the low-level bytecode version.

It's a little annoying that `aref` needs to be explicit; in Forth you can

define and use an array-creating word as follows:

```
: array create cells allot does> swap cells + ;  ok
5 array myarray  ok
57 0 myarray !  68 1 myarray !  ok
0 myarray ?  1 myarray ? 57 68  ok
```

Here we have defined a 5-entry array called `myarray` and stored 57 in its entry 0 and 68 in its entry 1. No explicit `aref` is needed, but this being Forth, an explicit `?` or `@` is needed to fetch from it in rvalue context anyway! But we could imagine avoiding that in an AST format designed for C, somehow.

However, aside from being (it seems) bulkier than the low-level approach, this structure seems like it would be much more difficult to interpret.

Notes on Tanenbaum 01978

Tanenbaum and his students tackled this idea in 01978, with the idea of implementing the interpreter as microcode and making their computer faster and cheaper. Since they couldn't measure how fast unimplemented microcode was, they settled for measuring how small their programs were, which was under the streetlight and is vaguely related. The paper is sometimes cited as inspiring RISC, due to its emphasis on measuring the frequencies of real operations in real programs, but the actual "EM-1" instruction set they arrived at is about as far from RISC as it is possible to be.

Where RISCs have many general-purpose registers, EM-1 has none, not even a single accumulator, instead using an operand stack (which is also the call stack and storage for globals). Where RISCs have a single instruction width, EM-1 has instructions of 1, 2, 3, and 4 bytes. Where RISC instructions typically have 3 operand fields, EM-1 instructions have 0 or 1. Where RISCs have carefully laid out bit fields to minimize instruction decoding latency, EM-1 declares, "There is no need to have distinct "opcode" and "address" bits." Where RISCs are, like nearly all CPUs, untyped, and strive to keep all instructions constant-time so that they can be implemented without microcode, EM-1 has hardware support for bounds-checked Numpy-like arbitrary-dimensional array descriptors and for "accessing intermediate lexicographical levels in block structured languages", which latter seems to be defined handwavily and buggily, and involves a single load or store instruction following an arbitrarily long linked list through RAM.

However, the EM-1 *is* RISC-like in that it's roughly a load-store machine: ALU operations operate strictly on the stack, not even including the add-immediate instruction, which is included even in the very frugal RV32E, except that the EM-1 has 14 increment instructions, including one for TOS. And, like the EM-1, the Berkeley RISC-I and -II (and their demon offspring SPARC) are designed around reducing the cost of procedure call and return.

The strawman bytecode outlined earlier is remarkably similar to the EM-1! The biggest differences are:

• the EM-1 is a single-stack design.
• the EM-1 is substantially less aggressive about packing operands into single-byte instructions.
• the EM-1 is not designed to be able to execute C, which makes demands on pointer arithmetic that the EM-1's array descriptors are ill-suited to fulfill.
• the EM-1 apparently entirely lacks structs or records.

The EM-1's instruction set consists of the following:

• 12 one-byte pushlocal opcode bytes ("all the locals in 94.6% of procedures");
• 8 one-byte pushglobal opcodes;
• 3 one-byte pushconst opcodes (0, 1, and 2);
• 12 one-byte poplocal opcodes;
• 8 one-byte popglobal opcodes;
• 4 two-byte {push,pop}{global,local} opcodes;
• 2 two-byte pushconst opcodes for numerical constants in [-256, 255];
• 4 one-byte ALU opcodes (+, -, ×, ÷);
• 24 one-byte opcodes and 2 two-byte opcodes for zeroing and incrementing locals;
• 4 two-byte local and global array-access opcodes (pushelement, popelement);
• 1 two-byte lealocal opcode "for call-by-reference";
• 1 two-byte dereference opcode, for the same reason;
• 3 one-byte "mark" opcodes for starting to set up a parameter list for a subroutine call, which increment, decrement, or leave unchanged the static nesting levels (for nested subroutines);
• 1 two-byte subroutine-call opcode for programs containing up to 256 subroutines, though they suggest that in most cases the special context following a "mark" operation would allow some 200 opcodes to context-dependently specify which subroutine to call;
• 1 two-byte opcode for allocating a stack frame, which is odd since the destination of the call instruction is a "procedure descriptor" and not a raw code address;
• 2 three-byte opcodes for FOR (one counting up and one down; the second byte tells which variable to use, and the third gives the branch offset), though you additionally need an unconditional jump back at the end of the loop;
• 14 (?) two-byte opcodes for conditional branches;
• 141 unallocated opcode bytes;
• lots of other three- and four-byte instructions whose first byte is 255; they propose "accessing intermediate lexicographical levels", "multiple precision arithmetic, floating point, shifting, rotating, Boolean operations, etc."

I seem to have omitted 10 opcode bytes somewhere. Unfortunately they didn't include an opcode table.

Discussion of records (structs) is, bizarrely, completely lacking; in the EM-1 design it seems to be impossible to heap-allocate linked list nodes. Perhaps multidimensional arrays were the only data-structuring mechanism contemplated, but 01978 is about 10 years too late for such an omission. Even arrays were accessed via "descriptors" in the stack frame, suggesting that the EM-1's stack

frames were considered to be homogeneous vectors of machine words.

There are lots of good ideas in this paper. They mention that 95% of FOR loops have steps of +1 or -1, so maybe it's best to support those two with special for-loop instructions, and relegate other steps to compilation as `while` loops. Their conditional branches combine testing with jumping, like RISC-V, for example, and they propose an assembler that sorts local variables by number of references.

Topics

- History (p. 1153) (24 notes)
- Performance (p. 1155) (22 notes)
- Lisp (p. 1174) (11 notes)
- Assembly-language programming (p. 1175) (11 notes)
- Compilers (p. 1178) (10 notes)
- Virtual machines (p. 1182) (9 notes)
- C (p. 1194) (8 notes)
- Microcontrollers (p. 1211) (6 notes)
- Instruction sets (p. 1214) (6 notes)
- FORTH (p. 1231) (5 notes)
- Bytecode (p. 1236) (5 notes)
- Reverse Polish notation (RPN) (p. 1243) (4 notes)
- Pascal (p. 1247) (4 notes)
- Sorting (p. 1272) (3 notes)
- AVR8 microcontrollers (p. 1387) (2 notes)
- Arduino (p. 1388) (2 notes)
- Ambiq (p. 1391) (2 notes)

Wiki models

Kragen Javier Sitaker, 02021-08-19 (updated 02021-12-30) (1 minute)

I've been thinking about how to build simple models.

Suppose I write the equations:

```
area = height * section.perimeter + 2 * section.area
volume = height * section.area
```

These describe a cylinder or prism, and they imply some things about `section`: it should have a property `area` that can be multiplied by an integer or whatever `height` is, and a property `perimeter` that can be multiplied by whatever `height` is and then added to an integer times `area`.

My editor should offer to create `section` and `height`, and in `section` it should offer to create `perimeter` and `area`.

Topics

- Human-computer interaction (p. 1156) (22 notes)
- End user programming (p. 1217) (6 notes)
- Numerical modeling (p. 1229) (5 notes)
- Editors (p. 1257) (4 notes)
- Wiki (p. 1311) (2 notes)
- Calculation

Residual stream windowing

Kragen Javier Sitaker, 02021-08-21 (updated 02021-09-11)
(5 minutes)

Suppose you want to transmit video data for a windowing system, but you want something that's simpler and less CPU-hungry than H.264 encoding. Could you simply transmit a stream of lossily-encoded residuals?

PNG doesn't really compress image data, necessarily; what it compresses, like a lot of lossy audio and video codecs, are the prediction residuals from some pixel predictor (called "filters" and defined in §6.1 of the PNG spec), an approach Paeth calls "prediction-correction coding," a term which has not caught on. These predictors are specified on a per-scanline basis:

- 0: predict 0, so the residual is the original pixel value, and we *are* compressing the original image data;
- 1: predict the pixel to the left, or zero if it's the first pixel on the line, so the residuals are the pairwise differences between pixels on the line;
- 2: predict the corresponding pixel in the line above, or 0 on the first line, so the residuals are the pairwise differences between pixels in a column;
- 3: predict the average of prediction #1 and prediction #2, rounded down;
- 4: predict the Paeth predictor, a choice of one of the three pixels above, to the left, and above and to the left, depending on a simple linear calculation on those three pixels that estimates the local gradient. This is a slight variation on the predictor Paeth published in "Image File Compression Made Easy", chapter 9 of *Graphics Gems II*; he calls the gradient estimation calculation the "poor man's Laplacian".

The Paeth predictor is almost never much worse than any of the previous four alternatives, but often it's slightly worse, and you can of course construct examples where it's arbitrarily worse; you could get a simpler graphics file format by using the Paeth predictor alone, as Paeth does in his chapter, and it would compress only a little worse than PNG. In fact, for small enough images, it might compress better, because PNG specifies the predictor to use at the beginning of each scan line, which adds extra data to compress.

Now, in PNG, these residuals are compressed *losslessly*, so the previous decompressed pixels input to the decompressor's predictor are exactly equal to the original image input pixels. So the benefit for PNG is primarily that things like continuous gradients become repeating patterns.

A potentially much more exciting application is the residual-coding approach used for lossy video encoding, where instead of compressing the exact residual sequence, we compress a crude, low-bandwidth approximation of the residual sequence. This means that the decompressor's predictor is working from pixels corrupted by the approximation noise, so to get the system to work optimally, we need

to include the lossy-encoding step inside the feedback loop of the compressor's predictor, so that the "context pixels" it's predicting from are the same corrupted pixels the decompressor will see, rather than the original image pixels. This ensures that the errors will not accumulate nonlinearly.

In the context of encoding a video stream, particularly of a windowing system, it would be useful to use the previous frame, or *a* previous frame, as context as well. If a screen region is unchanged from the previous frame, the predictor will predict its contents perfectly, so if there was no error in the previous frame, all the residuals will be zeroes, which will compress very well. If it's unchanged from the previous frame, but the previous frame was corrupted by lossy compression, then the residuals that are transmitted will reduce the error.

You could conceivably have 9 neighboring pixels in the previous frame and 4 neighboring pixels in the current frame to use as a "neighborhood" for your prediction. (Normal video codecs also use motion compensation, but continuous motion is less common in GUIs (though modern GUIs use it a bit more).) You could use those 13 values to compute a least-squares approximation of the four-dimensional gradient and average, or just 7 of them (like Paeth's 3), or you could use any four of them (the symmetrical choice would be the corresponding pixels in the previous row, column, and frame, and the diagonal pixel in the previous frame, but that doesn't give you any information about whether pixels are changing frame to frame). Moreover, you could very reasonably use a measure of the neighborhood diversity to choose whether to do Paeth's trick of choosing one of them or to use the gradient extrapolation in the continuous domain.

Once you have your prediction, you need to code your residual lossily. I think doing this in YUV space is probably perceptually desirable, because it allows you to trade off more Y bits against less UV bits. In many frames, you could use literally zero UV bits for many of the pixels, as long as you eventually send some UV bits for them so that they will converge to the correct color.

Topics

• Algorithms (p. 1163) (14 notes)
• Protocols (p. 1206) (6 notes)
• Compression (p. 1263) (4 notes)
• Video (p. 1312) (2 notes)
• The Paeth predictor (p. 1345) (2 notes)

Sandwich panel optimization

Kragen Javier Sitaker, 02021-08-21 (updated 02021-09-11)
(3 minutes)

Wikipedia says the stiffness of a sandwich panel is $\frac{1}{2}(f(2h + f)^2C)$ where C is, I think, the stiffness (Young's modulus) of the face materials, f is the thickness of the face, and h is half the thickness of the foam (or similar filling). This is $\frac{1}{2}C(4fh^2 + 4f^2h + f^3)$, which is roughly $2Cfh^2$ if h is big enough relative to f. If you want to maximize stiffness for a given (areal) density, then $\rho_0 f + \rho_1 h$ is some constant total density k, half our total density budget (here ρ_0 is the density of the face material and ρ_1 the density of the filling), so $f = k/\rho_0 - (\rho_1/\rho_0)h$, and the stiffness is $2C(k/\rho_0 - (\rho_1/\rho_0)h)h^2$. To find the maximum we can drop out the constant $2C/\rho_0$ and set the derivative with respect to h to zero; differentiating $(k - \rho_1 h)h^2$, getting $-\rho_1 h^2 + (k - \rho_1 h)\cdot 2h = -\rho_1 h^2 + 2hk - 2\rho_1 h^2 = 2hk - 3\rho_1 h^2$. This derivative obviously has a zero at h=0, where the stiffness is also zero, and the other extremum will be when $0 = 2k - 3\rho_1 h$, $2k = 3\rho_1 h$, $h = 2k/3\rho_1 = \frac{2}{3}k/\rho_1$.

I must be doing something wrong. This says the filling should just be ⅔ of the total mass, regardless of the density or modulus of the face material. Stiffness, fine, that's just a constant linear factor on the stiffness you get at any point along the tradeoff spectrum. But shouldn't it depend on the density of the face material? No, because higher density just gives you proportionally less face material and thus less stiffness, so it's also just a constant linear factor.

And I guess this is actually correct. If ⅓ of your mass is in the faces, then making the faces 2% thicker makes the panel 2% stiffer, but steals 1% of the mass from the filling, making the panel 1% thinner. Since stiffness is quadratic in thickness, that 1% thinning reduces the stiffness by 2% (actually 1.99%), and the resulting stiffness is only 99.9702% of the original thickness. A similar thing happens if you make the faces 2% thinner.

Interestingly, this generalizes to properties other than density that scale with the volume of the material as well, in particular cost. If you want to maximize sandwich panel stiffness with given materials at a given cost, you should have ⅓ of the cost in the faces, ⅔ in the filling. This is applicable to the roofing problem in Leaf vein roof (p. 600).

Topics

- Filled systems (p. 1161) (16 notes)
- Strength of materials (p. 1164) (13 notes)
- Foam (p. 1185) (9 notes)
- Composites (p. 1187) (9 notes)

Glass wood

Kragen Javier Sitaker, 02021-08-21 (updated 02021-12-30)
(4 minutes)

I was making some waterglass foam today, just by heating up some liquid waterglass on aluminum foil in an aluminum-foil-covered steel bowl over a fire, and it occurred to me that maybe it's possible to get the most crucial aspects of the structure of wood in a glass-fiber composite in a very simple way.

Specifically, I'm thinking that you ought to be able to soak a glass-fiber tow or stack of unidirectional cloths in waterglass, then bake it, just as I did without the fiber today. The waterglass will attack the glass-fiber tow at high temperatures, but I think the whole process can be done at a low enough temperature to keep that to a minimum. The result should or might be a glass composite material consisting of a lightweight foam holding together a lot of parallel fibers.

There are a number of potentially interesting aspects of this kind of composite, but the primary one that interests me is the combination of load-bearing capability with ease of mechanical shaping. In general, materials that can bear a lot of load, such as diamond, tungsten carbide, sapphire, quartz, steel, are also hard to shape mechanically. There are at least two ways to escape from this: post-shaping hardening (like concrete or heat treatments of steel, or casting), and foams.

Natural wood is a foam, with continuous parallel cellulose fibers connected together with lignin glue, plus lots of empty space. The empty space has several big advantages: it makes wood much easier to cut; it makes the wood much stiffer, like a sandwich panel; it inhibits crack propagation from one fiber to another, dramatically improving impact strength, because cracks perpendicular to fibers must propagate along a jagged zigzag path ("splintering"), greatly increasing the energy required; and it allows impacts to plastically deform the material without damaging fibers. (Bamboo is also a fiber-foam composite, but with a different structure; see DOI 10.1007/s00226-007-0127-8.)

Other solid foams, such as refractory firebrick, waterglass foam, styrofoam, and polyurethane cushions, are also much easier to shape than the corresponding bulk material would have been. Earlier tonight I cut through waterglass foam with a box cutter; I could push the unsupported blade all the way through 15 mm or more of foamed waterglass, corresponding to about 1 mm of solid glass, only because the particles displaced by the knife could move aside into the voids in the foam.

Ceramic-matrix composites get their improvement in impact strength by recruiting a longer section of crack-bridging fibers to elastically resist crack-widening movements than the section actually within the crack; this is enabled by weakening the bonding between the matrix and the fiber reinforcement. This seems closely analogous to the splintering behavior of natural wood, but I'm not sure it's quite

the same.

So my thesis is that maybe a composite of glass foam and glass fibers will be lightweight, rigid, easy to cut, and impact-resistant.

Alternative or supplementary fiber reinforcements might include steel, basalt, copper, carborundum, or carbon fiber.

If the foaming of the glass is to happen rapidly and uniformly, it would be helpful for the heat to be applied by a reaction within the mixture (so-called “self-propagating high-temperature synthesis”), but it is probably crucial that the direction of propagation of this reaction be at right angles to the fiber direction; if it is in the direction of the fibers it will not only stretch them but also kink them. For this purpose it might be helpful to lay the reagents into the foam in the form of “fibers” parallel to the structural fibers, so that the reaction can propagate very rapidly along each “fiber” but much more slowly from one “fiber” to another; to make these reagent “fibers” hollow; and to apply the initial ignition simultaneously along the whole length of the material. SHS could perhaps be usefully applied to higher-temperature glass foaming reactions which might yield stronger and more water-resistant foams than waterglass; the conventional one seems to be reacting manganese dioxide with carbon in a matrix of soda-lime glass, but many others are possible.

Topics

• Filled systems (p. 1161) (16 notes)
• Strength of materials (p. 1164) (13 notes)
• Foam (p. 1185) (9 notes)
• Composites (p. 1187) (9 notes)
• Anisotropic fillers (p. 1218) (6 notes)
• Self-propagating high-temperature synthesis (SHS) (p. 1241) (4 notes)
• Ceramic-matrix composites (CMCs) (p. 1265) (4 notes)

Maximizing phosphate density from aqueous reaction

Kragen Javier Sitaker, 02021-08-21 (updated 02021-12-30)
(8 minutes)

If you want to form phosphates of calcium, magnesium, or aluminum with a reaction between a finely divided solid and a liquid, for maximum strength, it would be desirable for a maximum amount of the reaction mass to be incorporated into the final phosphate product, and a minimal amount to be lost as waste products. But you'd also like the reaction product to not be too much larger than the original finely divided solid.

What would the ideal materials be?

Phosphate sources

The phosphate anion alone has a molar mass of 94.9714 g/mol.

Monoammonium phosphate has a molar mass of 115.025 g/mol (containing one phosphate, so it's 83% phosphate by weight), weighs 1.80 g/cc, dissolves 36g in 100ml of water at 20°, and dissolves 173g in 100ml of water at 100°. So one ml of solid MAP contains 1.49 g of phosphate. It decomposes at 200°.

Diammonium phosphate is 132.06 g/mol (72% phosphate) and 1.619 g/cc, dissolves 57.5g/100ml at 10°, and decomposes at 155°. So one ml of solid DAP contains 1.16 g of phosphate.

Phosphoric acid, of course, has a molar mass of 97.994 g/mol and also contains one phosphate, so it's 97% phosphate. Its density is 1.834 g/cc when solid. It dissolves 548 g per 100ml of water at 20°, and melts at 40–42.4°, so above that temperature no water is needed at all to make it liquid; but dehydrating it can be very difficult. One ml of solid phosphoric acid contains 1.78 g of phosphate.

Trisodium phosphate is 163.939 g/mol (58% phosphate), has a density of 2.536 g/cc (when not hydrated!), and dissolves 12g/100ml of water at 20° or 94.6g/100ml at 100°. So one ml of solid TSP contains 1.47 g of phosphate.

Disodium phosphate is 141.96/mol (67% phosphate) and 1.7 g/cc and dissolves 7.7 g/100ml at 20°, so one cc of it contains 1.14 g of phosphate.

Calcium sources

Calcium weighs 40.078 g/mol.

Calcium chloride is 110.98 g/mol (36% calcium) and weighs 2.15 g/cc (0.8 g Ca/cc), and water dissolves 74.5 g/100ml at 20°. It's also soluble in acetic acid, ethanol, methanol, pyridine, all kinds of crazy stuff. It's a sticky pain in the ass to dry out, though.

Slaked lime is 74.093 g/mol (54% calcium) and 2.211 g/cc, so each cc contains 1.2 g of calcium. At 20° water only dissolves 0.173 g/100ml of it, but apparently it's soluble in glycerol?

Quicklime is 56.0774 g/mol (71% calcium) and 3.34 g/cc (2.4 g Ca/cc) but reacts with water rather than dissolving in it.

Calcium nitrate is 164.088 g/mol (24% calcium) and 2.504 g/cc, so each cc contains only 0.6 g of calcium. Water dissolves 121 g/100ml at 20° or 271g/100ml at 40°.

Calcium acetate is 158.166 g/mol when anhydrous (25% calcium) and 1.509 g/cc (0.4 g Ca/cc) but very hygroscopic, dissolving 34.7 g/100ml in water at 20°.

Calcium formate is 130.113 g/mol (31% calcium) and 2.02 g/cc (0.6 g Ca/cc) and water dissolves 16 g/100ml at 0°. It decomposes at 300°.

Aluminum sources

Aluminum itself weighs 26.9815384 g/cc, and is a candidate source for aluminum ions. However, it ordinarily resists attack by phosphoric acid reasonably well.

Aluminum trihydroxide is 78.00 g/mol (35% Al) and 2.42 g/cc (0.8 g Al/cc). It starts releasing its hydroxyls at 300°. Water only dissolves 0.1 mg/100ml of it.

Sodium aluminate is 81.97 g/mol (33% Al) and 1.5 g/cc. Its advantage over the hydroxide is that it's highly water-soluble, especially at high temperature and pH.

(Di)aluminum (tri)sulfate (papermaker's alum) is one of the standard water-soluble aluminum salts; it weighs 342.15 g/mol (15.8% Al) when anhydrous, and 2.672 g/cc. Water dissolves 36.4g/100ml of it at 20°, making it acidic.

Aluminum (tri)chloride is the other, even more soluble one; it weighs 133.341 g/mol (20% Al) (when anhydrous) and 2.48 g/cc (0.5 g Al/cc). Water dissolves 45.8 g/100ml at 20°. Interestingly, the anhydrous form sublimes at 180°, suggesting the possibility of *gassing* a mixture with aluminum. However, you cannot dehydrate the hexahydrate by heating it, and the anhydrous compound fumes in moist air! You have to form the anhydrous form anhydrously, perhaps via SHS from copper chloride an aluminum metal.

There are also soluble acetates of aluminum.

Aluminum *oxide* weighs 101.960 g/mol (53% aluminum) and can be extremely stable, or it can be relatively reactive, depending on how it's been treated since it was formed. It's totally insoluble in anything.

Phosphate sinks

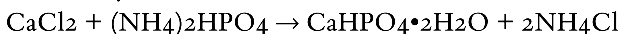
Tricalcium (di)phosphate, one possible end goal, is 310.18 g/mol (containing two phosphates, so 61% phosphate by weight, the rest being calcium) and weighs 3.14 g/cc. Its solubility in water is 1.2 mg/kg, which I guess is 0.12 mg/100ml.

Monocalcium (di)phosphate, another, is 234.05 g/mol when unhydrated (81% phosphate) but tends to disproportionate into phosphoric acid and "dicalcium" (di)phosphate, especially above 203°. It's dramatically more water-soluble than the tricalcium salt, at 2 g/100ml. Soluble acid monobasic phosphates like this may be useful

not only as phosphate sinks but also as phosphate sources to calm down the reaction with things like calcium oxide, which might otherwise be too violent.

Dicalcium (mono)phosphate, a third, is actually $CaHPO_4$, 136.06 g/mol (70% phosphate). WP says:

In a continuous process $CaCl_2$ can be treated with [diammonium phosphate] to form the dihydrate:



$CaCl_2 + (NH_4)_2HPO_4 \rightarrow CaHPO_4 \cdot 2H_2O + 2NH_4Cl$

A slurry of the dihydrate is then heated to around 65–70 °C to form anhydrous $CaHPO_4$ as a crystalline precipitate, typically as flat diamondoid crystals, which are suitable for further processing.

There's also a "dicalcium diphosphate", to which it converts when heated to 240°-500°, which is actually calcium pyrophosphate.

Aluminum orthophosphate ($AlPO_4$) weighs 121.9529 g/mol and 2.566 g/cc, thus being 78% phosphate and 22% aluminum, and containing 2.0 g/cc of phosphate and 0.57 g/cc of aluminum.

Monoaluminum phosphate is commonly used as a refractory cement since Kingery's dissertation, but unfortunately is not in Wikipedia.

Fillers

The idea of all this stuff is to form a lasting solid precipitate, especially for 3-D printing applications. There are several possible problems to solve.

The first is reactions that are too slow and allow the materials to escape.

The second is reactions that complete adequately fast, but do not produce enough solid product to form a solid mass.

The third is reactions that are too *fast* and throw the materials apart rather than producing a solid product.

The fourth is that the phosphate material, even when fully formed, may not be very strong, or may have other undesirable properties such as being electrically insulating.

Inert fillers should help with all four of these problems. The filler particles impede diffusion and convection, allowing more time for the reaction to complete. The new crystals need only bridge the existing filler particles together rather than interlocking with other new crystals. The filler particles add a great deal of thermal mass and reduce the temperature rise associated with any exothermic reaction, allowing even very exothermic reactions to take place safely in the interstices. And they can contribute many, though not all, desired properties to the finished product.

Kingery's dissertation used kaolin and quartz sand as fillers for his various refractory "bonds" such as monoaluminum phosphate.

Wet-ground vermiculite may be an expedient functional filler for this purpose, since vermiculite is easy to find and wet grinding is easy to do. I suspect that talc and clays are probably superior for most purposes.

Other promising fillers include apatite, berlinite, synthetic aluminum phosphates, sapphire, quartz sand, mullite, glass fiber,

basalt fiber, carbon fiber, recycled glass, glass microspheres, graphite, carbon black, carborundum, and zeolites.

Topics

- Materials (p. 1138) (59 notes)
- Filled systems (p. 1161) (16 notes)
- Aluminum (p. 1180) (10 notes)
- Phosphates (p. 1184) (9 notes)
- Composites (p. 1187) (9 notes)
- Vermiculite (p. 1238) (4 notes)
- Kingery, the father of modern ceramics (p. 1288) (3 notes)

Lazy heapsort

Kragen Javier Sitaker, 02021-08-22 (updated 02021-09-11)
(6 minutes)

Heapsort's initial heapification phase is linear time, while its sorting phase is linearithmic. One common reason for sorting things is to see the first N items; heapsort, unlike quicksort, mergesort, or library sort, can produce those top N items much sooner than it can produce the rest of the sorted results.

Heapsort's laziness

I wrote a simple heapsort program that generates random integers and heapsorts them. For ten items, it takes 4–7 swaps to initially heapify them, then 21–25 swaps to finish sorting; for a hundred, 64–74 swaps and 500–520 swaps respectively; for a thousand, 700–740 and 8300–8400; for ten thousand, 7400 and 120 thousand; for a hundred thousand, 74 thousand and 1.5 million; for a million, 740 thousand and 18 million; for ten million, 7.4 million and 220 million; for a hundred million, 74 million and 2.5 billion. The heapifying phase does about 4.7 record compares per swap, so about 3.5 per input record.

So, even with very small input sets, heapsort can generate the first output value in only 20% of the time required to generate the whole output set (though admittedly at this scale insertion sort is probably faster), and for reasonable-sized results, the difference is more than an order of magnitude, with the heapifying phase going as low as 3% of the total. Moreover, it takes only about 3.5 comparisons per input record, plus a logarithmic number per output record.

This is relevant if you're writing a sort utility that generates output lazily, as when the shell `sort` command is piped to some other command. This laziness-friendliness seems like a relevant attribute for a generic standard-library sorting routine or toolkit: by heapifying a data array into a max-heap and then performing a few extract-max operations, we have a top-N algorithm.

If you know at the outset how many output values you're going to need, you can do better than this by iterating over the input values, conditionally adding them to a "shortlist" heap of the right size, from which future values may possibly evict them. In the case where most values never make it onto the shortlist, it's easy to keep the number of comparisons per input record below 1.2, but of course it cannot go below 1.

Can we improve heapsort's locality of reference? An idea that fails

Normally heapsort has relatively poor locality of reference, making it unusable for external sorting. This has probably already been investigated, but I think this can be cured by dividing the heap into smaller heaps connected by queues.

Suppose you have 256 GiB of 16-byte records (16 gibirecords) to

sort in 16 GiB of RAM (1 gibirecord). Your auxiliary storage is a tebibyte SSD, which takes 100 μs to do a 4096-byte read or write, potentially containing 256 records.

One way to approach this problem is to build a bunch of 255-record min-heaps, each associated with a 255-record queue containing records that precede its top (minimal) record.

First, consider the output phase: we repeatedly consume a record from the root queue. When a queue goes empty, we refill it from the heap dangling off of it by repeatedly removing minimal items from that heap and appending them to the queue. This usually involves sifting up items that we would normally find in child heaps, but in this case the child heaps are at the other end of their own queues, so unless one of those queues goes empty, we only need the root queue, the root heap, and its 255 (?) child queues in RAM, a total of 2056 kibibytes.

However, every item added to the output queue shifts an item into the root heap from one of those child queues, which has a 1/255 chance of going empty. At that point, we need to refill that queue, so we temporarily switch to refilling that queue by draining 255 items from its heap, each of which has a 1/255 chance of emptying one of *its* child queues, unless it's a leaf node. So on average we will recurse all the way down to the leaves when we empty the root's output queue, but only once; sometimes we'll get lucky and end the recursion early, and sometimes we'll get unlucky and have to recurse down to the leaves two or three times.

Draining a leaf node in this way only requires 4 KiB of RAM buffer instead of 2056 KiB. (Its empty queue was already in RAM because its parent node was draining it.)

Each heap+queue node holds 512 records, so we need 32 mebinodes; with 255-way branching, these are about 3e-6% the root node, 0.0008% its children, 0.2% the third level, 49.4% the fourth level, and 50.4% the fifth level. The first three levels (and 6% of the fourth level) fit in RAM, so every time we drain the root queue and recurse down to our on-average-one-leaf, we're paging in 50.4% of the time a fourth-level 2052-kibibyte node with all its child queues, plus one of the fifth-level leaf nodes. So we need, I think, 515 iops, 51.5 milliseconds, every other time we drain the root queue, which is about 1 iops per record, which is... still unusably slow. If I've calculated this correctly, it'll take us 20 days to sort our data file this way.

By contrast, we can trivially mergesort the data file in two passes: one to divide it into (worst-case) 16 16-gibibyte internally-sorted hunks, and a second pass doing a 16-way merge. That's a little less than 4 hours.

Topics

- Programming (p. 1141) (49 notes)
- Performance (p. 1155) (22 notes)
- Algorithms (p. 1163) (14 notes)

- Real time (p. 1195) (7 notes)
- Sorting (p. 1272) (3 notes)
- Latency (p. 1358) (2 notes)

Recursive bearings

Kragen Javier Sitaker, 02021-08-23 (updated 02021-12-30) (1 minute)

ABEC-7 608 roller-skate bearings cost 20¢ in the US, 40¢ here in Argentina, but are only rated for like 1–2 kN of static load. SKF rates their deep-groove 608 bearings at 1.37 kN static load. So what if you need more than twice that?

Well, you could support your shaft between three rollers at each end, and support each of the six rollers on two bearings. That ought to give you twice the load capacity: 4×1.37 kN. Also, it ought to give you lower friction if the shaft is smaller than the rollers and the rolling friction between the shaft and the rollers doesn't itself contribute an overwhelming amount of friction.

If that isn't sufficient, you can repeat the process recursively, by mounting a roller on your new shaft and making six such assemblies to support another shaft, containing 36 rollers of the smaller size and 72 skate bearings.

This is silly, though, because at the point that you're spending US$15–30 on bearings, you might as well just buy a bigger bearing.

Topics

- Contrivances (p. 1143) (45 notes)
- Pricing (p. 1147) (35 notes)
- Mechanical (p. 1159) (17 notes)

A construction set using SHS

Kragen Javier Sitaker, 02021-08-24 (updated 02021-09-11)
(5 minutes)

I was thinking earlier about "voxel 3-D printing" with spot-welded bearing balls, where each new ball added to the workpiece is located by contacts with three existing balls, then spot-welded to each of them in turn. That way, the end effector doesn't need to have extremely precise location abilities, because the location is in the precision of the bearing balls, which can easily and very inexpensively be submicron.

However, spot-welding them will both induce stresses on the structure as the nugget expands and then contracts, and also reduces the precision of the distance between the centers of the balls; the bigger the spot welds, the bigger this effect. And with any practical size of spot weld, the resulting structure will be much weaker than . Also, it requires high power input from the end effector, even though it's electrical power.

The standard mechanical-engineering solution to locating and fastening parts accurately is to use locator pins (or other features) separate from the fasteners, such as screws. By chamfering or tapering locator features, it becomes possible to assemble parts with greater precision than the precision of the manipulators, in the same way that the balls in my first paragraph would provide high precision. Somewhat analogously, the position of a lathe saddle can be indicated much more precisely by a dial indicator (or, nowadays, a digital readout) than by the graduations on the handwheel, because (aside from backlash) the kinematic chain of the dial indicator does not have to bear the load of feeding the tool into the work.

The wedging action of a tusk-tenon joint makes such a permanent joint for a somewhat related reason: the load on the joint is orthogonal to the direction of movement of the wedge, so it does not tend to dislodge it.

So I think that perhaps the best way to assemble things for a permanent, rigid mechanical connection is:

- Position them in a precise place using positioning features such as a Maxwell kinematic coupling.
- Hold them in that place using a fastening system that can handle all the variations in position that the positioning system can produce, without producing large enough loads during the holding operation to create positioning errors. For example, two parallel plates sliding against one another are a planar joint, with three degrees of freedom, until one or more screws through oversize holes in one into tapped holes in the other add enough friction to prevent movement. A spherical ball-and-socket joint also has three degrees of freedom until enough friction is similarly added. With a serial kinematic chain of three joints (of two or three degrees of freedom), you can provide all six degrees of freedom; putting them close together and putting more than one such chain in parallel can provide greater rigidity. (There might be a way to do it with just two joints, but I can't see it.)

• Lock the holding/fastening mechanism with something adequately permanent, like self-propagating high-temperature synthesis to fuse parts together, some safety lockwire, a jam nut, or just a circlip or similar spring. The loads will be borne by the holding/fastening mechanism, not by the positioning mechanism or by the locking mechanism, because the locking mechanism only serves to prevent the fastening mechanism from coming unfastened.

These three functions are not always so independent; in a four-jaw lathe chuck, for example, each jaw fulfills both the positioning function (when the other jaw is far away) and the holding function (when it's adding pressure to the part and thus friction to both the part and the other jaw). But I think separating them will generally improve precision. At times, in a lathe chuck, the moving function in two rotational degrees of freedom is provided by tapping the workpiece up against the flat face of the chuck, before holding the part in place by tightening the jaws.

You could perhaps drench the balls in a viscous liquid that later forms a glass, which can perhaps later be annealed into a glass-ceramic, so that they are positioned by the precise Hertzian contact between the balls, but then held in place by the glass or glass-ceramic matrix. This will work best if the matrix is nearly as hard as the balls (in the sense of Young's modulus) or even harder. In effect, the matrix foam is the real object; the balls are just there to provide it with precise dimensions, and they could be hollow bubbles or even removed entirely after the matrix hardens. Hollow fused-quartz bubbles would probably be especially useful for this purpose.

Topics

• Materials (p. 1138) (59 notes)
• Digital fabrication (p. 1149) (31 notes)
• 3-D printing (p. 1160) (17 notes)
• Composability (p. 1188) (9 notes)
• Self-propagating high-temperature synthesis (SHS) (p. 1241) (4 notes)
• Fasteners (p. 1297) (3 notes)

Sorption vacuum pumps really can’t operate continuously

Kragen Javier Sitaker, 02021-08-24 (updated 02021-09-11)
(5 minutes)

Wikipedia tells me that sorption vacuum pumps cannot operate continuously. I think that this might be incorrect, even though a continuously-operating sorption pump may not have been built yet. However, initial simulation results are not promising.

Consider a very thin pipe divided into a series of stages, which I will label A, B, and C:

```
A B C A B C
===========
```

We alternately cool and heat these stages. Initially let us suppose that all the stages are cold.

In the first phase let us heat the A stages, desorbing their contents; the leftmost A stage ejects half of its contents to the left, out the tube, and half to the right, where they are sorbed by the B phase. The other A stages eject their contents similarly, but into the C stages to their left and the B stages to their right.

In the second phase, let us heat the B stages. The leftmost B stage, similarly, ejects half of its contents into the leftmost A stage, whence they continue out the end of the pipe, and half of its contents into the C stage to its right. (Under some assumptions, these proportions are less even; perhaps some gas molecules make their way back to B from A and proceed on to C. We will come back to that.) The other B stages eject half their contents into the C stages to *their* right, and the other half into the A stages to their left, whence they continue to the following C stage to the left. So now all our gas is sorbed in the C stages, except for the part that escaped out of the left side initially.

In the third phase, we cool the A stages again. Because all the gas in the pipe is already sorbed in the C stages, they do not sorb any gas, except for the leftmost A stage, which sorbs gas that was previously to the left end of the pipe.

In the fourth phase, we finally heat the C stages. The rightmost C stage expels half of its contents to the right end of the pipe, while the others expel them evenly into the A stages to their left and right.

In the fifth phase, we cool the B stages back down, which has no effect on the gas distribution, because all of the gas is sorbed in the A stages.

In the sixth phase, we heat the A stages again, as in the first phase. This expels half the contents of the leftmost A stage, which it sorbed in the third phase, into the B stage to its right, and half back out the end of the pipe.

In the seventh phase, we cool the C stages back down, which results in sorbing some gas from the right end of the pipe.

The eighth phase is a return to the second phase, heating the B stages, but now with a different distribution of gases. In particular, half the gas sorbed from the leftmost A stage, which took it from outside the pipe, is ejected back out the pipe, while the other half (one quarter of the total initially sorbed) moves to the C stage to its right. The cycle repeats from there.

My thought is that, although gas diffuses in both directions through the tube, it diffuses faster to the right.

In the following simulation, however, this does not work at all:

```
def sim(output):
    gas = [1.0] * 20
    a = [i for i in range(1, len(gas)-1) if i % 3 == 1]
    b = [i for i in range(1, len(gas)-1) if i % 3 == 2]
    c = [i for i in range(1, len(gas)-1) if i % 3 == 0]
    temp = [0.0] * len(gas)
    temp[0] = temp[-1] = 3.0
    for t in range(1000):
        output(gas, temp)
        f = t // 5 % 6 + 1
        if f in [1, 3, 5]:
            for i in (c if f == 1 else a if f == 3 else b):
                temp[i] = 0.0
        else:
            for i in (b if f == 2 else c if f == 4 else a):
                temp[i] = 7.0

        deltas = [0.0] * len(gas)
        for i in range(len(temp)):
            for neighbor in [i-1, i+1]:
                if neighbor < 0 or neighbor >= len(gas):
                    continue
                d = temp[i] * gas[i] * 0.05
                deltas[neighbor] += d
                deltas[i] -= d

        for i in range(len(temp)):
            gas[i] += deltas[i]
```

In the simulation, the phases are applied correctly (from the second phase, anyway) but the net gas movement is zero.

Topics

- Contrivances (p. 1143) (45 notes)
- Physics (p. 1157) (18 notes)
- Python (p. 1166) (12 notes)
- Vacuum

Electrodeposition welding

Kragen Javier Sitaker, 02021-08-25 (updated 02021-09-11)
(2 minutes)

It's relatively straightforward and doesn't take much force to bend a shape precisely out of wire, for example copper or steel wire; this can even be a complex three-dimensional shape, although you have to take into account springback and work-hardening. However, unless the wire is very thick (which makes it harder to bend) the resulting object is fairly weak unless the wires are then twisted together, which may not always be feasible. If you weld adjacent wires together, you can get a fairly solid structure, but this both distorts the overall structure as the welds cool and introduces surface positioning error locally due to surface tension.

Electrodeposition is an interesting alternative to welding in this context. It's a terrible alternative to welding in the usual contexts, because material deposition at any given position is very slow, and you get almost no penetration into joints. But in the case of a wireframe structure, the "joints" may initially have only grazing contact or no contact at all, and it's only necessary to deposit a thickness of material comparable to the thickness of the original wire to get a fully-strong structure. Even much smaller thicknesses of deposition may provide adequate strength for many applications.

Not only does the process introduce little or no side loading or distortion, but submerging the workpiece in water actually reduces the side loading resulting from the weight of the wire, though this is not very significant if the wire is a dense material like copper.

As Wolf Hilbertz discovered in 01976, cathodic deposition is significant even in seawater, where it results not in electroplating but in the precipitation of a hydromagnesite-aragonite stone due to the hydroxyls formed at the cathode; apparently he got about 5 cm/year (1.6 nm/s) with under 4 volts, though I vaguely remember the Mother Earth News article recommending 12 volts. If you get to choose the electrolyte you can do electrodeposition of metals much faster than that; presumably that's true of these carbonate minerals as well. See Fast electrolytic mineral accretion (seacrete) for digital fabrication? (p. 779) for more notes on seacrete.

(Of course electrodeposition, either of metal or of stone, is also interesting as a way to add strength to aluminum-foil structures, including aluminum-foil origami.)

Topics

- Manufacturing (p. 1151) (29 notes)
- Electrolysis (p. 1158) (18 notes)
- Welding (p. 1181) (9 notes)

Better screw head designs?

Kragen Javier Sitaker, 02021-08-25 (updated 02021-09-11)
(4 minutes)

I was watching some videos the other day of factories in Sialkot, Pakistan, and I was especially interested in the vises used in one factory. It's common to turn vise screws with a sliding round rod permanently stuck through a round hole at one end of the vise screw, but these guys just had the round hole; to turn the screws they had a portable long L-shaped round-rod handle with a right-angle hook on the end of it. By passing the hook through the hole in the vise screw, they had a variable-angle lever available; without removing the hook from the vise, the handle could then be rotated to be parallel to the vise screw, perpendicular to it, or anywhere in between, thus providing them with whatever mechanical advantage they wanted, up to the limit of the handle's length.

In file `swashplate-screwdriver.md` in Derctuo, I described a way to achieve the same thing with a mechanism inside a screwdriver. But I thought this idea of drilling a hole through screw shafts to turn them, rather than sawing a slot in the head, was appealing. Unfortunately, I don't think it's compatible with screw heads being driven in flush with a surface.

The Robertson square drive, Allen hex drive, and Torx drive approaches are also appealing, and although they don't inherently offer variable leverage, with ball-head screwdrivers they do offer the possibility of off-axis driving, like a sort of universal joint.

With such hole-based systems, there's a three-way tradeoff between screw-head strength, screwdriver strength, and screw-head size. Making the screwdriver head larger without changing the screw-head size makes the walls of the screw thinner, thus reducing their strength, at least when they are not fully counterbored into equally hard material. Ultimately the strength of the screw against torsion is limited by its material and diameter — enough torsional load, and the head will twist off — but, in grub-screw-like cases where the screw head is not larger than the rest of the screw, I think the weak point is either the screw head itself or the screwdriver shaft where it enters the screw head, because these both must have scantier cross-sections than the solid neck.

I wonder if you can get some advantage by making a very deep hole drive, perhaps going all the way to the end of the screw. This would unavoidably weaken the screw, though. If you can't, is there a natural limit on how deep you can go before you stop gaining an advantage?

In hole-based screw heads like Torx, you could make the screwdriver head slightly helical in order to pull the driver down into the screw head as you're twisting it, by way of elastic deformation of the screw head. This would work best if you had separate screwdriver and screwundriver heads with helices in opposite directions, so that an increase in available force in one direction doesn't translate to a decrease in the other direction.

By making the hole in the outermost part of the screw head circular and gradually tapering it into the shape of the square, Star of David, etc., you could make the screwdriver auto-align to the screw head's rotational position as it is inserted, at least if the user is providing enough torsional compliance during insertion. This would be very convenient, especially for screws you can't see. Similarly, you could taper the hole outward to the full diameter of the screw head so that the screwdriver auto-aligns to the screw head's translational position as it is inserted, as box-end wrenches have done (on the wrench side) for many generations.

Probably a buttress thread profile like the ones used on toothpaste tubes would be a better default (for fasteners, maybe vises and the like too, though they might be better off with planetary roller screws). I think it roughly doubles the screw thread's strength against pullout.

All in all, though, I think screw fasteners were a mistake. Much of the above also applies to camlocks, scroll drives, etc.

Topics

- Contrivances (p. 1143) (45 notes)
- Mechanical (p. 1159) (17 notes)
- Fasteners (p. 1297) (3 notes)

Dense fillers

Kragen Javier Sitaker, 02021-08-25 (updated 02021-12-30)
(7 minutes)

Making things feel heavy is cool because people feel like light things are cheap and worthless. But there are cheap high-density fillers.

One of the very cheap things Ecoquimica sells (see Material observations (p. 633)) is baryte. I was thinking I didn't have any use for that, but it occurs to me that it might be useful as a high-density filler, with density theoretically 4.48 g/cc. And at Mohs 3–3.5 it should be easy to mill to finer granulometry. Baryte-filled silicone might be 3.4 g/cc and ought to be a bright, pure white, and much less resonant than pure or quartz-filled silicone. I see it's also used for X-ray shielding, for making the first synthetic phosphor, *Lapis Boloniensis* (through carbothermal reduction to the anomalously water-soluble barium sulfide, which doesn't melt until 2235°), and as a metal-casting mold release (melts at 1580°, decomposes at 1600°), as well as a source of barium for other materials like the remarkable and dangerous BaO_2. I'm wary of using it in reactions, though, because of barium heavy-metal toxicity.

For just adding weight and whiteness to things, zinc oxide (5.6 g/cc) might be superior, and is also interesting for oxychloride and phosphate uses, but from Ecoquimica it costs AR$5200/kg (US$29/kg) to baryte's AR$83/kg (46¢/kg). Other vendors have it for lower prices like AR$1500/kg but nothing in the neighborhood of baryte. And if you *just* want density, you can get scrap lead, copper, and steel pretty cheap; a 30kg lead ingot is AR$13500 (AR$450/kg, US$75/30kg, US$2.50/kg), and I'm pretty sure the scrap metal guy around the corner pays something like AR$950/kg for copper, AR$700/kg for brass, and AR$100/kg for lead, all of which are more expensive than baryta (but denser). Steel is too cheap for him to even deal with.

For building machine tool frames, baryta's vibration-damping and density properties might be highly desirable.

As a small dog nipped at my heels in the street, it occurred to me that maybe magnetite might be denser than zinc oxide and also cheaper, but as it turns out magnetite is only 5.17 g/cc, slightly less dense than zinc oxide. Related fillers that really *are* higher density include black cupric oxide (79.545 g/mol, 6.315 g/cc, boils at 2000°, -156 kJ/mol) and the safer but less air-stable red cuprous oxide (143.09 g/mol, 6.0 g/cc, boils at 1800°, -170 kJ/mol), both of which contain one oxygen and some copper (63.546 g/mol, 8.96 g/cc, boils at 2562°).

Also, any of those four oxides could serve as an oxidizer for self-propagating high-temperature synthesis (cf. SHS of magnesium phosphate (p. 608)); magnetite is 231.533 g/mol and -1120.89 kJ/mol (-280.223 kJ/mol O_1) and commonly used; zinc oxide is 81.406 g/mol, 5.606 g/cc, and a very tame -350.5 kJ/mol, and while it boils at 1974°, zinc metal boils at only 907°, which is why welding on

galvanized metal is dangerous. In theory I ought to be able to make black cupric oxide from recycled copper for AR$900/kg × 63.546/79.545 ≈ AR$700/kg; copper dihydroxide (97.561 g/mol, 3.368 g/cc, -450 kJ/mol) dehydrates into cupric oxide at 80°.

On 02021-08-24 I walked by the neighborhood recycler ("COMPRO METALES, X MAYOR Y MENOR, Pago Mas!...") to check on scrap prices. He's offering AR$600/kg for brass and bronze (US$3.3/kg), AR$100/kg for aluminum (56¢/kg), AR$170/kg for lead (95¢/kg), and AR$900/kg for batteries (US$5.0/kg), which it turns out refers to lead-acid batteries, not disposable alkaline batteries (mostly manganese dioxide and zinc) — the 95¢/kg is the price for other lead, I guess things like tire weights, while the factory buys back the dead batteries for more than US$5/kg, presumably because of the high-purity lead content. He's not buying copper at all today, it looks like; I wonder if that means he has scrap copper he can't sell?

I've made copper hydroxide in the past by electrolysis, mixed, probably, with cupric chloride and the acetate, both of which are water-soluble, which should make separation easy. (Cuprous chloride is less water-soluble, but its K_{sp} is still 1.72×10^{-7}, 13 orders of magnitude more than cupric hydroxide's 2.20×10^{-20}, and at any rate it's difficult to make.) If my objective were to make a lot of pure cupric oxide, I could use a plain vinegar electrolyte or a sulfate electrolyte to eliminate the chlorides. I bet Mina would appreciate the pigment, too.

So the points on the Pareto tradeoff curve for density to cost are something like:

- Osmium: US$13000/kg, 22.65 g/cc, or possibly iridium at more than twice that price
- Tungsten: US$30/kg, 19.3 g/cc
- Tungsten carbide? Not sure what it costs but its density is 15.6 g/cc.

- Lead scrap: 95¢/kg, 11.3 g/cc
- Steel scrap: 21¢/kg, 7.9 g/cc
- Magnetite: 10¢/kg or so, 5.2 g/cc
- Quartz (as construction sand): 3¢/kg, 2.6 g/cc
- Water: .06¢/kg or so, 1 g/cc

More briefly:

22.65 g/cc Os (US$13000/kg), 19.3 g/cc W (US$30/kg), 15.6 g/cc WC ($???), 11.34 g/cc Pb (95c/kg), 7.9 g/cc Fe (21c/kg), 5.2 g/cc Fe3O4 (10c/kg), 2.6 g/cc SiO2 (3c/kg)

I feel like WC is probably cheaper than W, because Wikipedia says you can make it by heating WO_3 to 900° with graphite, and the USGS's Mineral Commodity Summaries put WO_3 prices at US$148–US$270 *per tonne* for the last several years.

Mercury, litharge, minium, and cinnabar should probably be on the curve, too.

Not making the curve but still pretty cool, in part because of their colors:

- Baryte: 46¢/kg, 4.48 g/cc
- Zinc oxide: US$29/kg, 5.6 g/cc

- Cupric oxide: US$3.90/kg, 6.315 g/cc
- Manganese dioxide: 5.026 g/cc

I visited the recycler and bought a kg of copper wire from him, which cost AR$1200 (US$6.70/kg). He explained that lead-acid battery lead "is a different alloy" which is a reason for the higher price, also indicating the plastic case as another reason, but I think he sort of has those both backwards. Still, as a heavy filler, scrap lead with other impurities is probably close to the same density as pure battery lead.

Topics

- Pricing (p. 1147) (35 notes)
- Manufacturing (p. 1151) (29 notes)
- Filled systems (p. 1161) (16 notes)
- Argentina (p. 1200) (7 notes)
- Copper (p. 1234) (5 notes)

Selective laser sintering of copper

Kragen Javier Sitaker, 02021-08-30 (updated 02021-12-30)
(6 minutes)

Copper selective laser sintering and similar powder-bed processes have some interesting benefits. Metal powder for 3-D printers apparently costs US$300-600 per kg, but copper is easy to powder electrolytically. Also, copper powder is not an explosive hazard in air as most other metal powders are. It's not quite noble enough to sinter in air, but you should be able to sinter it in nitrogen (Cu_3N does exist at room temperature, but decomposes with a little heating and is estimated to have a positive enthalpy of formation around +74.5 kJ/mol, being a potential conductive copper ink material by this route, and is consequently difficult to synthesize, requiring either sputtering or both pre-oxidized copper and pre-cracked nitrogen), or possibly even a reduced-oxygen air atmosphere or carbon dioxide.

Lasers are a desirable way to pattern the surface because they can achieve high precision and high power in a small area, which is particularly important for very thermally conductive metals like copper. Possible alternatives include electron beams (in vacuum), arcs, localized electrodeposition (in an electrolyte), and an inkjet-printed light-absorbing layer followed by illumination with a strobe light.

This last alternative requires delivering enough heat in the light of the strobe to melt or at least sinter the surface copper layer before the heat can diffuse out of the surface layer into the bulk material, similar to skin burns from the flash of an atomic bomb; but it does need to diffuse to the non-illuminated side of the copper particles in the surface layer. Air-gap flashes can achieve 500 ns speed, but typically their emission spectrum has a lot of blue and green, which might be suboptimal, since copper's reflectivity is not very high in those colors, and we want the un-inked copper to be reflective. Doping the plasma with something like strontium, lithium, or sodium might help to increase emissivity in the red and green, increasing the contrast between the ink and the copper. (Carbon dioxide mostly emits at 4300 nm, but I don't think you can get enough power out of it.)

Copper melts at 1084.62°, boils at 2562°, has a heat capacity of 24.440 J/mol/K near room temperature, and weighs 8.96 g/cc and 63.546 g/mol. This works out to 0.385 J/g/K or 3.45 J/cc/K, and so reaching the melting point (almost necessary for sintering) starting from 20° requires about 400 J/g or 3.7 kJ/cc. Its thermal conductivity is 401 W/m/K, and this is the point at which I suddenly wish I understood the heat equation.

(You don't really want to melt it, but if you did, its heat of fusion would be another 13.26 kJ/mol = 0.2087 J/g = 1.87 J/cc.)

Because I don't understand the heat equation very well, I'm going to work with a really dumb approximation to get a feel for orders of growth.

Suppose we have an 0.1-mm surface molten layer of copper (plus an insignificant amount of carbon susceptor) which ranges from 1100°

to 2200°, and the 0.1-mm layer below it ranges from 20° to 1100°, and that the specific heat and conductivity numbers are unchanged over this range (which they aren't, of course, but this is an approximation). The thermal gradient then is 11 MK/m, giving us a heat flow of 4.4 GW/m², or in less overwhelming terms, 4.4 J/m²/ns. Moreover the thermal energy present (disregarding the heat of fusion) is 3.7 kJ/cc × 0.2 mm = 740 kJ/m², 74 J/cm². So reaching a situation somewhat like this would require depositing those 740 kJ/m² at at least 4.4 GW/m², which requires a flash of less than 0.17 ms.

A faster flash could melt a thinner surface layer, which would contain less energy (proportional to the thickness of the surface layer) and conduct it away from the surface faster (inversely proportional to that same thickness). So, for example, for an 0.01 mm layer, ten times thinner, you would need to deliver only 74 kJ/m², but at 44 GW/m², so instead of 0.17 ms you would need to do it in 0.0017 ms, a hundred times faster, which is getting down to the limits of what an air-gap flash can do.

This approach would probably require stepping a focused area over the copper surface and emitting repeated flash pulses, both in order to keep the energy of a given flash manageably low, and in order to bombard the copper from many directions with the light from a small flash tube in order to be able to achieve a high temperature.

Of course, a Q-switched laser can produce much faster pulses, and much brighter than a blackbody, albeit at much lower efficiency. And a gas-discharge laser might be limited to millisecond or longer pulses, depending on the gas's relaxation time, but you can easily focus it into a 50-micron-diameter area, so even a 10-joule pulse gives you 5 GJ/m² and 5 TW/m², a couple of orders of magnitude higher than you need to melt the surface of copper.

Topics

- Physics (p. 1157) (18 notes)
- 3-D printing (p. 1160) (17 notes)
- Frrickin' lasers! (p. 1168) (12 notes)
- Powder-bed 3-D printing processes (p. 1226) (5 notes)
- Copper (p. 1234) (5 notes)

Negative feedback control to prevent runaway positive feedback in 3-D MIG welding printing

Kragen Javier Sitaker, 02021-08-30 (updated 02021-12-30)
(3 minutes)

3-D printing with a MIG welder ("WAAM") suffers from a positive-feedback problem which impairs geometrical precision: a little bump on one layer tends to attract the arc of the next level and consequently the droplets of metal, becoming a bigger bump on the next level. (Marcin Jakubowicz says, "if you blast the power up, that issue goes away," but apparently it's an issue lots of WAAM companies have, and Joshua Pearce reports in the same conference call that he was scrubbing his prints with a wire brush between layers to reduce this problem.)

Adam Blumhagen has proposed to Open Source Ecology that maybe if you run a grinder over the surface after each layer, like the Ability 3D and Big Metal, you could solve this problem (also potentially getting better resolution than what MIG suffers from surface tension). However, an alternative is to stabilize the system with negative feedback: if you detect that the surface is slightly higher, you can compensate by adding less metal to it.

There are a variety of ways you could detect this. MIG welders in particular are not really designed for this; they try to maintain a constant arc length by maintaining a constant voltage across the arc, but the amount of wire stickout is sort of uncontrolled, being the integral of the difference between the meltoff rate (which is nearly proportional to the current, which you can measure) and the wire extrusion rate. A small error in measuring either of these will work out to a large error in estimating the stickout over time.

You could still use the wire as a conductive CMM probe by letting it cool down first, then using a much smaller voltage and current to probe the surface. If you instead periodically probe a known surface that isn't changing significantly, ideally a piece of graphite or something, you can find out what the current stickout is, correcting the accumulated error in the stickout estimation. Your stickout error will still drift pretty fast, but maybe not fast enough for the positive feedback problem to get out of control.

Using a much thicker electrode, as in stick welding, would largely solve the problem by reducing the linear speed of meltoff; so would using an electrode that isn't constantly melting off, as in TIG welding, although if you're constantly crashing your tip into the work it may not retain its nominal geometry for very long, even if you wait for the work and the tip to cool first.

Topics

- Digital fabrication (p. 1149) (31 notes)
- 3-D printing (p. 1160) (17 notes)
- Welding (p. 1181) (9 notes)
- Control (cybernetics) (p. 1262) (4 notes)

Fast electrolytic mineral accretion (seacrete) for digital fabrication?

Kragen Javier Sitaker, 02021-09-01 (updated 02021-12-30)
(52 minutes)

I was thinking that seacrete (electrolytic mineral accretion) is a really interesting possible digital fabrication process, but it's slow. Seacrete deposition is claimed to be around 5 cm per year normally, which is about 1.6 nm/s, so doubling the thickness of a 10-μm-thick aluminum-foil origami figure would take about an hour, and getting to a thickness of 3 mm would take about 11 days. So speeding up the process would be very worthwhile. Fortunately, there seem to be several likely routes to increase the deposition rate by an order of magnitude or more.

More concentrated electrolytes

First, maybe you could get better Faraday efficiency for depositing seacrete by using a feedstock solution more concentrated than seawater, and with less useless ions. Seawater is only 0.13% magnesium and 0.04% calcium, so the large majority of cations you'll attract to the cathode are sodium (it's 1.1% sodium). (Simply boiling down seawater, or drying it out in ponds, to crystallize out most of the sodium chloride leaving bitterns, would greatly improve the situation.) Making most of the cations something that can participate in the mineralization reaction ought to boost the Faraday efficiency of the process by a factor of 6 or so.

WP gives chalk's solubility as .013 g/ℓ, or 1.3 mg/100mℓ, so maybe that's about how dilute a solution of bicarbonate and calcium ions would need to be, though it gives calcium bicarbonate's solubility as 16.6 g/100 mℓ, so maybe you can go that high; WP says bicarbonate ions are 0.14% of all dissolved ions in seawater, which is 3.5% dissolved ions, so maybe that's about 5 mg/100 mℓ already (depending on whether those numbers are both by weight, or whether one of them is molar). Alternatively, magnesium hydroxide (brucite, Mohs 2.5–3), might be a more practical primary electrolytic mineral accretion material. However, maybe enough dissolved choke-damp could enable higher chalk deposition rates.

Supposedly Epsom salt dissolves 26.9 g/100 mℓ (anhydrous basis) in water at 0°, and only 35.1 g/100 mℓ at 20°, so you could probably dissolve at least 10 g/100 mℓ at -20°. It's 120.366 g/mol, while magnesium is 24.305 g/mol, so it's 20.193% magnesium by weight, so this would be a solution of about 2% magnesium, 15 times as high as seawater; so we might expect a deposition rate on the order of 15 times as high, even without the improved Faraday efficiency from getting rid of the useless sodium.

Upper bound estimations

100% Faraday efficiency with divalent cations would be 3.12×10^{18} ions per coulomb, which is 5.18 micromoles per coulomb, or 0.126 mg

of magnesium per coulomb. At 58.3197 g/mol, that would be 0.302 mg of brucite per coulomb. So at 10 amperes you'd deposit 3 mg of brucite per second, or 10.8 g per hour. If this were distributed over 100 cm², with brucite's density of 2.3446 g/cc, it would be 0.46 mm/hour, which is about 80× the reported rates for seacrete. This is probably an achievable level of current with a tabletop (or household freezer) setup, and depending on the voltage, it might be 10–50 watts, which is an acceptable power level; and you might be able to reach 10–60% Faraday efficiency in real life.

These 10 A/dm² would be 93 "amps per square foot" in the medieval units used by US electroplating shops, which is on the high end of the current levels commonly used for electroplating, so it may be a little high but it's not outrageously so. Nickel sulfamate baths for nickel plating are operated as high as 15 A/dm² at the cathode.

At the other end of the spectrum, suppose we can accrete smithsonite, $ZnCO_3$ (125.4 g/mol, 4.5 g/cc). 5.18 micromoles per coulomb here gives us 0.650 mg of smithsonite per coulomb, or 6.5 mg/s at 10 A, or 23.4 g/hour, and over 100 cm² that's... 0.52 mm/hour. I was hoping for a way more exciting number, but I guess the higher density of the smithsonite mostly cancels out its higher molar mass.

In theory you can use higher current densities than that, but if you keep increasing the current density eventually you will boil your electrolyte; there's also the risk of primary nucleation somewhere other than your cathode, forming a particle which might be swept away and consumed. Somewhere in between, though, I suspect you might be able to control the porosity with current density; higher or lower porosities might be desirable in different situations. In particular, higher porosity will give you higher strength, hardness, and density, while lower porosity will give you easier ionic conduction, faster mineralization (at least volumetrically), higher filler fractions (if you're using fillers), and greater flexural rigidity per material mass.

Temperature and pressure

Doing the whole operation in the freezer would decrease the solubility of the salts and products, but also of the product, which might be a win on net. It would also *increase* the solubility of choke-damp, as would higher pressures.

I think lower temperature would also increase the resistance of the electrolyte, which means lower thermodynamic efficiency. In combination with inert fillers, it might reduce ion mobility to an undesirable degree.

So it might turn out that *high* temperatures rather than low temperatures are most desirable.

Inert fillers

If you buried the cathode "form", of origami aluminum foil or whatever, in sand or silt, perhaps the brucite or calcite forming around the form would concrete the sand together, forming a *real* concrete rather than the ersatz limestone type; this could both dramatically increase the strength of the resulting material multiply the effective deposition rate by a factor of 2–5, since sharp sand and

surface soil typically has a void ratio around 0.4, and subsurface soil is commonly around 0.2. Void fractions as low as 0.1 (and thus multipliers of 10×) might be practical with a combination of fillers, though sufficiently low void fractions will impede ionic conduction.

This could boost maximum theoretical performance in the scenario described above from 0.46 mm per hour of brucite deposited to as high as 4.6 mm per hour of brucite-cemented sand.

Burying a pre-shaped cathode in sand may pose practical difficulties; aluminum foil origami, bent thin wire, or a formed sheet of knitted wire might bend under the weight of even a fairly small amount of sand that settles on top of it. Depending on the shape, you might be able to manipulate the cathode into an existing sand bed so as to diminish the problem; an origami crane, for example, can be inserted vertically into the sand up to its wings, which can then be laid out horizontally on top of the sand, while its head and tail might be able to stand up to the falling sand.

The problem diminishes when the water becomes denser, adding buoyancy to the sand, and when the sand is low-density, like quartz, rather than high-density, like sapphire or forsterite. Using aluminum-wire screening rather than aluminum foil is one attack on the problem; another is to initially pour deflocculated silt over the form, so that it flows around the form on all sides while being barely denser than water, then flocculate it by adding flocculants.

More generally, colloidal fillers, such as deflocculated clay, deflocculated silt, or especially deflocculated micron-sized barytes, might reduce the problem significantly; unlike sand and similar particles, they can flow around the aluminum, and you can adjust them to have density that is very close to that of aluminum to make it closer to neutrally buoyant.

Additional fillers like ground mica, talc, mullite, glass fiber, basalt fiber, chopped rock wool, carbon fiber, and cellulose fiber should help to improve the mechanical properties of the result, as well as decreasing porosity and thus increasing effective deposition rate. Low concentrations of metal fibers or graphite might be sufficient to increase the strength of the result without being present in high enough concentrations to form a conductive network; they might also work to increase the effective surface area of the cathode, thus enabling the use of higher currents and therefore higher deposition rates. If they formed a continuous conductive network, the mineralization would occur at the surface of the sand bed rather than surrounding the cathode within it.

Clays are commonly used as functional fillers in plastics, but I suspect that they might be counterproductive in this application, since even kaolin contracts significantly when dried. Less hygroscopic phyllosilicates like mica and talc can fill their role; so, too, could platy nanocrystals of non-phyllosilicate minerals.

Normally in electrodeposition dendrites are to be avoided, since they screw up the geometry. But in this case it might be worthwhile to start with an initial stage of dendrite-forming electrodeposition to increase the surface area of the cathode, safely ensconced within its powder bed, before switching electrolytes for mineralization.

Powder-bed 3-D printing of cathodes

By powder-bed 3-D printing the sand bed with conductive fillers in it (for example, carbon black, powdered aluminum, powdered silver, powdered gold, or powdered copper) you could produce an anode with an elaborate form; it will not mineralize consistently, since parts of the cathode that are shielded from the ionic current will not electrodeposit, but this is likely acceptable for many uses.

An alternative to printing conductive fillers is printing materials that break down to conductive coatings; for example, silver oxalate readily decomposes to silver and gases on gentle heating. Copper formate similarly decomposes to copper and gases, and has the additional advantage of being water-soluble and thus suitable for inkjet printing; selectively coating sand grains with conductive copper might also allow the use of smaller amounts of copper.

More anions, more and better cations

Other useful abundant mineral-forming cations might include ferrous, cupric, aluminum, ferric, nickel, cobalt, and zinc; less abundant but still useful might be manganese, trivalent chromium, beryllium, tin, vanadium, and titanium. In most cases the sulfate salts would be nearly the most soluble, but using a variety of anions (not just bicarbonate and sulfate but bromate, perchlorate, chloride, chlorate, nitrate, nitrite, acetate, formate, fluorosilicate, bromide, iodate, iodide, etc.) would permit a higher total number of cations than the use of any single anion type. The ordering by solubility (g/100 mℓ at 20°) might be something like the following:

- Calcium: bromate (230), chlorate (209), perchlorate (188), bromide (143), nitrate (129), nitrite (84.5), chloride (74.5), iodide (66), acetate (34.7), bicarbonate (16.6), formate (16.6), fluorosilicate (0.518)
- Magnesium: chlorate (135), iodide (140), bromide (101), nitrate (69.5), chloride (54.6), acetate (53.4), perchlorate (49.6), sulfate (35.1), fluorosilicate (30.8), bromate (<58), formate (14.4), iodate (8.6), bicarbonate (0.077)
- Ferrous: perchlorate (299), bromide (117), nitrate (87.525, unstable at room temperature), acetate ("highly soluble"), chloride (62.5), sulfate (28.8)
- Cupric: chlorate (242), bromide (126), nitrate (125), fluorosilicate (81.6), chloride (73), sulfate (32), formate (12.5), iodate (0.109)
- Aluminum: perchlorate (133), nitrate (73.9), chloride (45.8), sulfate (36.4)
- Ferric: perchlorate (368), nitrate (138), chloride (91.8), sulfate (25.6), iodate (0.36)
- Nickel: iodide (148), chlorate (133), bromide (131), perchlorate (110), acetate ("easily soluble"), nitrate (94.2), chloride (66.8), sulfate (44.4), bromate (28), formate (3.25)
- Cobalt: iodide (203), chlorate (180), fluorosilicate (118), bromide (112), perchlorate (104), nitrate (97.4), chloride (52.9), bromate (45.5), sulfate (36.1), iodate (1.02), nitrite (0.4)
- Zinc: bromide (446), iodide (432), chloride (395), chlorate (200), nitrate (between 98 and 138), sulfate (53.8), acetate (30), formate (5.2)
- Beryllium: perchlorate (147), nitrate (108), chloride (42), sulfate (39.1)

Where an anion is missing from the list, it is sometimes because the salt is insoluble and sometimes because I don't know.

So you could, for example, use a mix of calcium, magnesium, and ferrous cations, with nitrate, chloride, and acetate ions; I think the lowest solubility of the six is calcium acetate, 34.7 g/100 mℓ, so you could probably get an electrolyte with over 100 g/100 mℓ of solutes. Because of the low solubility of magnesium bicarbonate, you probably can't get a lot of calcium bicarbonate into solution if you also have magnesium. (I suspect that most other cations would simply precipitate the bicarbonate as an insoluble carbonate.) Similarly, you can't use sulfate as part of the mix if you're trying to mineralize with calcium ions.

It might be worthwhile to pick anions that won't corrode your anode, which is another thing that's less of a problem at lower temperatures. If you have the luxury of using a gold, platinum, or pyrolytic graphite anode, this is straightforward, but if you're making do with baser metals, you may have to be choosy. See below, however, about anode protection.

There's the possibility of electrolytic iron plating, but it seems that getting an iron metal deposit is actually somewhat difficult, involving extreme pH and temperature conditions (like pH 0.5–1.5 and 87°-99°) and often chelating ligands like tartrate or cyanide or TEA and EDTA. So I suspect that under normal conditions you'll get iron oxides and oxyhydroxides.

Nickel, cobalt, and copper plating out as metals might be a more difficult problem to solve.

Carbonatation

If only a metal hydroxide forms, it is likely possible to convert it to a carbonate afterwards by exposing it wet to air or choke-damp, at the cost of some volume expansion. For example, the very rare theophrastite, Mohs 3.5, might form from nickel salts in the absence of carbonate, but I think should carbonate to nickel carbonate (gaspéite), Mohs 4.5. Aluminum is the exception, since it does not generally form a carbonate, so electrolyzed by itself it would probably produce only gibbsite.

cation	hydroxide	carbonate
calcium	soluble portlandite (2)	chalk (3)
magnesium	brucite (2.5-3)	magnesite (3.5-4.5)
		hydromagnesite (3.5)
ferrous	???	siderite (3.75-4.25)
cupric	spertiniite (soft)	malachite (3.5-4),
		azurite (3.5-4)
aluminum	gibbsite (2.5-3)	µ
ferric	goethite, sorta (5-5.5)	???
cuprous	???	???
nickel	theophrastite (3.5)	gaspéite (4.5)
cobalt	(rather unstable)	spherocobaltite (4)
zinc	soluble rare sweetite (3)	smithsonite (4.5)
		(a type of calamine)
beryllium	behoite (4)	??? soluble

I think you could probably directly precipitate the carbonates by keeping enough choke-damp dissolved in the water under pressure and low temperature, which in most cases would produce a stronger result.

Goethite (FeOOH) is particularly interesting as the hardest mineral in this table; limpets use goethite *fibers* (whiskers) to harden their radula teeth, achieving tensile strengths of 3.0–6.5 GPa, several times higher than Kevlar, any steel, or even spider silk, which is how limpets can literally eat rocks; the elastic modulus measured 120 GPa. Probably electrolytic deposition is not a useful way to grow whiskers, much less protein/whisker nanocomposite metamaterials, but it's useful to have this approximation to goethite's mechanical properties.

Goethite is formed naturally through, among other processes, the oxidation of siderite ($FeCO_3$), a process I suppose must swallow water and throw off choke-damp and hydrogen. Ferrous ions tend to oxidize to ferric in the atmosphere, and there seems to be no ferric carbonate.

Firing the result

If you heat the resulting accreted mineral, you may be able to transform it into something more useful, but generally this will involve the expulsion of some material (water, choke-damp, perhaps hydrogen) and a reduction in volume, which can cause the shape to crumble. Fillers as suggested above may be useful in keeping the resulting stresses below cracking limits.

Brucite (2.3446 g/cc, 58.3197 g/mol) decomposes to magnesia at 350° (periclase, Mohs 6, 3.6 g/cc, 40.304 g/mol) by the loss of a water and, apparently, about 55% of its volume. Magnesia is not very strong but is notable for not melting until 2852°, far exceeding wüstite (1377°), sapphire (2072°) and even quicklime (2613°).

Similar to brucite, aluminum hydroxide (gibbsite, 2.42 g/cc, 78.00 g/mol, Mohs 2.5–3) decomposes at 300° into poorly structured alumina, which converts to α-alumina (sapphire, 3.987 g/cc, 101.960 g/mol) upon further heating, apparently losing 60.33% of the original hydroxide's volume. If it's cementing together grains of silica, you would expect the formation of aluminum silicates like mullite at the sapphire-silica grain boundaries, and the silica might be able to prevent the overall structure from crumbling from this dramatic volume decrease, perhaps instead developing internal porosity. (Silica has its own dunting problems, but they are much less severe than the volume loss from dehydrating gibbsite.)

Alternative fillers that might have less cracking problems include sapphire, forsterite, mullite, larnite, phosphates such as those of calcium and aluminum, and of course zirconia and other well-known refractory materials.

Anode protection and solution replenishment by digestion

Perhaps you could put the anode (which ought to be something inert, maybe lead or carbon) in a block of chalk or slaked lime, so that

the acid formed at the anode harmlessly converts to calcium sulfate rather than being released into the solution to attack the workpiece being formed. (This is another advantage of the sulfate anion, aside from its high solubility with most candidate cations.) However, carbonic acid would attack the chalk and be neutralized by it. Other anions like the chloride should instead liberate calcium ions from the chalk to renew the solution, allowing the use of a smaller amount of solution and lower solute concentrations, since the great mineral reservoir is in the block of chalk. This process would also prevent the anions from reaching the anode itself, where they could be oxidized into undesired byproducts and erode the anode.

Insoluble hydroxides and carbonates of other cations would also work for this purpose. Of course, so would a sacrificial metal anode; but perhaps smithsonite, magnesite, and siderite are cheaper than their corresponding metals, and you can't put calcium in water.

Alternative solvents

Water has many nice features for electrolysis, particularly for this purpose: it's highly polar, relatively nontoxic, relatively stable at commonplace temperatures and pressures, it provides hydroxyl ions to the minerals being formed, and it will dissolve any water-soluble products, which is desirable if you want the final product to withstand contact with water. Still, many other polar solvents are known, and one of them might be better for this purpose; polar solvents include anhydrous ammonia, dimethyl sulfoxide, molten phosphoric acid, acetonitrile, ethanol, ethyl acetate, sulfur dioxide, tetrahydrofuran, nitromethane, dichloromethane, anhydrous formic acid, propylene carbonate, acetone, hydrogen fluoride, anhydrous nitric acid, anhydrous sulfuric acid, glacial acetic acid, formamide, molten salt systems including low-temperature ionic liquids, deep eutectic systems, and dinitrogen tetroxide, but there are many others.

Some of these can contribute their own radicals to the substances being formed through electrolysis; ammonia, for example, could contribute amide and ammonium ions rather than hydroxyl ions, say to precipitate struvite or metal amides like sodamide. I suspect that molten phosphoric acid can dissolve many phosphates; for example, perhaps you could dissolve magnesium ions or even frank (tri)magnesium (di)phosphate in molten phosphoric acid, then electrophoretically accumulate the magnesium anions at the cathode, without reducing them to metallic magnesium? Phosphoric acid has been used for 40 years as an electrolyte in phosphoric-acid fuel cells, operating between 170° and 220°, but in that case it only has to carry hydrogen, of which it is of course eminently capable. I don't know how to find out what else it can solvate.

Propylene carbonate is another particularly promising solvent; it's commonly used as an electrolyte in primary lithium batteries, has a stronger dipole moment and a wider liquid range than water (-48° to 242°), and, like phosphoric acid, is nontoxic and not inflammable.

Oh dude, what about aqueous phosphates?

Phosphate has multiple protonation states very similar to carbonate, with a similar effect on solubility.

Acid (mono)calcium (dihydrogen) (di)phosphate dissolves 2 g/100 mℓ in water and melts at only 109°. By contrast, "dicalcium" (hydrogen) (mono)phosphate, the mineral brushite (Mohs 2.5), is 100× less water-soluble, 0.02 g/100 mℓ, and tricalcium (di)phosphate, the dehydrated version of hydroxyapatite, is less water-soluble still, at 0.00012 g/100 mℓ. Hydroxyapatite itself (Mohs 5) is perfectly insoluble in water. So, if you have a saturated solution of monocalcium phosphate, you ought to be able to get precipitation of the more basic calcium phosphates around a cathode, where there's less phosphate and more calcium, as long as the region doesn't get phosphate-depleted by a factor of 100×. Perhaps more exciting, you ought to be able to do this in molten monocalcium phosphate at very accessible temperatures.

5.18 micromoles per coulomb of dicalcium phosphate (136.06 g/mol) is 0.705 mg per coulomb, and the density is only 2.929 g/cc, so under the conditions considered earlier (10 A over 100 cm²) we'd get 0.866 mm/hour at 100% Faraday efficiency, about twice the growth rate of brucite. For brushite. Except probably you'd end up converting most of it to TCP or hydroxyapatite, which cuts the growth rate in half again.

(Mono)magnesium (dihydrogen) (di)phosphate is not as friendly to water, since it hydrolyzes into phosphoric acid and the insoluble dimagnesium form, and I'm not sure about the aluminum salts.

Other polyprotic anions

I don't think there are corresponding opportunities with the analogous ammonium/ammonia/amide and sulfate/bisulfate/sulfuric-acid systems, at least not in water solution. The sulfates of calcium, lead, strontium, and barium are reasonably water-insoluble, but even calcium bisulfate doesn't seem to exist at all, except in homework-cheating websites and the catalogs of fraudulent chemical merchants.

The other polyprotic acids I'm familiar with are mostly similarly unhelpful; nobody knows what hydrogenchromates would look like, and hydrosulfide ("bisulfide") and hydrogenoxalate ("bioxalate") are similar to bisulfate, with soluble sodium, potassium, and ammonium compounds but no polyvalent cations. Potassium bioxalate is notable as "salt of sorrel"; I'm not sure there's a (di)potassium oxalate.

I'm not sure about citrate, which is triprotic. The magnesium/citrate system does at least have known trimagnesium and (highly soluble) monomagnesium forms, but different sources vary on whether the trimagnesium form is highly soluble or, like tricalcium citrate, sparingly soluble. I also don't know about monocalcium and dicalcium citrate.

Boric acid is triprotic but it's hard to get it to react with things other than itself and to form insoluble compounds; however, as with silicates, there are nesoborates, soroborates (hypothetically), cycloborates (hypothetically), inoborates, phylloborates, and tektoborates. Boracite (Mohs 7–7.5, "very slowly soluble in water") is a tektoborate $Mg_3B_7O_{13}Cl$, which I guess is sort of like trimagnesium heptaborate. The other known tektoborates are chambersite (same thing but with manganese) and hilgardite (same

thing but with calcium (2½ borons per calcium) and a different crystal structure). Why chlorine is always involved in these tektoborates is a mystery to me (well, londonite and rhodizite lack it, but they're beryllium-based and may lack boron entirely). There's a pure calcium borate called nobleite (Mohs 3, 6 borons per calcium) which is a phylloborate, and another called colemanite (Mohs 4.5, 3 borons per calcium) which is an inoborate.

I suspect the calcium borates might work for this: if the overall electrolyte is mostly dilute boric acid with some anions to help keep calcium dissolved (chloride or acetate, say) maybe the calcium concentration around the cathode would get high enough to precipitate insoluble calcium borates. But I'm just speculating, lacking any real evidence that you could maintain an adequately soluble calcium/borate electrolyte at any pH. Magnesium borates are maybe more promising given their natural occurrence.

It would be really interesting if you could solidify waterglass electrolytically; maybe you could drive out those pesky alkali ions that reduce its hardness and glass transition temperature so badly. You'd be left with a silica gel rather than fused quartz, though. (See below about Veeraraghavan et al., who seem to have had success.)

There are about another 35 inorganic polyprotic oxoacids known that might conceivably support the same kind of mineralization as carbonic acid, but I mostly don't know anything about them.

3-D printing stone by directing electric fields

The mineralization reaction, like any electrodeposition reaction, can be limited by ion concentrations or by electric field strength. By using a pointed anode, especially one that's insulated except for the tip, you can concentrate the electric field in a particular area and thus accelerate the electrodeposition there. By varying the current through several such anodes, you can vary the electric field spatially, and possibly also the ion concentration; so by moving them around you can perhaps deposit stone where you like. In the more immediate vicinity of the anodes, however, you form acid, which will *erode* hydroxide and carbonate minerals.

Demolding

Suppose you electrolytically mineralize a thick sheet of hydroxide or carbonate stone onto one surface of a metal sheet cathode which has been bent into some sort of desirable shape. If you now reverse the polarity of the electrolysis, acid should form at the surface of the old cathode and attack the part of the stone that's in direct contact with it; in most cases, this will dissolve it, although there are exceptions such as sulfate attacking calcium compounds or phosphate attacking most things. Then you can remove the stone from the old cathode easily. This could enable the use of such a metal sheet as a reusable mold in a process similar to pottery slipcasting, making many stone copies of the same metal form.

The anodic dissolution process will tend to attack the form, but it's probably possible to get a long form life anyway with metals that are

fairly resistant to that kind of attack, like lead, chromium, gold, or platinum. And it may not be a very serious problem, since the stone will quickly neutralize the acid thus generated.

Previous work

Sometimes a month in the lab can save you an hour in the library. What does the library have to say about this stuff?

Deng et al.

Deng et al. report Faraday efficiencies of 50% to 76.5% in electrolytic precipitation of 99.6% pure magnesium hydroxide with a 100cm² graphite anode 4 cm from a stainless cathode of the same size, with a room-temperature electrolyte of 100 g of magnesium chloride hexahydrate dissolved in 2 ℓ of deionized water and <1% Na^+, running 40 mA/cm² (thus 4 A) for 4 h. They report the best efficiency at 0.5mol/ℓ Mg^{++}. This paper tells me exactly what I wanted to know on the first page (the reproducible setup for their experiment and their major results); the only missing information is the voltage.

The paper then goes on to explain what factors they found affect Faraday efficiency: low magnesium concentrations lower Faraday efficiency; current densities below 40 mA/cm² also lower Faraday efficiency (above which it's basically constant, though they only tried up to 70 mA/cm², i.e., 7 A/dm² or 700 A/m²), which is the opposite of what I expected; so do interelectrode distances under 4 cm because they allow the $MgCl_2$ to recombine; so do Na^+ concentrations under 1% (because they increase electrolyte resistance) or of 3% or over (for obvious reasons), dropping Faraday efficiency from 70.5% at 1% Na^+ down to 65% at 10% Na^+. They don't say whether that's weight percentage or mole percentage, or whether the denominator is the salt, the cations, or the solution.

The brucite precipitated was in the form of 40–200 nm nanoparticles, so it must have been a royal pain in the ass to filter out.

My only complaint is that I would have liked to see some numbers from particular runs, though, including the actual measured mass of the dried magnesium hydroxide rather than the calculated Coulomb efficiency.

At 76.5% faradaic efficiency they'd be getting 36.9 kilocoulombs per mole of brucite (2.3446 g/cc, 58.3197 g/mol), thus 1.58 mg/C and 0.674 μℓ/C, or the other way around, 1.48 MC/ℓ or 633 kC/kg. If we suppose that they were using 3 V, which has to be in the ballpark, then that's 4–5 MJ/ℓ or about 2 MJ/kg. At a nominal price of 4¢/kWh (11 nanodollars per joule, typical for wholesale power, though solar has brought this down by a factor of 4 for new projects in much of the world) this is about 2¢/kg, which is cheaper than construction sand and enormously cheaper than portland cement. Of course that doesn't include the cost of getting hold of bittern, much less purified magnesium salts.

Sano, Hao, and Kuwahara

Sano, Hao, and Kuwahara report efficient electrolytic extraction of magnesium from seawater as 99% pure magnesium hydroxide by

using a cation exchange membrane and “deaerating” the seawater first (either by boiling or by acidifying) in order to remove the choke-damp and thus avoid precipitating chalk. Their objective seems to have been to get magnesium as a structural metal or battery electrode. They ran a solution of 5% sal mirabilis through their “anode channel” (they didn’t want to use a chloride salt to avoid chlorine production) and used platinum-plated titanium electrodes.

If they bothered to say anything about the material properties of the brucite thus formed, or the currents, voltages, current densities, or electrode spacings they used, I must have missed it.

Johra et al.

Johra et al. tested some seacrete; they produced some at 0.8 cm/year with 2.5 V in 25°–31° seawater off Thailand, and tested some more that was accidentally produced by parasitic currents around the Italy–Greece 400 kV submarine power transmission cable, which had more magnesium and was consequently software. They report that brucite deposits from seawater when the pH locally reaches 9.2, and confirm my inference above that brucite is weaker than calcite. The paper contains the following text apparently plagiarized from http://www.globalcoral.org/faq/ (identical matching text *italicized*):

Regarding the CO_2 budget of the calcium carbonate precipitation, one could intuitively think *that since limestone deposition is removing dissolved inorganic carbon from the ocean, this should be compensated by absorption of atmospheric CO_2* into *the ocean*. However, the opposite phenomenon occurs. This can be explained by the fact *that there is* actually *much more dissolved inorganic carbon in the ocean (in the form of bicarbonate ion* HCO_3^-) *than there is CO_2 in the atmosphere*. Consequently, *the predominant reaction* for the precipitation o [sic] calcium carbonate is as follows:

$Ca^{++} + 2HCO_3 = CaCO_3 + H_2O + CO_2$

Therefore, for every two molecules *of bicarbonate precipitated as limestone in the ocean, one molecule* of CO_2 is released into the atmosphere. On the *geological time scale, this is the major source of atmospheric CO_2 along with volcanic* activity 9. More information about Seacrete and materials formed by electrodeposition of minerals in seawater can be found in the publications of Goreau [9,10].

The corresponding text on the Global Coral Reef Alliance site says (again, with identical matching text italicized):

It seems intuitively obvious that *since limestone deposition is removing dissolved inorganic carbon from the ocean, that this should be compensated by one molecule of atmospheric CO2 [sic] dissolving* in *the ocean*, but in fact the opposite happens.

The reason is *that there is much more dissolved inorganic carbon in the ocean, in the form of bicarbonate ion, than there is CO2 [sic] in the atmosphere*, and the ocean is a pH buffered system due to dissolution of limestone sediments and also acid base reactions [sic] involving weathering of oceanic basalts to clay minerals. So *the predominant reaction* is:

Ca++ + 2HCO3- = CaCO3 + H2O + CO2 [sic]

That is to say, in order to preserve pH and charge balance, for each molecule *of bicarbonate precipitated as limestone in the ocean, one molecule* is released as CO2 [sic] to the atmosphere. On a geological time scale, this is the major source of atmospheric CO2 [sic] along with volcanic* gases.

Note that the text that isn’t copied verbatim is only a slight paraphrase.

This text has been on the Global Coral Reef Alliance website since at least 02014-05-05, so it’s clear they didn’t plagiarize it from this 02021 paper, though it’s possible that the authors of the paper *are* the

Global Coral Reef Alliance, in which case no plagiarism would be involved; or that both plagiarized the text from some third source, such as the Goreau papers cited.

However, the paper is not listed in http://www.globalcoral.org/gcra-papers/, and the authors of those papers are Thomas Goreau (husband of Dra. Nora Isabel Arango de Urriola y Goreau, who died in 02016), Verena Vogler, Raymond Hayes, Ernest Williams, Charles Mazel, Paul Andre DeGeorges, R. Grantham, H. Faure, T. Greenland, N.A. Morner, J. Pernetta, B. Salvat, V.R. Potter, Paulus Prong, Munandar, Mahendra, Muhammad Rizal, Chair Rani, Ahmad Faizal, and herrzoox, who Johra et al. fail to list as co-authors. Of these I think Thomas Goreau, herrzoox, and maybe Paul Andre DeGeorges are actually part of GCRA, and Wolf Hilbertz and Dra. Arango de Urriola were also involved. Goreau seems to have predeceased his wife, but they had a son also named Tom, and there seem to be new papers by "Thomas Goreau" from 02020, so there may be two Thomas Goreaus publishing on this topic.

One of the Goreau papers is open access under CC-BY, and doesn't contain this text, being mostly a catalog of ways people have screwed up their seacrete experiments, up to and including connecting the cables backwards!

Their apparent plagiarism aside, Johra et al. report that their low-voltage seacrete was 80.8% aragonite, 18.9% brucite, and 0.3% calcite, while the high-voltage seacrete was 52.3% brucite. They also detected significant amounts of silicon, aluminum, strontium, iron, chlorine, and sulfur in the seacrete samples. They report 2499.2 kg/m^3 (σ=9.1 kg/m^3) for the low-voltage seacrete and 1771 kg/m^3 (σ=17.4 kg/m^3) for the high-voltage seacrete due to higher porosity. Disappointingly, they measured the compressive strength of only the high-voltage seacrete (16.8 MPa), though they did some imprecise improvised tests that suggest that the low-voltage seacrete should be in the neighborhood of 25 MPa.

Alamdari et al.

Alamdari et al. report that magnesium goes from 1272 ppm in seawater to 30,000 ppm in the "end bitterns of NaCl production units from seawater", and they precipitated brucite from that bittern using lye.

Veeraraghavan, Haran, Slavkov, et al.

These researchers succeeded at electrodepositing sodium silicate on galvanized steel for corrosion resistance in 02003, and they mention that Speers and Cahoon had success with "anodic deposition at high voltages" (E. A. Speers and J. R. Cahoon, J. Electrochem. Soc., 145, 1812 (1998)):

> Speers and Cahoon report the deposition of Si from alkaline silicate electrolytes by anodizing Al at 350 V. However, this process is limited to Al or similar metals which have stable anodic oxide films and also involves application of large potentials. Recently, Chigane et al. reported formation of silica thin films on copper substrates through cathodic electrolysis of pH 3.3 ammonium hexafluorosilicate solution. In acid solutions, fluoride ions help keep the silica stable in solution. In the absence of fluoride ions, the bath becomes unstable and precipitates as $Si(OH)_4$. However, high pH and presence of fluoride ions limits the

process developed by Chigane et al. to metals capable of withstanding corrosive environments. Further, the deposits obtained by them were highly porous and hence not suitable as a protective coating.

I'm not sure that what precipitates is orthosilicic acid rather than frank silica, but whatever.

Veeraraghavan et al. were using a 3.22 SiO_2:Na_2O mole ratio solution for their electrodeposition, which I think is similar to the bottle I have here, and platinum-niobium anodes, and apparently they electrodeposited zinc onto their workpieces themselves before beginning the silicate deposition. Initially they diluted the waterglass to 5.6 wt% sodium silicate, pH 10.5, and electrodeposited with a potentiostat† at 12 V for 15' at 75°, and I guess they finally got a 1μm-thick silicate layer of zinc silicate followed by silica and pyrosilicate (Si_2O_7). They report that at room temperature no silicate formed, but further heating to 85° gave highly porous deposits but no faster deposition.

(For my purposes, the higher porosity would be desirable, but their objective was to replace chromate conversion coatings for metal protection, not grow rocks in a tank, and by that measure they achieved an order of magnitude better performance, for which the porosity was undesirable.)

They were definitely depositing on the cathodes, not the anodes; they say, "The silicate deposition was carried out in a two-electrode plating cell made of glass with Pt-niobium anodes. Zinc-plated steel panels (EZG-60G) of surface area 116 cm² each side, as-received from ACT labs[,] were used as the cathodes." They contrast their process with the anodic deposition process of Speers and Cahoon:

> Under an applied potential, before Si anions can be electrochemically reduced on the surface of Zn, all the solvent water will be electrolyzed. The soluble silicate is a complex mixture of silicate anions. Hence it can be expected that under large applied electric fields, the negatively charged silicate species migrate to the anode and are deposited. Speers and Cahoon report that the thickness of the silicate layer formed using such method is limited only by the time of anodic deposition. They report thickness up to 100 μm for 20 min of deposition. ... Note that the silicate layer is not more than 1 μm thick. Unlike anodic silicate deposition, the deposits are very thin 1–3 μm. The maximum thickness seems to be limited to 3 μm. These results indicate that the mechanism of cathodic Si deposition in our case is more complex than was previously reported.

They report that drying the layer at 100° instead of room temperature made it 0.69—2 orders of magnitude more resistive, presumably by affecting the structure of the silica layer, and they show that drying at 175° or 200° made the layer much less full of cracks, and it retained corrosion resistance better afterwards.

They explain the electrodeposition through an increase in hydroxyls and thus pH around the cathode, which increases the polymerization of the silicate and thus decreases its solubility. Weird thing about that, waterglass usually precipitates with *decreasing* pH, but maybe that's because normally you're adding Na^+ or K^+ ions to the solution to increase the pH, and in this case they're adding Zn^{++} ions, or rather bizincate $ZnO(OH)^-$ ions.

If I had to criticize something about this paper, it would be that they don't mention anything about stirring, turbulence, anode size, electrode distances, or the nature of the surface of their anode, which could be important considerations to reproducing their results. Also,

they only analyzed the surface film with EDAX, which can show the concentration of Zn, Si, Fe, and other such heavy elements, but is useless for light elements like Na and O (their EDAX results for these oxide coatings include an obviously spurious “0.00 wt%” oxygen entry, and don’t mention sodium at all). But, from my point of view, the sodium content of these protective films is one of the most important questions, even for their declared objective of corrosion resistance: the lower it is, the better the films will resist years-long immersion in water. Presumably a Keim-like treatment with alkaline-earth cations would help, but whether it’s necessary is clearly an important question for the wide deployment of their process.

† A “potentiostat” is usually the chemist’s name for a voltage regulator. The main difference is that they cost US$3000. There are also three-electrode potentiostats that use a third reference electrode to keep the working electrode (the cathode in this case) at a fixed voltage relative to the electrolyte, but Veeraraghavan et al. don’t mention such a reference electrode and specifically say their “plating cell” was “two-electrode”, so I think they were just using a 12-volt voltage regulator.

Harman 01924

R.W. Harman wrote an article in 01924 about reducing the alkalinity of sodium silicate solutions by a method quite similar to the chlor-alkali process, but starting from sodium metasilicate rather than salt. They comment that silica precipitation on the anode was a problem at high current densities:

> The second method[,] of increasing the C.D. considerably hastens the removal of the alkali and gave good results; but it has its limitations in the fact that, above a certain limit, increase of C.D. causes separation of solid silica on the platinum anode. This limiting C.D., above which silica separates on the anode, varies not only with the dilution but also markedly with the ratio. The more concentrated the silicate solution and the greater the proportion of silica in the ratio, the lower must be the limiting C.D.

They give the sodium/silicon ratios as “gram-equivalent” ratios of sodia and silica, and concentrations as “weight normality” numbers N_w which I think are sodium ion molarities:

> Throughout the whole of this work, the different silicates and mixtures will be designated by the ratio Na_2O:SiO_2 in equivalent proportions, this being the simplest and most convenient system of nomenclature and one already finding general and serviceable use in industry. Thus a ratio of 1:2 contains one equivalent of Na_2O in grams to two equivalents of SiO_2.
>
> All concentrations, except where otherwise stated, are expressed in weight normality (N_w) with regard to their sodium content, i.e. in gram-equivalents of sodium per 1000 grams of water. Thus, a 1 N_w solution of ratio 1:4 contains ½($Na_2O \cdot 4SiO_2$) expressed in grams, in 1000 grams of water.

WP explains:

> By this definition, the number of equivalents of a given ion in a solution is equal to the number of moles of that ion multiplied by its valence. If 1 mol of NaCl and 1 mol of $CaCl_2$ dissolve in a solution, there is 1 equiv Na, 2 equiv Ca, and 3 equiv Cl in that solution. (The valency of calcium is 2, so for that ion 1 mole is 2 equivalents.)

Given the above definition, I’m not totally sure but I guess a mole of SiO_2 would be “two equivalents”. The problem is that SiO_2 isn’t an ion! But we see that Harman considers $Na_2O \cdot 4SiO_2$ to be “1:4”.

So, given all that, we have Harman’s account of what

circumstances were necessary for rapid mineralization on his rotating-platinum-disc anode:

> With a $2N_w$ [2 mol/ℓ Na^+] solution of ratio 1:1 [$Na_2O{\cdot}SiO_2$] or 1:2 [$Na_2O{\cdot}2SiO_2$] a C.D. of 0.044 amps per sq. cm. scarcely diminishes the alkalinity of the solution; a C.D. of 0.15 amps per sq. cm. diminishes the alkalinity quite rapidly but yet does [not?] cause separation of solid silica.

The "not" here is missing from the published paper, but the sentence structure (as well as a paragraph I will quote later) seems to strongly suggest that it should be there, and the numerous typographical inconsistencies and errors also suggest that such an error is eminently possible, which unfortunately completely changes its meaning.

Harman continues:

> With ratio 1:4 [$Na_2O{\cdot}4SiO_2$] a $3N_w$ solution gave a very thick deposit of silica with a C.D. of 0.11 amps per sq.cm. A $2N_w$ solution with a C.D. of 0.11 also gave a thick deposit[,] but with a C.D. of 0.044 amps per sq.cm., although silica was deposited on the anode, at the end of 4 hours the solution on analysis was found to be $0.8N_w$ and its ratio was 1:5.2 [$Na_2O{\cdot}5.2SiO_2$]. This solution was very opalescent and after two days set to a gel and later on exhibited synaeresis. A 0.5 N_w solution of ratio 1:4 with a C.D. of 0.13 amps per sq. cm. also deposited silica, but at the end of 10 hours, during which time the C.D. gradually fell [I guess they weren't using a galvanostat], the resulting solution was found to be 0.08 N_w, with a ratio of 1:40 [$Na_2O{\cdot}40SiO_2$!!]. This dilute solution showed no signs of gel formation.
>
> Thus with ratio 1:1 a $2N_w$, solution may be electrolysed with a C.D. of 0.15 amps per sq. cm. but with ratio 1:4 a $1N_w$ solution gives a deposit with as low a C.D. as 0.044. It has been found possible by this means to prepare 2 N_w solutions of ratios 1:2, $1N_w$ 1:3, and 1:4. Higher ratios than these set to a gel, viz., 1:5 above $0.1N_w$, but in very dilute solutions the removal of alkali can proceed until the solution is practically one of pure silicic acid.

It seems that they consider anodic deposition something to be avoided (perhaps since the silica deposit robs the solution of silica, while the objective of the procedure is to increase the silica-to-sodium ratio), but they considered the 1:1 $2N_w$ 0.15A/cm² result to be acceptable. This reinforces my inference that the "not" I inserted above should be there.

Unfortunately the deposits are only ever described in qualitative terms, as "thick" or "very thick". I'd really like to know whether "thick" means 10μm and electrically insulating, 100 μm, 1 mm, or 10 mm. WHAT DID YOU SEE, HARMAN?

But Harman's main concern here was measuring the conductivity of the solutions, not electrolyzing them.

Weird metals

Aerographite can be 180 g/m³ (180 μg/cc, six times lighter than air) and 1 kPa UTS, soaring in strength to 160 kPa at 8500 μg/cc. The fabrication process involves CVD of graphite at 760° onto a template of ZnO (m.p. 1974°), which is then etched away with H_2, since zinc boils at only 907°.

Zinc is very similar to magnesium (b.p. 1091°) in many ways, but electrodepositing zinc in aqueous solution is feasible, while for magnesium you need to use molten salt. After selectively electrodepositing zinc microwires, you could change chemistry to electrodeposit zinc hydroxide (3.053 g/cc, 99.424 g/mol, amphoteric, soluble in aqueous ammonia) in the same way described above for

brucite, then calcine it to ZnO at 125° with a loss of more than half its volume (5.606 g/cc, 81.406 g/mol, and thus 69 mol/ℓ to the hydroxide's 30.7). Then, you can use it as a template for CVD or PVD (of carbon or anything else that can withstand 907°) and hydrogen-etch away the ZnO.

Topics

- Materials (p. 1138) (59 notes)
- Electrolysis (p. 1158) (18 notes)
- 3-D printing (p. 1160) (17 notes)
- Filled systems (p. 1161) (16 notes)
- Phosphates (p. 1184) (9 notes)
- Composites (p. 1187) (9 notes)
- Magnesium (p. 1213) (6 notes)
- Aluminum foil (p. 1237) (5 notes)
- Solubility (p. 1273) (3 notes)

Patterning metal surfaces by coating decomposition with lasers or plasma?

Kragen Javier Sitaker, 02021-09-03 (updated 02021-12-30)
(7 minutes)

Normally you need high-power lasers (kilowatts) to mark metal. One alternative is to lay down a layer of resist (for example, black electrical tape, or permanent marker) and use a small laser (tens of watts) to burn off the resist, then etch the metal electrolytically, with acid, or with reactive ions.

It occurred to me that when you're burning a layer of resist off the metal, you're also producing reactive ions, and maybe those could etch the metal directly if the "resist" were chosen for that purpose. This is different from normal reactive ion etching, where you first lay down a layer of real resist, and then later put the resist-patterned workpiece into a vacuum chamber that you fill uniformly with a plasma; in this technique, the "resist" *is* the plasma, after you blast it with the laser, so you can do the whole thing at atmospheric pressure. This saves you the trouble of running corrosive byproduct gases through your vacuum pump.

For example, nickel (tetra)carbonyl boils at 43°, and maybe you could form it by heating an oxalate or formate salt on a nickel surface; disodium oxalate, for example, decomposes at 290°, releasing carbon monoxide. Similarly, iron (penta)carbonyl boils at 103°. (To make iron carbonyl, sulfur is used as a catalyst and the process is normally carried out at 5–30 MPa.) Carbonyl complexes are also known of molybdenum, chromium, manganese, cobalt, titanium, ruthenium, rhodium, osmium, iridium, platinum, tungsten, and vanadium, not to mention many more reactive metals, so perhaps they could be etched in the same way, but it might not be possible for all of them; it might be easiest for nickel, followed by iron and rhenium. Doing the whole process under a liquid solvent capable of dissolving the carbonyl complex, such as glacial acetic acid, acetone, or carbon tetrachloride, would likely help with increasing the pressure (and thus the equilibrium carbonyl-complex concentration), controlling the temperature, and reducing hazardous carbonyl emissions.

Carbonyls also decompose under heating to deposit the metal, and so if you were to laser-heat points on a surface in a metal carbonyl atmosphere, you could selectively deposit the metal. (Historically this chemical-vapor deposition process was used for nonselective nickel plating, but abandoned for reasons of toxicity.) Many other chemical vapor deposition processes could be selectively applied in the same way by localized laser heating. This is called laser chemical vapor deposition, and of course electron-beam-induced deposition is a higher-resolution vacuum variant of the same technique.

Alternatives to the ridiculously toxic carbonyls might include seriously toxic organometallic complexes like tetraethyllead (boils at 85°), diethylzinc (boils at 117°, pyrophoric), or triethylaluminum

(boils at 128°, hypergolic with liquid oxygen.) In the particular case of etching aluminum, chloride salts such as ammonium chloride (reversibly decomposes at 337.6°) could perhaps be used to produce aluminum trichloride (sublimes at 180°). If cupric chloride (melts at 498°, decomposes at 993°) were the chloride salt, you might be able to deposit some metallic copper at the same time as etching the aluminum.

Either for etching or for deposition, you could probably use other CVD feedstocks like tungsten hexafluoride; silane (pyrophoric gas at room temperature), trichlorosilane, or TEOS; ammonia (e.g., for depositing silicon nitride); germane; ferrocene (forms from hot cyclopentadiene reacting with iron pipes, boils at 249°); uranium hexafluoride; uranocene; nickelocene (formerly used to prepare nickel films); cobaltocene (sublimes at 171°); cyclopentadienylcobalt dicarbonyl (boils at 140°); pentachlorides of molybdenum, tantalum, and titanium; methane; etc.

If you can start by cooling the substrate metal to cryogenic temperatures, you might be able to use "resists" that are unstable or evaporate at higher temperatures. Fluorine, for example, boils at -188.11° at one atmosphere, and somewhat more practical cryogenic temperatures at higher pressures, and so if you cool your substrate enough, you can bathe it in liquid fluorine which you then encourage to etch it with local laser heating. A liquid "resist" would have the earlier-mentioned advantage of producing higher pressure through confinement, thus making it much easier to produce large complexes containing low-boiling substances like tungsten hexafluoride, and also permit repeated etching of the same spot.

At higher temperatures, such as room temperature, you ought to be able to use other liquids that are fairly inert until heated with lasers. Gasoline or mineral oil, for example, could serve as a source of hydrogen for etching carbon, silicon, or glass, while perfluorohexane or hexafluorobenzene could serve as a source of fluorine for etching, say, tungsten or glass. And of course there are all kinds of metal/acid combinations that etch not at all or very slowly at STP, but which, at high temperature and pressure, or when a surface film is disrupted, etch very rapidly.

There are a surprising number of very stable materials that have low-boiling hydrides, silicon and carbon of course being the most conspicuous, but boron also does. Virtually any acid "resist" could contribute protons to facilitate such etching, but phosphoric acid seems like one of the most promising alternatives due to its high boiling point (over 800°) and willingness to part with its hydrogens. At lower temperatures ammonium compounds might be a promising alternative.

Arcs, as in electric discharge machining, are another possible way of inducing localized high temperatures and pressures at chosen points on the surface of a workpiece, and so could also serve either to promote the deposition or the destruction of solid material at the surface of a workpiece.

Traditional reactive ion etching feedstocks like sulfur hexafluoride and carbon tetrafluoride would also be viable candidates for producing the reactive ions on the surface of the workpiece, but

heavier perfluorocarbons like perfluoropentane, or sometimes even carbon tetrachloride, would probably work better, both because by virtue of being a liquid at room temperature, they have several hundred times more fluorine atoms close to the surface, and because of the increased pressure as mentioned above. Gases like carbon tetrafluoride dissolved in some kind of solvent might also work better than just gases.

Glow-discharge plasmas are another possible way to supply reactive ions. By moving one or many sharp-pointed electrodes around close to a workpiece, you could selectively apply the reactive ions to certain parts of it; this is routinely done with air for "activating" surfaces with nonthermal plasmas, including even biological tissues, but of course other gases would have different effects. The reactive ions will tend to attack the sharp point as well, so it either needs to be inert to the ions, or consumable like a mechanical-pencil lead.

Topics

- Materials (p. 1138) (59 notes)
- Digital fabrication (p. 1149) (31 notes)
- Frrickin' lasers! (p. 1168) (12 notes)
- Patterning (p. 1282) (3 notes)
- Toxicology (p. 1316) (2 notes)
- Plasma (p. 1339) (2 notes)

Rock-wool-filled composites

Kragen Javier Sitaker, 02021-09-03 (updated 02021-12-30)
(2 minutes)

I've been thinking about using rock wool as a cheap fibrous reinforcing filler, but it's difficult to find information about its strength. A study in 01996 by Cáceres, García Hernández, and Rincón tested some rock wool fibers they'd made in their laboratory from Canary Islands basalt, using what seems to have been a sort of 10,000-rpm graphite cotton-candy machine they didn't include a drawing of, getting 639 and 717 MPa, with Young's modulus of 89 and 86 gigapascals.

I don't know if this is comparable to the strength of commercial rockwool, but it's a few times higher than mild steel (A36 is 250 MPa) and a few times lower than Spectra (2500–3500), basalt fiber (4840), or S-glass (4710), and *much* higher than polypropylene (12–43), PMMA (72), or HDPE (26–33). They remark that it's lower than other people report for basalt fibers, but point out that frictions among the fibers in the wool would be expected to provoke surface defects and breakage.

The Young's modulus they report is higher than concrete (30 GPa), bone (14), wood (9–12), plastics (0.228–3.5), magnesium alloy (45.2) or pure aluminum (68); similar to tooth enamel (83), Kevlar (70–112), carborundum (90–137), or stinging nettle fiber (87); and lower than brass (106), bronze (112), copper (110), titanium (116), or A36 steel (200).

A few different papers report that incorporating rock wool into plastics weakens them; for example, Aykanat and Ermeydan, which incorrectly claims that PLA biodegrades in 2–4 weeks, and didn't use any coupling agents. Also they seem to have gotten a lot of porosity in their PLA.

Topics

- Filled systems (p. 1161) (16 notes)
- Strength of materials (p. 1164) (13 notes)
- Composites (p. 1187) (9 notes)
- Poly(lactic acid) (PLA) (p. 1281) (3 notes)

Weighing balance design

Kragen Javier Sitaker, 02021-09-06 (updated 02021-12-30)
(9 minutes)

You can get these US$6 scales that use a load cell to measure weights of up to 500 g with 0.1 g precision, and they work surprisingly well. Similar scales with 0.01 g precision are available, but everyone seems to be out of stock. But for things like food coloring or agar, 0.1 g precision is really not good enough even for making things; and for evaluating the results of things like my waterglass foam tests, it requires unreasonably large samples to get good precision.

A simple balance design to solve this problem is an unequal beam. You put a weight of, say, 400 g on the pan of the balance, and you tie a heavy wire or chain to the top of the weight and loop it over a notch in a horizontal beam. A centimeter away on the bottom side of the beam, there's a fulcrum notch that rests on a knife edge; ten centimeters past that, there's another notch in the top, from which a sample pan is hung. Each gram added to the sample pan pulls up on the 400 g weight with a force of the weight of 10 g, so now you can weigh 50 g with 0.01 g precision.

The principle here is similar to the principle of a bismar balance, in that the balance arms are unequal, but instead of measuring the point along the beam where the center of gravity is located, we're directly measuring the force induced on one end of the lever by an unknown force on the other end. It's more similar to the Roman-steelyard type of balance commonly used in doctors' offices. Since we are not moving the fulcrum, the moments exerted by the unknown mass distribution of the beam remain constant and are tared out automatically; and the inaccuracy of the position of the fulcrum can be calibrated out by calibrating the apparatus with known weights, and it will not move thereafter. Moreover, the movement of the apparatus during weighing is very small (perhaps 100 microns), greatly reducing the the importance of factors like the shifting distribution of the weight of the balance arm itself, the imperfect sharpness of the fulcrum inducing a change in effective fulcrum due to changing beam orientation, or static friction from bearings at pivot points. None of these advantages are shared by the steelyard in practice.

This design inherently limits the force on the delicate load cell; once the force applied is large enough to lift the 400-gram counterweight entirely off the electronic scale's weighing pan, the system goes nonlinear and no further force is applied to the load cell.

Of course after you build the thing you have to measure the leverage multiplier; calibrating to a given multiplier is easy to do in software if you have a way of getting the scale readings digitally.

A second stage of leverage could add another factor of 10 or so, allowing you to, say, measure weighs up to 5 g with 1-milligram precision.xs

Varying local gravitational fields can still introduce errors of ±0.5%, as with any load-cell weighing scale.

It occurred to me that maybe strain-gauge load cells are not the best way to measure mass, not only for that reason, but also because ultimately your measurement of the load cell is an analog voltage, and those are hard to measure with any precision. Microchip app note AN3183 goes into some of the issues involved, using a differential amplifier to cancel things like unknown errors in the power supply voltage; it mentions additional sources of error including manufacturing imperfections, aging, ambient temperature, heating by the excitation voltage, nonlinearity in the resistance change, hysteresis, creep, and noise on the low-level output signal. The cheap scales I mentioned earlier cancel some of these sources of error in part by powering off at random times and retaring when you turn them back on, which is extremely inconvenient if you had tared something like a sample boat. Microchip's appnote strongly recommends using multiple different temperature sensors (which, conveniently, they sell) and correcting temperature-induced errors in software.

Nonlinearity and hysteresis in their sample load cell are ±0.05% of full scale, but creep is ±0.05% per five minutes, and the temperature effect on the zero point is ±2%/°, which means that if the scale changes temperature by 20° during operation, its zero could drift by 40% of full scale; you wouldn't even get one significant figure of precision, much less the 3½ sig figs the other sources of error suggest (and that the cheap scales mentioned earlier seem to deliver).

It occurred to me that if you could convert the mass into a time measurement rather than a voltage, maybe you could avoid some of these problems; quartz crystal oscillators for wristwatches are commonly accurate to 4½ significant figures. In simple harmonic motion the angular frequency is sqrt(k/m), where k is the spring constant, so if you calibrate to a fixed k, then you can square the measured period to get a measurement of the mass. A small error in the period will result in an error twice as large in the estimated mass; so if the period is wrong by 0.001%, the mass will be wrong by 0.002%.

This approach has the advantage of being immune to variation in local gravitational force (unlike ordinary types of spring scales), although it might suffer from creep and aging in your spring. I think common spring steels are pretty stable, though, and their spring constant isn't temperature-dependent by anything close to 2%/°.

The physical size of the oscillations in question could be very small indeed. Ideally you'd like them to be fast enough that you can measure them quickly, and you'd like to maintain a high Q for the physical sprung-mass oscillation. If the whole sample pan with the sample weighs 60 g, for sqrt(k/m) to be 10 kiloradians/s, the spring constant must be 6 MN/m, which is 6 N/micron, about 10 times the weight of the 60 g per micron; so to maintain the acceleration below that of gravity (probably necessary to treat the sample mass as a lump rather than separate particles interacting nonlinearly) we would need to maintain the displacement below 100 nm.

Such small displacements are challenging to detect accurately, even though we don't have to measure their amplitude; but a few microns would be fairly easy. If the spring constant is only 600 N/m, then sqrt(k/m) is 100 radians/second, and the displacement under the 60 g

weight is only 0.98 mm.

I suspect that suspending the whole resonating mechanical system with high compliance (or even locally zero rigidity by canceling negative rigidity with positive rigidity) would be adequate to get high Q. Without high Q your measured frequency will be subject to a lot of error.

You might have to measure six degrees of freedom between the two masses in the resonating part of the scale; trying to restrain unwanted vibrational modes with sliding-contact mechanisms would surely introduce far too much friction to get good Q. Modern flexure design techniques might allow you to restrain them in a friction-free fashion and with adequate linearity.

If, instead of getting the restoring force for resonance from a spring, you got it from gravity, you'd have a sort of pendulum-clock scale. (Reversing the usual situation, this kind of pendulum scale would be subject to errors from local gravitational acceleration, while its spring-based sibling described above is not.) By adding mass to the pendulum, you can increase or decrease its effective length. However, pendulum frequency is inversely proportional to the square root of length (and directly proportional to the square root of gravity), and a meter-long pendulum has a period of close to two seconds (about 2.006 seconds), so a 100-Hz pendulum would have to have an effective length of 25 microns, a scale at which gravity and even mass are comparatively unimportant compared to effects like air resistance and surface adhesion. So I think probably that is not a good design direction.

Topics

- Contrivances (p. 1143) (45 notes)
- Electronics (p. 1145) (39 notes)
- Mechanical (p. 1159) (17 notes)
- Ghettobotics (p. 1169) (12 notes)
- Bootstrapping (p. 1171) (12 notes)
- Precision (p. 1183) (9 notes)
- Metrology (p. 1212) (6 notes)
- Weighing (p. 1267) (3 notes)

Fast-slicing ECM

Kragen Javier Sitaker, 02021-09-08 (updated 02021-12-30)
(3 minutes)

ECM can remove material at a speed limited by the area of the interelectrode process gap, assuming adequate flushing. That is, with the same current density distributed over twice as much electrode area, you can remove twice the volume of material per unit time. The maximum speed obtainable from a given composition of electrodes and electrolyte can thus be expressed in meters per second, and commonly it's a few microns per second.

If you want to slice a block of metal in half, one way to do this is to use a wire electrode, which can cut a plane one line at a time. But if you instead use an electrode in the shape of a saw blade, with electrical insulation everywhere except the edge, the electrode surface area is potentially many times larger, and so you can potentially cut much faster.

Taken to the extreme, this logic suggests using a long line of needles or teeth, a few times wider than the process gap, as the cutting cathode; or possibly two converging blades like a "mandolin" cutter. If the tooth edges have a 10:1 slope, so that for each 1 mm of advancement of the blade into the work they get closer together by 0.1 mm, they have 10.05 times the effective surface area of a wire and thus perhaps ought to be able to remove material about 10.05 times faster. Such aspect ratios, or even more extreme ones, should be viable where they are not viable for a saw, because the only mechanical stress experienced by the teeth is the process fluid flowing around them (and probably through them). By the same token, it should be feasible to make them thinner than a saw blade.

However, around the point of each tooth, it would seem that the material removal rate could be a limiting factor, since the point is advancing into the material without this 10:1 "mechanical" advantage. I'm pretty sure there are ways to overcome this and get a locally higher material removal rate; the problem might take care of itself spontaneously (for example by virtue of the high electrical field around a sharp point), or we might be able to advance the tooth into the work nearly parallel to one of its edges, so that only the other edge "cuts" and the MRR required at the point of the tooth is no higher than anywhere else.

At first, when only the points of teeth are advancing into the workpiece, the MRR will be lower than it would be with just a wire, and also at the end when only small bits of workpiece are left in between the valleys of the teeth; so the length of the teeth ought to be small compared to the thickness of the workpiece.

By applying the same principle in three dimensions with pyramid-shaped "teeth", we ought to be able to rapidly *disintegrate* blocks of metal. By using many independently movable such teeth, we ought to be able to rapidly shape a surface to any shape we want.

Topics

- Contrivances (p. 1143) (45 notes)
- Manufacturing (p. 1151) (29 notes)
- Electrolysis (p. 1158) (18 notes)
- Machining (p. 1165) (13 notes)
- ECM (p. 1186) (9 notes)

Switching kiloamps in microseconds

Kragen Javier Sitaker, 02021-09-09 (updated 02021-12-30) (1 minute)

What would you do if you wanted to dump a capacitor holding 100 joules at 1000 amps in a millisecond, briefly dissipating 100 kilowatts?

Most IGBTs are not equipped to deal with pulse currents like this, and I don't think you can parallel them the way you can MOSFETs, due to current hogging by the hottest device; the IXGX320N60B3 costs US$22 and is rated to dissipate 1700 watts itself, and IXYS doesn't publish a datasheet for it. Similarly for triacs.

MOSFETs like the SIHB33N60E-GE3 are maybe more promising: for US$6 you can switch 600 V and pulses of 88 A with 0.1Ω with a 150-nanosecond turn-on delay plus rise time, and you can safely parallel them. (88 A is almost three times their maximum continuous current of 33 A.) So if you put a dozen of them in parallel (US$70) the datasheet claims you can get them to control a kiloamp pulse. (Probably a good idea to add enough series inductance to keep the pulse current from going higher than that.) The Infineon IPA60R099C6XKSA1 is US$8 in quantity 1 and rated for 112-amp pulses, so you'd still spend US$70.

I wonder if a simple mercury-wetted reed relay would be a better choice.

Topics

- Electronics (p. 1145) (39 notes)
- Pricing (p. 1147) (35 notes)
- Pulsed machinery (p. 1167) (12 notes)
- Power supplies (p. 1176) (10 notes)

Spot welding

Kragen Javier Sitaker, 02021-09-09 (updated 02021-12-30)
(8 minutes)

In theory you should be able to do spot welds with arbitrarily small energies if you do them fast enough and can somehow get the energy to deposit at the place you want it.

But suppose you want to melt a spherical nugget of steel of radius 1mm, which should be a practical thing to do with a carbon arc. Iron is 7.8 g/cc and 55.845 g/mol, holds 25.10 J/mol/K, and sucks up 13.81 kJ/mol in melting, which it does when pure at 1538° but as steels perhaps more typically at 1400° or so, which can be reasonably approximated as 1400° above room temperature. This should require about 30J:

```
You have: (1400 K * (25.10 J/mol/K) + 13.81 kJ/mol) / (55.845 g/mol)
You want: J/g
    * 876.53326
    / 0.001140858

You have: (1400 K * (25.10 J/mol/K) + 13.81 kJ/mol) / (55.845 g/mol) * spherevol(⚬
⚬1mm) * 7.8 g/cc
You want: J
    * 28.638589
    / 0.034917922
```

Maybe make it 100 J to be safe.

You could maybe preheat the workpiece to 200° to make this a little cheaper, but the effect on the required energy is surprisingly small. Preheating it more would rapidly oxidize the surface.

```
You have: (1200 K * (25.10 J/mol/K) + 13.81 kJ/mol) / (55.845 g/mol) * spherevol(⚬
⚬1mm) * 7.8 g/cc
You want: J
    * 25.701598
    / 0.038908087
```

If we very roughly approximate that the temperature gradient is 1400°/mm over the surface of a 1mm-radius sphere, and use pure iron's thermal conductivity of 80.4 W/m/°, we can derive a power for how fast this little weld puddle will be quenched, about 1500 watts:

```
You have: 4 pi (1 mm)**2 1400 K/mm * 80.4 W/m/K
You want: W
    * 1414.4707
    / 0.00070697825
```

That means we have to deliver all of our 30 or 100 or 200 joules within about 20–100 ms, or the steel will suck the heat away faster

than we're pouring it in, and we'll never get a melt. But 20–100 ms is really not a very demanding specification at all.

If, like a handheld point-and-shoot camera with an on-camera strobe, we build up the energy over a period of time before releasing it in a sudden burst to make the weld, thus avoiding the need for a high-power power source, we need to store it in some kind of energy storage element that can release it very rapidly, probably an inductor or capacitor, although a flywheel attached to a motor-generator that can briefly handle multi-kilowatt pulses is an amusing idea too. You might reasonably be able to build up the energy over 100 ms or 1000ms.

The energy of an ideal inductor is $\frac{1}{2}LI^2$, so if you want 100 J in a 1-henry inductor, like those used in old 120V fluorescent-light ballasts, you need to be running 14 amps through it. Actually, let's derive this. WP says, "A general lighting service 48-inch (1,219 mm) T12[30] lamp operates at 430 mA, with 100 volts drop. High output lamps operate at 800 mA, and some types operate up to 1.5 A. The power level varies from 33 to 82 watts per meter of tube length (10 to 25 W/ft) for T12 lamps." To drop 20 V at 430 mA the ballast needs a reactance of 47Ω, which is $2\pi fL$, so we need about 120 mH:

```
You have: 20V/430mA
You want: ohms
    * 46.511628
    / 0.0215
You have: 47 ohms / 2 pi 60 Hz
You want: H
    * 0.12467137
    / 8.0210876
```

So maybe it's common to use inductors of less inductance than that. Regardless, it's going to be a pretty annoying energy storage device, the weight of my fist.

The energy stored in a capacitor is analogously $\frac{1}{2}CV^2$. A 1μF 2100V microwave oven capacitor is also about the weight of my fist but is only rated to hold a couple of joules:

```
You have: half (2100V)**2 1 microfarad
You want: J
    * 2.205
    / 0.45351474
```

Electrolytic capacitors might be a better option. A 400V 1000μF capacitor from a switching power supply is rated to hold 80J and is actually smaller:

```
You have: half 1000μF (400V)**2
You want: J
    * 80
    / 0.0125
```

Electrolytics don't have spectacular ESR and ESL ratings, so they're not useful for fast pulses, but 20ms is not a fast pulse at all.

The Cornell Dubilier 1200μF 380LX122M400A082 costs US$6 and is supposed to have 0.152Ω, though that is of course at 120Hz, but unexpectedly it's supposed to be only 0.053Ω at 20 kHz; it's 82 mm tall and 35 mm in diameter, dramatically smaller than a microwave oven capacitor. Discharging it from 400V in 20ms would take about 24 A, dropping an insignificant 4V across the ESR.

```
You have: 400 V 1200 μF / 20 ms
You want:
        Definition: 24 A
You have: 24 A .152 ohms
You want: V
    * 3.648
    / 0.27412281
```

You'd also need some kind of switch that could turn on the capacitor rapidly, handle 24 amps, and not drop much more than 100 volts itself (so under 4Ω, ideally under 0.4Ω). A US$2 IRF540N MOSFET could almost do the job, but its voltage rating is a little low at only 100V. Something like the US$8 STB45N65M5 would probably be vast overkill; it's rated for 650 V, 0.078Ω, 210 W (!), and 35 A continuous, 140 A pulsed. Something like the US$1.70 STU6N62K3 would work if we ease up a little: 22 amps pulsed drain current, 620 V, 90 watts, 0.95Ω. And the US$1.40 TK650A60F-S4X would be ample: 44 A pulsed current (11 A continuous), 600 V, 45 W, 0.54Ω.

(You could probably even use a bipolar power transistor for this, but MOSFETs seem to have higher pulse currents relative to their continuous currents.)

I suspect that the correct circuit for this is actually fairly similar to a traditional fluorescent-light setup: the storage capacitor has an inductor on its output, and the switching transistor shorts the inductor to the other end of the storage capacitor, allowing current to build up. When the current has risen to the correct level, the transistor is turned off, and the resulting high voltage strikes the arc to the workpiece, which then converts the small amount of energy stored in the inductor and the large amount stored in the capacitor into heat, while the inductor prevents the negative-resistance characteristic of the arc from creating oscillations (although, would it matter if it did?). Because DC-electrode-negative delivers about two thirds of the power to the positively charged workpiece, perhaps because electrons evaporating from the cathode cool it, you probably don't want to be shorting the inductor *to ground* with the transistor, assuming your workpiece is grounded.

That last reference, though, brings the alarming news that "the actual melting efficiency of the arc welding process is relatively low (i.e., on the order of 20 percent or less)." So maybe delivering 30 joules to the metal would involve dissipating 150 or 200 joules, which eats up most of the safety factors in my notes above. However, if you can deliver the same energy in much less than 20 milliseconds, or deliver several kilowatts for a longer time, you should still get a molten nugget.

Perhaps resistance welding would have higher efficiency; it would

certainly simplify the circuitry.

Topics

- Pricing (p. 1147) (35 notes)
- Manufacturing (p. 1151) (29 notes)
- Physics (p. 1157) (18 notes)
- Pulsed machinery (p. 1167) (12 notes)
- Welding (p. 1181) (9 notes)
- Thermodynamics (p. 1219) (5 notes)

Qfitzah: a minimal term-rewriting language

Kragen Javier Sitaker, 02021-09-10 (updated 02021-12-31) (62 minutes)

Today Andrius Štikonas got the `hex0_riscv64` bootstrap seed program down to 392 bytes; it translates from hexadecimal into binary, though much of the bulk of the program is opening and closing files. This led me to thinking about the question of Qfitzat haDerekh, shortening the path: can we teleport directly from a few hundred bytes of machine code to something much more amenable to writing compilers?

Term rewriting languages like Q, Pure, Mathematica, Maude, or Aardappel seem more amenable to writing compilers not only than imperative languages like C but also more so than traditional Lisps; it implicitly provides conditionals, pattern-matching for arguments, ad-hoc polymorphism with multiple dispatch, and parametric polymorphism. As Oortmerssen's dissertation on Aardappel points out (p. 9), term rewriting can be top-down (normal-order, leftmost outermost) or bottom-up (applicative-order, eager, innermost), nondeterministic, as well as other variants; and it can select rule precedence according to source order, uniqueness, according to some kind of specificity, nondeterministically, or in some other way. I think the easiest thing to implement is probably bottom-up source-order rewriting; though top-down evaluation could give you laziness, you don't need laziness or special forms to get conditionals with term rewriting, the way you do with the λ-calculus or the ur-Lisp.

An interesting thing about this is that, despite the vastly increased expressive power (especially for things like writing compilers), the code to implement a term-rewriting interpreter is roughly as simple as the code to implement an ur-Lisp interpreter, just much slower. However, I don't think it's actually any *simpler* than the ur-Lisp, and it's surely a bit larger than `hex0_riscv64`. The surprising thing is that the difference may not be that much.

Still, the unfinished draft Qfitzah interpreter I wrote in assembly is almost 1000 bytes, and for RISC-V (without the C extension) it would probably be even bigger.

A Scheme strawman interpreter for term rewriting

The interpreter is pretty simple. Basically, to evaluate a non-atomic expression, you first evaluate its components, and then you loop over the rewrite rules, trying to match each one against the expression, and when one of them succeeds you instantiate its replacement with the match values, then evaluate the instantiated replacement. Something like this in Scheme:

```
(define (ev t)                                    ; eval, for term rewriting
```

```
  (if (pair? t) (ap (map ev t) rules) t))      ; atoms don't get rewritten

;; apply, for term rewriting, but the arguments are the top-level term
;; to rewrite, after all its children have been rewritten as above, and
;; the remaining set of rules to attempt rewriting with
(define (ap t rules)
  (if (null? rules) t               ; no rewrite rules left? don't rewrite
      (let ((m (match t (caar rules) '()))) ; initially no vars () match
        (if m (ev (subst (cadar rules) m)) ; rule matched? substitute & eval
            (ap t (cdr rules))))))        ; otherwise, try the other rules
```

That depends on definitions of `match`, `emptyenv`, and `subst`. `subst` is easy enough (though I got it wrong at first, and it might be nicer to handle the case where the variable is undefined):

```
(define (subst t env)
  (if (var? t) (cdr (assoc t env))
      (if (pair? t) (cons (subst (car t) env) (subst (cdr t) env))
          t)))
```

So, for example, `(subst '(You #(vt) my #(np)) '((#(np) . wombat) (#(vt) . rot)))` evaluates to `(You rot my wombat)`, as you'd expect.

Then `match` needs to compute whether there's a match, which requires it to distinguish variables from other things. In its simplest form we can consider variables that occur more than once an error, but an error we don't try to detect; then it might look like this:

```
(define (match t pat env)
  (if (var? pat) (cons (cons pat t) env)  ; vars match anything
      (if (pair? pat)  ; pairs match if the cars match and the cdrs match
          (and (pair? t)  ; a pair pattern can't match an atom
               (let ((a (match (car t) (car pat) env)))  ; try to match car
                 (and a (match (cdr t) (cdr pat) a))))   ; then, try the cdr
          (and (equal? pat t) env))))  ; atoms match only themselves
```

Then you just need some kind of convention for marking variables. The simplest thing in Scheme would be to use `,x`, which is syntax sugar for `(unquote x)`, but that would have the unfortunate effect that you can't use the atom `unquote` in head position in either a pattern or a replacement template. Instead I am using `#(x)`:

```
(define var? vector?)
```

(In other environments, you might use a different data type.)

And that's it. 19 lines of Scheme in terms of `define`, `if`, `'()`, `cons`, `pair?`, `let`, `car`, `cdr`, `caar`, `cadar`, `#f`, `and`, `equal?`, `assoc`, `map`, and `null?` gives you a bottom-up, source-order-precedence term-rewriting interpreter. If we want to include implicit equality testing in patterns when a variable occurs more than once, it's a couple more lines of code, but that's about a 10% total complexity increase.

Of course this approach to term-rewriting interpretation is very inefficient, far more so than the standard Lisp tree-walker approach, because every expression evaluation involves iterating over potentially

all the code in the entire program to see which ones apply. Aardappel (and, I think, Mathematica) requires the head of each term to be an atom, and compiles the usually small number of rules for each atom down into a subroutine, so that when attempting to rewrite any term, you can start by doing a hash table lookup, and then typically have a small number of conditionals after that. So you'd probably want to use this bootstrap interpreter only to run a bootstrap compiler.

Primitives and integers

Formally speaking we don't need arithmetic primitives, since we can define numbers in a variety of ways via term rewriting, but for practical efficiency we probably want access to machine arithmetic.

I'm thinking that the way to handle things like arithmetic is to have an additional class of constant atoms, the integers, and some built-in rewrite rules that are implemented in machine code. Darius Bacon suggests that perhaps only the right-hand side should be implemented in machine code, and the pattern-match itself maybe in a prelude.

The question is how to handle them for matching: can you pattern-match on integerness, or do you just have a "function"? In the first case, you could allow an integer like 283 to match a pattern like `Int x` and bind `x` to 283; or you could instead rewrite `Int? 283` and similar to `Yes` and `Int? Qfitzat` to `No`.

If you were doing the pattern-match in the prelude, these alternatives might look like:

```
+ (Int x) (Int y) :: 3
- (Int x) (Int y) :: 4
- x: - 0 x
```

where the `::` rather than `:` indicates that you're supplying a machine-code primitive index rather than a template, 3 being the index of the addition routine and 4 that of subtraction; or, in the other case:

```
If Yes a b: Do a
If No a b: Do b
&& x y: If x y x
|| x y: If x x y
Not Yes: No
Not No: Yes
Do Yes: Yes
Do No: No
+ x y: If (&& (Int? x) (Int? y)) (Int+ x y) (Add x y)
Do (Int+ x y) :: 3
- x: - 0 x
- x y: If (&& (Int? x) (Int? y)) (Int- x y) (Sub x y)
Do (Int- x y) :: 4
```

These If rules are convenient here but maybe not ideal, both because it's easy to forget the `Do` when you try to define the results, and because it's easy to forget that the *arguments within* the consequent and alternate blocks *will* be evaluated eagerly.

You could define the usual arithmetic operations in terms of a single machine-code primitive with multiple arguments, like Mulsubdiv a b c d = (a*b - c)//d, with definitions in the standard prelude like these:

```
+ (Int x) (Int y): Mulsubdiv -1 x y -1
- (Int x) (Int y): Mulsubdiv 1 x y 1
* (Int x) (Int y): Mulsubdiv x y 0 1
/ (Int x) (Int y): Mulsubdiv 1 x 0 y
```

Alternatively you could have Arithmetic x y which evaluates to a tuple containing all the results, like Results <x+y> <x-y> <x*y> <x//y> <x%y>, and you could write

```
+ (Int x) (Int y): R15 (Arithmetic x y)
- (Int x) (Int y): R25 (Arithmetic x y)
* (Int x) (Int y): R35 (Arithmetic x y)
/ (Int x) (Int y): R45 (Arithmetic x y)
% (Int x) (Int y): R55 (Arithmetic x y)
R15 (a b c d e f): b
R25 (a b c d e f): c
R35 (a b c d e f): d
R45 (a b c d e f): e
R55 (a b c d e f): f
```

Either would avoid needing four or five separate primitive subroutines, four or five separate conditional cases to call them, four or five separate subroutine table entries, etc. Though maybe a single conditional case would be sufficient, Primop op x y, with an op number, you'd still need a table of subroutines.

For bitwise operations you could similarly use Norshift a b c = ~(a | b) >> c and definitions like these:

```
B~ (Int x): Norshift x 0 0
Nor (Int x) (Int y): Norshift x y 0
| (Int x) (Int y): B~ (Nor x y)
& (Int x) (Int y): Nor (B~ x) (B~ y)
&^ (Int x) (Int y): Nor (B~ x) y
>> (Int x) (Int y): Norshift (B~ x) 0 y
^ (Int x) (Int y): Nor (Nor x y) (& x y)
```

Probably three arguments is the maximum that would be tolerated by ordinary decency, but if not, you could of course incorporate Mulsubdiv and Norshift into a single six-argument monster.

You'd also need some kind of comparison operation, minimally >0.

Left shifts are easy enough to implement as rewrite rules with addition:

```
<< (Int x) 0: x
<< (Int x) (Int y): If (>0 y) (Shift (+ x x) (- y 1)) (Negative-left-shift x y)
Do (Shift x y): << x y
```

Possibly a better way to implement that would be:

```
<< (Int x) 0: x
<< (Int x) (Int y): <<2 (>0 y) x y
<<2 Yes x y: Shift (+ x x) (- y 1)
```

There's a separate question of how to handle system calls, which I think can be bodged in pretty easily since evaluation is eager.

Some sketches of assembly implementations

But that's Scheme! In machine code it seems like it could be significantly larger, even without garbage collection, which isn't necessary for a bootstrap interpreter on a modern machine, and parsing, which is. You also have to actually *implement* `cons`, `pair?`, `car`, `cdr`, `equal?`, `assoc`, `map`, and `null?`. Most of these are not very difficult.

(None of the assembly code below is tested.)

Because I still know almost no RISC-V assembly, I'm going to sketch this out in the i386 assembly of my childhood.

Type tags in RAM

In i386 code, using the simple approach I took in Ur-Scheme, cons might be 18 bytes:

```
cons:   movl $0x2ce11ed, 0(%ebx)  # allocation pointer is in %ebx
        mov %eax, 4(%ebx)         # car was arg 1, in %eax
        mov %ecx, 8(%ebx)         # cdr
        mov %ebx, %eax            # return the old allocation pointer
        lea 12(%ebx), %ebx
        ret
```

Then `pair?`, leaving the predicate result in ZF, might be 7 bytes:

```
pairp:  cmpl $0x2ce11ed, (%eax)
        ret
```

And an unsafe `cdr` might be 4 bytes:

```
cdr:    mov 8(%eax), %eax       # probably better to open-code these 3 bytes
        ret
```

Type tags in pointer low bits

But the SBCL approach of tagging the pointer would be shorter:

```
sbcons: mov %eax, 0(%ebx)       # car was arg 1
        mov %ecx, 4(%ebx)
        lea 3(%ebx), %eax
        lea 8(%ebx), %ebx
        ret                     # 12 bytes, not 18

sbpairp:
        and $3, %al
        cmp $3, %al
        ret                     # this reduces to 5 bytes
```

```
sbcar:  mov -3(%eax), %eax
        ret                     # still 4 bytes

sbcdr:  mov 1(%eax), %eax
        ret
```

Type tags in pointer low bits where 00 denotes a pair

If we instead use two low-order 0 bits to tag cons pointers, list operations get smaller still, to the point where almost all of them are so small that they need to be open-coded:

```
00000036 <altcons>:  # 11 bytes
  36:   89 03                   mov    %eax,(%ebx)
  38:   89 4b 04                mov    %ecx,0x4(%ebx)
  3b:   89 d8                   mov    %ebx,%eax
  3d:   8d 5b 08                lea    0x8(%ebx),%ebx
  40:   c3                      ret

00000041 <altpairp>: # 3 bytes
  41:   a8 03                   test   $0x3,%al
  43:   c3                      ret

00000044 <altcar>:   # 3 bytes
  44:   8b 00                   mov    (%eax),%eax
  46:   c3                      ret

00000047 <altcdr>:   # 3 bytes
  47:   8b 40 04                mov    0x4(%eax),%eax
  4a:   c3                      ret

0000004b <altnullp>: # 3 bytes
  4b:   85 c0                   test   %eax,%eax
  4d:   c3                      ret
```

A sketch of subst in i386 assembly

Here's what subst might look like with that setup.

In i386 assembly:

```
        # subst t env returns a version of t with var substitutions from env.
subst:  push %ebp               # callee-saved variable used here
        push %eax               # %eax has t, %ecx has env
        push %ecx
        test $2, %al            # ...10 is the pointer type tag for vars
        jz 1f

        call assoc              # calls are 5 bytes. inherits both t & env
        mov 4(%eax), %eax       # get the cdr
2:      pop %ecx                # discard saved arguments; labeled for
        pop %ecx                # epilogue sharing
        pop %ebp
        ret
```

```
1:      test $3, %al            # ...00 is the pointer type tag for pairs
        jz 1f                   # if this isn't a pair:
        jmp 2b                  # %eax is already t; implicitly return it

1:      mov 4(%eax), %eax       # get cdr t for (subst (cdr t) env)
        call subst              # inherits our env.
        mov %eax, %ebp          # save subst result
        mov 4(%esp), %eax       # load saved t
        mov (%eax), %eax        # car t
        mov (%esp), %ecx        # second argument is saved env
        call subst
        mov %ebp, %ecx          # second cons argument is (subst (cdr t) env)
        call cons
        jmp 2b                  # return cons result
```

That's 24 instructions and 58 bytes of machine code, and, although I'm sure I missed a few tricks, I don't think it's going to get more than about 30% smaller. That's 14½ bytes per line of Scheme, which I think is pretty good, but it puts the estimate of the whole 19-line Scheme program at 275½ bytes, which doesn't include the non-open-coded primitives like `cons` (and `assoc` and `map`), the parser, or I/O. I/O is actually almost all of `hex0_riscv64`.

I went through and coded the whole thing in assembly language; after trimming it down a bit, the resulting (untested) program is 100 instructions and 237 bytes of machine code (12.5 bytes per line of Scheme), containing `cons`, `subst`, `assq`, `match`, `evlis`, `ev`, `ap`, and no undefined symbols; so 275½ was actually a little high. I'm pretty sure I could squeeze it down a bit more, but probably not below 200 bytes. It's still missing I/O, the reader, and the printer. In the process I trimmed down `subst` itself to 20 instructions and 55 bytes, then later 22 instructions and 49 bytes.

Adding an input reading loop cost 49 more bytes of code, plus 4 of data; a printer (untested) cost another 81 bytes. Now I'm at 370 bytes of code. I think all it's lacking now is a reader, so I'm pretty sure it'll be under 512 bytes of machine code and data, thus under 1 KiB of executable. I'm currently suffering 414 bytes of executable-format overhead, mostly padding, but Brian Raiter's work suggests that it should be possible to get the executable-format overhead down to about 45–52 bytes, but with strict ELF conformance he couldn't get it below 76 bytes, and with dynamic linking he couldn't get it below 297 bytes; still, maybe I can get the whole executable under 512 bytes.

P.S. a reader was 287 bytes.

However, it'll still be missing library functions to do useful things like I/O and arithmetic.

Trying to do this in RV64 without the C compressed-instruction extension, like `hex0_riscv64`, would surely have much worse code density; *with* the C extension it might be slightly more compact.

A sketch of `subst` in a stack bytecode

In one of the bytecodes suggested in A compact bytecode sketch that should average about 3 bytes per line of C (p. 720) this might

look like this:

```
PROCEDURE subst argwords=2
    loadword 0  ; t
    call var?
    jztos 1f    ; if not a var (top of stack is 0/nil/false), skip
    loadword 0
    loadword 1  ; env
    call assoc
    call cdr
    ret
1:  loadword 0
    call pair?
    jnztos 1f   ; skip if it *is* a pair
    loadword 0  ; return t
    ret
1:  loadword 0
    call car
    loadword 1
    call subst
    loadword 0
    call cdr
    loadword 1
    call subst
    call cons
    ret
```

According to the hypotheses in that note, this might compile to 2 bytes of procedure header, 23 opcode bytes, 9 operand bytes for call instructions, 2 operand bytes for jump targets, and a 2-byte entry in a global subroutine table, for a total of 38 bytes. At this rate, the whole Scheme program irresponsibly extrapolates to 180½ bytes, but of course you'd have to add the bytecode interpreter on top of that. On the other hand, if the bytecode interpreter is customized specifically to run the term-rewriting interpreter, none of the call instructions will need an operand byte, because there's plenty of opcode space to allocate each subroutine in this program a single-byte opcode. That would bring it down to 29 bytes, half the size of the i386 machine code, irresponsibly extrapolating the whole Scheme program to 137¾ bytes of machine code.

A stack bytecode specifically designed for list processing might have a `decons-or` instruction, which unpacks the pair on top of the stack into a car and cdr, or if it's not a pair, jumps to the given destination, because usually `pair?` is associated with subsequent calls to `car` and `cdr`. We could represent that in Scheme as a special binding form, here introducing variables called `a` and `d`:

```
(define (subst t env)
  (if-pair t (a d) (cons (subst a env) (subst d env))
    (if (var? t) (cdr (assoc t env)) t)))
```

In such a bytecode:

```
PROCEDURE subst argwords=2
```

```
    loadword 0  ; t
    decons-or 1f
    loadword 1  ; env
    call subst
    swap
    loadword 1
    call subst
    call cons
    ret
1:  loadword 0
    call var?
    jztos 1f    ; if not a var (top of stack is 0/nil/false), skip
    loadword 0
    loadword 1
    call assoc
    call cdr
    ret
1:  loadword 0
    ret
```

This reduces subst from 23 bytecode instructions to 19, and I think it's likely that a single omnibus pair?-and-jztos-and-car-and-cdr procedure will also be smaller than three separate ones and their opcode table entries. The cost is that, in the relatively infrequent case where only one of those three operations is called for, it costs you an extra byte per unwanted result.

Dynamic dispatch

Because patterns can dispatch on more than just the function being called, you can define "methods" on "classes"; one of the examples in the Aardappel dissertation is defining a hash method for a point class:

```
hash(point(x,y)) = x*y
```

Or, in S-expression syntax using vectors:

```
(hash (point #(x) #(y))) => (* #(x) #(y))
```

Which you could feed into the Scheme strawman interpreter above as follows:

```
(define rules '(((hash (point #(x) #(y))) (* #(x) #(y)))))
```

Elsewhere you might define how to rewrite hash expressions applied to other types of values; then a hashtable implementation can invoke hash without worrying about the types of its arguments. The compiler transformation described above would gather all the hash rules togther into a subroutine, which then performs a sequence of conditional tests to dispatch to one of them.

This also permits CLOS-style multiple dispatch:

```
(* (scalar #(s)) (vec #(x) #(y) #(z))) =>
    (vec (* #(s) #(x)) (* #(s) #(y)) (* #(s) #(z)))
```

```
(* (scalar #(s)) (scalar #(t))) => (scalar (* #(s) #(t)))
(* (m #(r1) #(r2) #(r3)) #(vec)) => (vec (dot (vec . #(r1)) #(vec))
                                         (dot (vec . #(r2)) #(vec))
                                         (dot (vec . #(r3)) #(vec)))
(dot (vec #(a) #(b) #(c)) (vec #(d) #(e) #(f))) =>
    (+ (* #(a) #(d)) (* #(b) #(e)) (* #(c) #(f)))
```

Those rules, for example, will rewrite

```
(* (m (1 2 3) (4 5 6) (7 8 9)) (vec x y z))
```

to

```
(vec (+ (* 1 x) (* 2 y) (* 3 z))
     (+ (* 4 x) (* 5 y) (* 6 z))
     (+ (* 7 x) (* 8 y) (* 9 z)))
```

Higher-order programming

In languages like C or Lisp, the atom-head requirement Aardappel has would prevent you from doing any higher-order programming, but not in term-rewriting languages, because you can use the same dynamic-dispatch trick. You can define a higher-order mapcar function as follows:

```
(map #(f) nil) => ()
(map #(f) (cons #(car) #(cdr))) =>
    (cons (call #(f) #(car)) (map #(f) #(cdr)))
```

Then you can define patterns like this:

```
(call (cover #(material)) #(base)) => (some #(material) covered #(base))
```

so that `(call (cover chocolate) raisins)` rewrites to `(some chocolate covered raisins)`. These rules compose so that

```
(map (cover leather) (cons armchairs (cons bikers (cons goddesses nil))))
```

rewrites to

```
(cons (some leather covered armchairs)
      (cons (some leather covered bikers)
            (cons (some leather covered goddesses) nil)))
```

I learned about this on p. 35 of the Aardappel dissertation, which gives the example (in slightly different syntax):

```
apply(qsortcompare(x),y) = y<x
filter([],_) = ([],[])
filter([h|t],f) =
  if apply(f,h) then ([h|a],b) else (a,[h|b])
    when (a,b) = filter(t,f)
```

In the syntax I used above, this would be written as

```
(apply (qsortcompare #(x)) #(y)) => (< #(y) #(x))
(filter nil #(_) => (pair) nil nil)
(filter ((cons #(h) #(t)) #(f))) =>
    (filter2 (filter #(t) #(f)) #(h) #(f) (apply #(f) #(h)))
(filter2 (pair #(a) #(b)) #(h) #(f) true) => (pair (cons #(h) #(a)) #(b))
(filter2 (pair #(a) #(b)) #(h) #(f) false) => (pair #(a) (cons #(h) #(b)))
```

(This also demonstrates how the term-rewriting paradigm implicitly provides conditionals.)

Oortmerssen discusses this further in pp. 48–50 (§4.1.2).

In the language implemented by the interpreter above, you could just as well define things this way:

```
(map #(f) (cons #(car) #(cdr))) => (cons (#(f) #(car)) (map #(f) #(cdr)))
((cover #(material)) #(base)) => (some #(material) covered #(base))
((qsortcompare #(x)) #(y)) => (< #(y) #(x))
(filter ((cons #(h) #(t)) #(f))) =>
    (filter2 (filter #(t) #(f)) #(h) #(f) (#(f) #(h)))
```

This would have the advantage that you could pass in the name of any existing function. But compiling this efficiently might be nontrivial.

It might be worthwhile to implement lambda-lifting to get closures. Aardappel experimented with this but ultimately rejected it.

A metacircular term-rewriting interpreter

If you love term rewriting so much, why don't you marry it, huh? Why'dja write that "strawman" above in *Scheme*? Are you *chicken*?

XXX the below lacks some bugfixes from the Scheme. Also I definitely do not want to use the shitty Scheme syntax.

Well, part of it is that I think Scheme is a better pseudocode for assembly language, but maybe it would look something like this, written in itself:

```
(ev (cons ,f ,a) ,r) => (ap (args (cons ,f ,a) ,r) ,r ,r)
(ev ,t ,_) => ,t
(args nil ,_) => nil
(args (cons ,a ,d) ,r) => (cons (ev ,a ,r) (args ,d ,r))

(ap ,t norules ,_) => ,t             # no rules left to match
(ap ,t (rule ,pat ,tem ,r) ,r0) =>  # try to match a rule
    (ap2 ,t ,r (match ,t ,pat emptyenv) ,tem ,r0)
(ap2 ,t ,r nomatch ,_ ,r0) => (ap ,t ,r ,r0)  # on failure try others
(ap2 ,t ,r ,env ,tem ,r0) => (ev (subst ,tem ,env) ,r0)  # or subst & eval

(subst (cons ,a ,d) ,env) => (cons (subst ,a ,env) (subst ,d ,env))
(subst ,t ,env) => (subst2 ,t (lookup ,t ,env))
(subst2 ,t nomatch) => ,t
(subst2 ,t ,v) => ,v
```

```
(match ,_ ,_ nomatch) => nomatch               # match failures always win
(match ,t (var ,v) ,env) => (bind ,v ,t ,env)  # otherwise vars always do
(match (cons ,ta ,td) (cons ,pa ,pd) ,env) =>  # match cars and cdrs
    (match ,td ,pd (match ,ta ,pa ,env))
(match ,t ,pat ,env) => (match2 (equal? ,pat ,t) ,env)
(match2 true ,env) => ,env
(match2 false ,env) => nomatch
```

So that's 21 lines, about the same as Scheme, but I left out `lookup`, which in Scheme is the standard procedure `assoc`:

```
(lookup ,_ emptyenv) => nomatch
(lookup ,v1 (bind ,v2 ,t ,env)) => (lookup2 ,v1 (equal? ,v1 ,v2) ,t ,env)
(lookup2 ,_ true ,t ,_) => ,t
(lookup2 ,v false ,_ ,env) => (lookup ,v ,env)
```

Tht brings the total to 25 lines.

I think I may have some unresolved confusion between `(var x)` and `x`; which is supposed to occur in the template? In Scheme, `(var x)`. Also, which is supposed to occur in the environment? Also `(var x)`. Maybe instead of

```
(subst ,t ,env) => (subst2 ,t (lookup ,t ,env))
```

I intended to write

```
(subst (var ,t) ,env) => (subst2 ,t (lookup ,t ,env))
```

I should go back and review this.

Also, `equal?` needs to be provided by the system, at least for atoms, which requires some sort of special case like this:

```
(ap (equal? ,x ,y) ,_ ,_) => (equal? ,x ,y)
```

This points out how easy it is to add special cases like this in a term-rewriting system, though we need to make sure the rule precedence is such that adding the rule has an effect. Of course, this implementation doesn't tell us anything about which definition of equality is being used; this sort of thing is one of the common objections to the use of metacircular interpreters to define semantics.

If, instead of being a magic function name, it is provided through multiple uses of the same variable name in a pattern, we could allow this definition of equality to flow through the metacircular interpreter in the same way; instead of

```
(match ,t (var ,v) ,env) => (bind ,v ,t ,env)
```

we have

```
(match ,t (var ,v) ,env) => (match3 ,v ,t (lookup ,v ,env) ,env)
(match3 ,v ,t nomatch ,env) => (bind ,v ,t ,env)   # new var, not bound
(match3 ,v ,t ,t ,env) => ,env   # already bound to the same value
(match3 ,_ ,_ ,_ ,_) => nomatch  # all other cases are conflicting bindings
```

There's an additional buglet: lookup should return its positive results in a form that can't be the symbol `nomatch`. So maybe instead of

```
(lookup2 ,_ true ,t ,_) => ,t
```

we should say

```
(lookup2 ,_ true ,t ,_) => (got ,t)
```

Also probably it's confusing that `lookup` and `match` return the same `nomatch` on failure.

At least playing with it mentally like this, I feel like this term-rewriting paradigm is a much more pliant medium than Lisps are.

A note on syntax

Above I've been slavishly following Scheme syntax; but it would probably be an improvement if instead of writing

```
(args (cons ,a ,d) ,r) => (cons (ev ,a ,r) (args ,d ,r))
```

we wrote something more like Darius Bacon's syntax for Pythological:

```
Args (Cons a d) r: Cons (Ev a r) (Args d r)
```

in part because that cuts down awkwardly verbose lines like

```
(match (cons ,ta ,td) (cons ,pa ,pd) ,env) =>
    (match ,td ,pd (match ,ta ,pa ,env))
```

to more manageable things like

```
Match (Cons ta td) (Cons pa pd) env: Match td pd (Match ta pa env)
```

The concrete syntax here is something like this:

```
program: /\n/* definition* expression /[ \n\t]*/
definition: expression _ ":" expression "\n"+
expression: term /[ \t]+/ expression | term
term: _ symbol | _ var | _ "(" expression _ ")"
_: /[ \t]*/
var: /[a-z_][a-z_0-9]*/
symbol: /[A-Z][a-z_0-9]*/
```

A draft Qfitzah interpreter

Here's my current test input for my draft interpreter:

```
(NB x) x
(NB (Simple Test File For Qfitzah))
```

```
(Do (Cover x) it) (Some x Covered it)
(Do (Cover Leather) Sofas)

(NB (Higher Order Programming))
(Map f Nil) Nil
(Map f (Cons car cdr)) (Cons (Do f car) (Map f cdr))

(Map (Cover Chocolate) (Cons Police (Cons Raisins (Cons Oreos Nil))))

(NB (Boolean Definitions))
(Not Yes) No
(Not No) Yes
(If Yes a b) (Do a)
(If No a b) (Do b)
(Do Yes) Yes
(Do No) No
(And a b) (If a b a)
(Or a b) (If a a b)

(NB (This Should Be A Built-in))
(Eq Yes Yes) Yes
(Eq No No) Yes
(Eq Yes No) No
(Eq No Yes) No

(NB (Boolean Truth Tables))
(Well (Not Yes) (Not No))
(Well (And Yes Yes) (And Yes No) (And No Yes) (And No No))
(Well (Or Yes Yes) (Or Yes No) (Or No Yes) (Or No No))

(All Nil) Yes
(All (Cons a b)) (And (Do a) (All b))
(Any Nil) No
(Any (Cons a b)) (Or (Do a) (Any b))

(Test Yes) Ok
(Test No) Fail

(NB (Boolean Tests))
(Test (All (Cons (And Yes Yes) Nil)))
(Test (Not (Any (Cons (And Yes No) (Cons (And No Yes) (Cons (And No No) Nil))))))
(Test (All (Cons (Or Yes Yes) (Cons (Or Yes No) (Cons (Or No Yes) Nil)))))
(Test (Not (Or No No)))

(NB (Peano Arithmetic))
(= Z Z)           Yes
(= (S x) Z)       No
(= Z (S x))       No
(= (S x) (S y))  (= x y)
(+ Z x)           x
(+ (S x) y)       (+ x (S y))
(2)               (S (S Z))
(3)               (S (2))
(4)               (S (3))
(7)               (S (S (S (4))))
```

```
(Test      (= (+ (2) (2)) (4)))
(Test (Not (= (+ (2) (2)) (2))))
(Test (Not (= (2) (+ (2) (2)))))
(Test      (= (+ (3) (4)) (7)))

(Empty List Is ())
```

Upon being fed into my incomplete draft interpreter one line at a time (the interpreter has a bug when it can read more than one line at a time), it produces this output:

```
↪ ↪ (Simple Test File For Qfitzah)
↪ ↪ ↪ (Some Leather Covered Sofas)
↪ ↪ (Higher Order Programming)

↪ ↪ ↪ ↪ (Cons (Some Chocolate Covered Police) (Cons (Some Chocolate Covered Raisins) (Cons (Some Chocolate Covered Oreos) Nil)))
↪ ↪ (Boolean Definitions)
↪ ↪ ↪ ↪ ↪ ↪ ↪ ↪ ↪ ↪ (This Should Be A Built-in)
↪ ↪ ↪ ↪ ↪ ↪ (Boolean Truth Tables)
↪ (Well No Yes)
↪ (Well Yes No No No)
↪ (Well Yes Yes Yes No)
↪ ↪ ↪ ↪ ↪ ↪ ↪ ↪ ↪ ↪ (Boolean Tests)
↪ Ok
↪ Ok
↪ Ok
↪ Ok
↪ ↪ (Peano Arithmetic)
↪ ↪ ↪ ↪ ↪ ↪ ↪ ↪ ↪ ↪ ↪ ↪ Ok
↪ Ok
↪ Ok
↪ Ok
↪ ↪ (Empty List Is ())
↪
```

Here's the draft interpreter source code in amd64 assembly:

```
        ## Qfitzah, a leap or shortening: from a kilobyte
        ## or two of i386 machine code to a
        ## higher-order programming language with pattern matching,
        ## flexible parametrically-polymorphic data containers, and
        ## dynamically dispatched method calls with multiple dispatch.

        ## To build:

        ## $ gcc -Wl,-z,noseparate-code -static -m32 -nostdlib qfitzah.s -o qfitzah.bloated
        ## $ objcopy -S -R .note.gnu.build-id qfitzah.bloated qfitzah

        ## This version does *not* use bytecode.  But it should
        ## be a good estimate for how much
        ## code is needed for a Qfitzah (with some primitive operations
        ## such as addition).  358 instructions, 731 bytes of code, 40
```

```
## bytes of data, 1184 bytes of executable.

## (Brian Raiter's sstrip utility reduced the 1056-byte
## version of the executable to 828 bytes, a 228-byte
## reduction, but I don't yet have that built into the build
## process.)

## The m4 manual says, "An important precursor of m4 was GPM;
## see C. Strachey, 'A general purpose macrogenerator',
## Computer Journal 8, 3 (1965), 225-41,
## https://academic.oup.com/comjnl/article/8/3/225/336044. GPM
## is also succinctly described in David Gries's book Compiler
## Construction for Digital Computers, Wiley (1971).
## ...GPM fit into 250 machine
## instructions!"  Well, Qfitzah isn't quite that small yet,
## and it may not get there, but it's a lot more agreeable to
## program in than GPM.

## Using variables in a template that are not matched in the
## pattern is a bug that the interpreter doesn't detect,
## instead crashing:
## ↪ (Do Nothing) no
## ↪ (Do Nothing)
## Segmentation fault
##
## Using the same variable more than once is also an unchecked
## bug, but doesn't crash:
## ↪ (Eq x x) (Yes x)
## ↪ (Eq 3 3)
## (Yes 3)
## ↪ (Eq 3 4)
## (Yes 4)

## On calling conventions:

## My initial thought was to use the standard i386 Linux ABI
## (stemming from Microsoft's cdecl), where %ebx, %esi, %edi,
## %ebp, and of course %eip and %esp are preserved across
## calls, but %eax, %ecx, and %edx are not; but since my
## objective is to make things as small as possible, it was
## immediately obvious that passing all the parameters on the
## stack is unacceptably code-bloaty, so I was passing up to
## three parameters in %eax, %ecx, and %edx, and getting
## return values in %eax, and booleans in the flags.  So,
## within some limits, I was free to use %ebx, %esi, %edi, and
## %ebp as global variables.  I used %ebx as the allocation
## pointer, saving 12 bytes of code in the `cons` function,
## %esi as the parsing pointer, and %edi as the pointer to the
## global set of rewrite rules.

## However, I've come to the conclusion that this was a
## mistake, and callee-saved registers are generally not
## useful for i386 size optimization, especially not these.

## If you want to preserve a value across a call, you have two
```

```
## choices: you can push it on the stack (1 byte to push, one
## byte to pop) or you can allocate it in a callee-saved
## register.  But that means that you have to save and restore
## the callee-saved register at entry and exit, which costs
## the same 2 bytes; moreover, if you got the value by calling
## another function, you need an additional byte to xchg the
## value from %eax into the callee-saved register.  Also,
## pushing and popping lets you relocate it to a different
## register for free.  In theory using a callee-saved register
## could still be a win if you have multiple calls across
## which to preserve a value, but so far it never has been.
## And you have to weigh this dubious benefit against the
## benefit of having fewer registers available for
## temporaries: 3 temporaries, shared with arguments, is
## pretty cramped!

## Moreover, in the 8086, the only registers that could be
## used as pointers were %ebx, %ebp, %esi, and %edi; while the
## i386 removed this restriction, addressing with those
## registers still produces tighter code in some cases (like
## lodsb and I think base+index), though not in the simplest
## cases:

## 804816e:     89 48 04                mov    %ecx,0x4(%eax)
## 8048171:     89 4b 04                mov    %ecx,0x4(%ebx)
## 804812c:     8b 09                   mov    (%ecx),%ecx
## 804832a:     8b 13                   mov    (%ebx),%edx
## 804816c:     89 03                   mov    %eax,(%ebx)
## 8048358:     89 01                   mov    %eax,(%ecx)
## 8048358:     89 07                   mov    %eax,(%edi)

## Indexing off %esp instead of %ebp does cost an extra byte
## tho.

## Using a register like %ebx instead of a memory location for
## a global variable like the allocation pointer costs an
## extra byte of initialization, because the initialization
## has to be done with a MOV instruction.  Reserving %ebx in
## particular to be call-preserved also costs an extra
## push/pop pair around every system call, which is currently
## 4 bytes.

## So, for the time being, I've removed %ebx from the set of
## call-preserved registers, leaving only %ebp, %esi, %edi,
## and of course %esp and %eip.  Somewhat to my surprise, this
## initially made the code a byte *larger*, plus 4 bytes of
## data; but I quickly recovered the difference by using my
## shiny new temporary register.

## All three of these remaining registers have global usages:
## %esi is used during parsing as the parsing input pointer,
## %ebp is used as a base pointer to the global variables, and
## %edi is used as the text output pointer (so you can output
## a bytes with three bytes: `mov $'(, %al; stosb`.)
```

```
        ## Here's how we'll define procedures:
        .macro proc name
        .text 1                 # use subsection 1 for library code
\name:
        .endm

        ## Global variables are in the data segment, but in order to
        ## use smaller instructions, we load a pointer to the data
        ## segment into %ebp at startup and then never change it.  So
        ## far I'm not using this everywhere.
        .data
globals:
        .macro my name, initval
        .pushsection .data
\name:  .long \initval
        .popsection
        .endm

        ## Let's define a macro for things to do at startup.
        .macro init
        .text 0
        .endm

init
        .globl _start
_start: mov $globals, %ebp

        ## We can use this mechanism to reduce the byte weight of
        ## calls to frequently called functions.  The i386 lets you
        ## index into a function pointer table in an indirect-call
        ## instruction: call *4(%ebp), and that's only 3 bytes, while
        ## a direct call would be 5 bytes, even though it's
        ## PC-relative.  So with 3 calls, we can reduce 15 bytes of
        ## direct call instructions to 9 bytes of indirect call
        ## instructions and 4 bytes of address data, thus shaving 2
        ## bytes.
        .irp routine, cons, skip_whitespace, read_factor, subst, ev, match
        my \routine\()_address, \routine
        .endr

        ## By using a conditional macro for our calls, we can switch
        ## calls to a given routine between direct and indirect just
        ## by editing the list above:
        .macro do name
        .ifdef \name\()_address
        call *(\name\()_address-globals)(%ebp)
        .else
        call \name
        .endif
        .endm

        ## This interpreter is largely concerned with manipulating
        ## list structure.  Computers nowadays have large memories, so
        ## for any program that runs for a short time, perhaps under a
        ## second, we can get by without a garbage collector.  The
```

```
        ## fundamental procedure for constructing list structure is
        ## cons, which creates a pair.  It's wrapped in a macro here
        ## to facilitate putting it physically after a later procedure
        ## that falls through into it.
        my allocation_pointer, arena
        .macro cons_here
proc cons
        ## This is 11 bytes instead of 21 bytes thanks in part to
        ## replacing two giant 6-byte memory access instructions with
        ## 3-byte things that index off %ebp.
        push %edi
        mov allocation_pointer-globals(%ebp), %edi
        stosl                   # arg 1, the car, is already in %eax
        xchg %eax, %ecx         # arg 2, the cdr, is in %ecx
        stosl
        xchg %edi, allocation_pointer-globals(%ebp)
        xchg %edi, %eax         # return value (old allocation pointer) in %eax
        pop %edi
        ret
        .endm

        ## We're going to use pointer alignment to distinguish pair
        ## pointers from other kinds of pointers, including the empty
        ## list (nil); specifically, the low 2 bits of a pair pointer
        ## should be 0.  For this, we define a couple of macros for
        ## jumping if a register does or does not point to a pair.  To
        ## avoid wasting bytes, the \reg here should be a low-byte
        ## register: %al, %bl, %cl, or %dl.  %al in particular makes
        ## the `test` instruction 2 bytes instead of 3.
        .macro jpair reg, dest
        test $3, \reg           # will set ZF if it's a pair
        jz \dest
        .endm
        .macro jnpair reg, dest
        test $3, \reg
        jnz \dest
        .endm

        ## Extracting the fields of a pair:
        .macro car src, dest=none
        .ifeqs "\dest", "none"
        mov (\src), \src        # this is only 2 bytes
        .else
        mov (\src), \dest       # 2 bytes
        .endif
        .endm

        .macro cdr src, dest=none
        .ifeqs "\dest", "none"
        mov 4(\src), \src       # 3 bytes
        .else
        mov 4(\src), \dest      # 3 bytes
        .endif
        .endm
```

```
        ## Finally, we need some representation for the empty list,
        ## which needs to test as not being a pair.  This is a little
        ## tricky; the chosen value (1) tests as a constant, but
        ## attempting to fetch its print name will crash.
        .macro setnil reg
        xor \reg, \reg
        inc \reg
        .endm

        ## For such a simple allocator to work, we need a large arena;
        ## and the allocation pointer needs to be aligned in it.  We
        ## do this by aligning the arena to a 4-byte boundary and
        ## then incrementing the allocator pointer by multiples of 4.
        .bss 1
        .balign 4
arena:  .fill 512*1024*1024

        ## The other kinds of elements in our list structure are
        ## constants, such as uppercase symbols and numbers, which are
        ## represented by words ending in ...01, and variables, which
        ## are represented internally by words ending in ...10, and
        ## externally as lowercase identifiers.  So we have
        ## conditionals for these types corresponding to jpair and
        ## jnpair to test these tag fields:

        .macro jvar reg, dest
        test $2, \reg           # will clear ZF if it's a var
        jnz \dest
        .endm
        .macro jnvar reg, dest
        test $2, \reg
        jz \dest
        .endm

        ## XXX these aren't actually used:
        .macro jconst reg, dest
        test $1, \reg           # will clear ZF if it's a const
        jnz \dest
        .endm
        .macro jnconst reg, dest
        test $1, \reg
        jz \dest
        .endm

        ## So, let's define what it means to substitute some variables
        ## into a template:
        ## (define (subst t env)
        ##   (if (var? t) (cdr (assq t env))
        ##       (if (pair? t) (cons (subst (car t) env) (subst (cdr t) env))
        ##           t)))

        ## This is wrapped in a macro in order to enable us to
        ## physically put it further down, where `ap` can fall through
        ## into it.
```

```
        .macro subst_here
proc subst
        jnvar %al, 1f          # if t is not a var, jump ahead; otherwise
        do assq                # look it up with assq (inheriting args), then
        cdr %eax               # we take its cdr, then
        ret                    # return it
1:      jpair %al, 2f          # if t is not a pair, we just return it
        ret                    # (it's already in %eax)
2:      push %eax              # we must preserve argument t across a call
        push %ecx              # and env too.
        cdr %eax               # get (cdr t) for that recursive call,
        do subst               # which inherits env, but might clobber args;
        xchg %eax, %edx        # socking away its return value in %edx,
        pop %ecx               # restoring env for the second recursive call,
        pop %eax               # and also t, before

        push %edx              # saving the first subst return value on the stac
k;
        car %eax               # what we want to subst now is (car t)
        do subst               # so now our substed car is in %eax,
        pop %ecx               # and our substed cdr in %ecx, so we can
        jmp cons               # tail-call cons and return the result

        ## `assq` is our function for doing a variable lookup in an
        ## environment.  To avoid an extra unconditional jump, I've
        ## relocated the tail end of the loop to before the loop entry
        ## point, which has the bizarre effect of putting it before
        ## the *procedure* entry point.  It happens to set ZF when it
        ## succeeds and clear ZF when it fails, but `subst` ignores
        ## that at the moment.  It might make things simpler if it
        ## returned its key argument when it fails, like `walk` from
        ## µKanren?

2:      cdr %ecx               # go to the next item before falling into assq
proc assq                      # look up an item %eax in a dictionary %ecx
1:      jnpair %cl, 1f         # nil or another atom terminates the dict
        car %ecx, %edx         # get the item
        cmp %eax, 0(%edx)      # is our dictionary key this item?  CISC 4ever!1
        jne 2b                 # if not, restart the loop, or
        mov %edx, %ecx         # on success we return the item, or on failure
1:      xchg %ecx, %eax        # return the non-pair we were examining
        ret
        ## Possibly it would be better to inline `assq` as a macro
        ## inside `subst`, since that's the only thing that uses it so
        ## far.
        .endm                  # subst_here

        ## In a sense the inverse of `subst` is `match`.  If #(vt) is a var vt,
        ## `(subst '(You #(vt) my #(np)) '((#(np) . wombat) (#(vt) . rot)))`
        ## evaluates to `(You rot my wombat)`, as you'd expect if
        ## you're some kind of psycho stalker, while
        ## `(match '(You rot my wombat) '(You #(vt) my #(np)) '())`
        ## evaluates to `((#(np) . wombat) (#(vt) . rot))`.

        ## In Scheme:
```

```
        ## (define (match t pat env)
        ##   (if (var? pat) (cons (cons pat t) env)  ; vars match anything
        ##       (if (pair? pat)  ; pairs match if cars match and cdrs match
        ##           (and (pair? t)  ; a pair pattern can't match an atom
        ##               (let ((a (match (car t) (car pat) env)))  ; match car?
        ##                 (and a (match (cdr t) (cdr pat) a))))   ; then, cdr?
        ##           (and (equal? pat t) env))))  ; consts match only themselves

        ## In addition to returning an environment result in %eax,
        ## `match` also needs to indicate success or failure, which it
        ## does with ZF: ZF set indicates match ("equality"), ZF clear
        ## indicates match failure.  Switching from CF to ZF reduced
        ## the weight of this subroutine from 85 bytes to 76 bytes,
        ## and with further work it's down to 55.

        ## This has a bug; it treats () the empty list as a var. So
        ## (Gallygoogle ()) matches the same patterns (Gallygoogle x)
        ## would.

proc match
        ## Case for pattern being an unadorned var:
        jnvar %cl, 2f           # If the pattern is a var,
        xchg %eax, %ecx         # we want to (cons pat t), not (cons t pat)
        push %edx               # save env
        do cons
        pop %ecx                # now cons that pair onto the original env
        do cons
        xor %ecx, %ecx          # set ZF to indicate success
        ret

        ## Case for matching a non-pair against a pair pattern:
2:      push %ecx               # for recursion, we must save pattern and
        push %eax               # t, the term being matched.
        jnpair %cl, 2f          # ensure pattern is a pair;
        jnpair %al, 1f          # if term is not a pair, fail (clearing ZF);

        ## To match two pairs:
        car %eax                # take car of both the term
        car %ecx                # and of the pattern
        do match                # and allow env to inherit in a recursive call;
        jne 1f                  # if that failed we bail out;
                                # otherwise,
        xchg %eax, %edx         # put the resulting env in the third param
        pop %eax                # and get original term
        cdr %eax                # for second recursion with (cdr t), and likewise
        pop %ecx                # pat for
        cdr %ecx                # (cdr pat).

        jmp match               # tail-recursing; it's my result, right (ZF) or w
rong (!ZF)

        ## To match a constant pattern:
2:      cmp %ecx, %eax          # Only succeed on exact equality,
        mov %edx, %eax          # returning the supplied env
```

```
        ## Shared epilogue (XXX maybe unshare it?)
1:      pop %ecx                # discard 2 saved arguments
        pop %ecx
        ret

        ## ev evaluates a term by first evaluating all its children
        ## with evlis (which is just `(map ev t)`), then invoking
        ## ap(ply) on the result.
        ## (define (ev t)
        ##   (if (pair? t) (ap (evlis t) rules) t))
        ## (define (evlis t)
        ##   (if (pair? t) (cons (ev (car t)) (evlis (cdr t))) t))
proc evlis
        jpair %al, 1f
        ret
1:      push %eax               # save original t
        cdr %eax
        do evlis
        pop %ecx                # restore original t
        push %eax               # save return value
        xchg %ecx, %eax         # 1 byte - shorter than mov %ecx, %eax
        car %eax
        do ev
        pop %ecx                # pass evlis return value as cdr arg to cons
        # FALL THROUGH into cons (tail call)
        cons_here

        ## I'm thinking I'll provide primitive procedures for
        ## arithmetic and file I/O by way of terms whose head is the
        ## integer "0".  For example: integer subtraction.  Here we
        ## have the term in %eax.  This untested strawman evprim
        ## weighs 25 bytes, plus 7 bytes for the test and branch in
        ## ev.
proc evprim
        cdr %eax
        car %eax, %ebx
        cdr %eax
        cmp $5, %ebx  # (0 0 x y) returns x - y assuming both are ints
        jnz 1f
        car %eax, %ebx
        cdr %eax
        car %eax
        sub %ecx, %eax      # XXX is this backwards?
        or $5, %al          # low-order bits got zeroed by subtraction
1:      ret

        my rules, -1            # global set of rules, initially nil
proc ev
        jpair %al, 1f
        ret                     # atoms always evaluate to themselves
1:      do evlis
        car %eax, %ecx                # check for primitive invocation
        cmp $5, %ecx                  # is the car of the list (tagged) 0?
        jz evprim
        mov rules-globals(%ebp), %ecx # initial rules argument to ap: the global
```

```
        # FALL THROUGH to ap

        ## (define (ap t rules)
        ##   (if (not (pair? rules)) t   ; no rewrite rules left? don't rewrite
        ##       (let ((m (match t (caar rules) '()))) ; initially empty env
        ##         (if m (ev (subst (cdar rules) m)) ; matched? subst & eval
        ##             (ap t (cdr rules))))))        ; otherwise, try others
proc ap
        jpair %cl, 1f
        ret                     # return input t

1:      car %ecx, %edx          # get first rule
        push %eax               # save input t
        push %ecx               # save input rules
        car %edx, %ecx          # get pattern part of rule
        setnil %edx
        do match                # see if this rule matches inherited t in %eax
        je 1f                   # if that succeeded, go to the success case; or

        pop %ecx
        pop %eax

        cdr %ecx                # move on to next rule and tail-recurse
        jmp ap

        ## Now we have found a match, with the env in %eax; now we
        ## must invoke subst with the template, then return the
        ## instantiated template.
1:      mov %eax, %ecx          # template is subst's second argument
        pop %eax                # load saved rules
        car %eax
        cdr %eax
        pop %edx                # discard saved input t
        do subst
        jmp ev

        subst_here                      # XXX no longer necessary to be here

        ## Here are some macros from httpdito:
        .equiv __NR_exit, 1     # linux/arch/x86/include/asm/unistd_32.h:9
        .equiv __NR_read, 3
        .equiv __NR_write, 4

        ## System calls with different numbers of arguments.
        ## `be x, y` is a macro that does `mov x, y` or equivalent.
        .macro sys3 call_no, a, b, c
        be \c, %edx
        sys2 \call_no, \a, \b
        .endm

        .macro sys2 call_no, a, b
        be \b, %ecx
        sys1 \call_no, \a
        .endm
```

```
        .macro sys1 call_no, a
        be \a, %ebx
        sys0 \call_no
        .endm

        .macro sys0 call_no
        be \call_no, %eax
        int $0x80
        .endm

        ## Set dest = src.  Usually just `mov src, dest`, but sometimes
        ## there's a shorter way.
        .macro be src, dest
        .ifnc \src,\dest
        .ifc \src,$0
        xor \dest,\dest
        .else
        .ifc \src,$1
        xor \dest,\dest
        inc \dest
        .else
        .ifc \src,$2
        xor \dest,\dest
        inc \dest
        inc \dest
        .else
        mov \src, \dest
        .endif
        .endif
        .endif
        .endif
        .endm

        ## To read input, we need an input buffer; to intern atoms, we
        ## need someplace to put the atom base+length pairs.
        .bss
input_buffer:
        .fill 65536
        .balign 8   # atoms need to be 8-byte aligned to free tag bits
atoms:  .fill 8192
        my inptr, input_buffer
        ## Output is handled by setting %edi to point into this output
        ## buffer, then using stosb to add stuff to it.
        .bss
outbuf: .fill 131072

init
        mov $outbuf, %edi

        .data
prompt: .ascii "↪ "
prompt_end:
init
repl:   mov $prompt, %eax
        mov $(prompt_end - prompt), %ecx
```

```
        do emit
        do flush
        sys3 $__NR_read, $0, inptr, $255
        test %eax, %eax         # EOF on input?
        jz quit
        ## XXX missing loops for \n; could be multiple lines or partial lines
        mov inptr-globals(%ebp), %esi # copy old inptr to %esi for parsing
        add %esi, %eax  # NUL-termination unnecessary due to zero fill
        mov %eax, inptr-globals(%ebp)
        do handle_line
        jmp repl
quit:   sys1 $__NR_exit, $0

        ## XXX this needs a lot of attention for reducing code space
proc print
        cmp $1, %eax    # treat nil like pairs (cmp is only 3 bytes!)
        je 5f
        jnpair %al, 1f          # non-nil atoms treated otherwise
5:      push %eax               # save S-expression to print
4:      mov $'(, %al
        stosb
        pop %eax
        ## loop over list items:
2:      jnpair %al, 3f          # XXX handle improper lists?
6:      push %eax
        car %eax
        do print
        pop %eax
        cdr %eax
        jnpair %al, 3f
        push %eax
        mov $32, %al
        stosb
        pop %eax
        jmp 6b                  # XXX too many jumps?
3:      mov $'), %al
        stosb
        ret
1:      and $~3, %eax        # convert var/constant → base/len pointer
        cdr %eax, %ecx
        car %eax
        ## FALL THROUGH into a tail call to `emit`

proc emit                       # output a string to output buffer
        push %esi
        mov %eax, %esi          # string base is arg 1, length is arg 2 (%ecx)
        rep movsb
        pop %esi
        ret

proc flush                      # Send output buffer to actual stdout
        mov $outbuf, %ecx       # base address of bytes to output
        push %ecx
        mov %edi, %edx
        sub %ecx, %edx          # number of bytes to output
```

```
        sys1 $__NR_write, $1
        pop %edi                # reset output pointer
        ret

        ## Our grammar looks something like:
        ## prog ::= _ (factor (_ "\n" | _ factor _"\n"))*
        ## factor ::= constant | var | "(" (_ atom)* _ ")"
        ## _ ::= " "*
        ## constant ::= [*-^][*-~]*
        ## var ::= [_a-~][*-~]*
        ##
        ## Constants and vars chew up as many characters as they can.
        ##
        ## A line with two S-expressions ("factors") defines a rule; a
        ## line with just one offers an expression to evaluate
        ## according to the rules so far.

        ## On inputting a rule, it is added
        ## to the front of the rules list (thus taking precedence over
        ## older rules).

        ## Here's a crude parser.  Input pointer in %esi points
        ## into NUL-terminated input string.
proc handle_line
        cld       # XXX not really necessary since DF is always clear
        do read_factor
        jz 1f                   # if blank line, ignore
        ret
1:      push %eax
        do skip_whitespace
        lodsb   # lodsb;dec: 2 bytes, cmp $'\n, %al: 2; cmp $':, (%esi): 3
        dec %esi # so this approach saves bytes only because of second cmp
        cmp $'\n, %al # if there's only one expression on the line, not a rule
        jnz 1f
        pop %eax
        do ev
        do print
        mov $'\n, %al
        stosb
        ret
1:      do read_factor # read replacement template for rule being defined
        pop %ecx                # pop pattern
        jnz parse_error
        ## XXX ignoring the possibility of more than two things on the line
        xchg %ecx, %eax
        do cons
        ## FALL THROUGH into tail call to add_rule

proc add_rule
        mov rules-globals(%ebp), %ecx # cons onto the existing set of rules
        do cons
        mov %eax, rules-globals(%ebp)
        ## debug print out rules:
        ## do print
        ## mov $'\n, %al
```

```
        ## stosb
        ## do flush
        ret

proc parse_error
        mov $'!, %al
        stosb
        mov $'\n, %al
        stosb
        ret

proc read_factor
        do skip_whitespace
        lodsb
        dec %esi                # peeking
        cmp $'(, %al            # Is there a nested list?
        jne 1f
        lodsb
        do read_term
        push %eax
        do skip_whitespace
        lodsb
        cmp $'), %al            # indicate success
        pop %eax
        ret
1:      cmp $'*, %al            # [*-^] starts a constant
        ## <https://stackoverflow.com/a/29577037> explains that with
        ## cmp $2, %eax, jg jumps when %eax > 2, though this is
        ## confusing as
        ## <https://en.wikibooks.org/wiki/X86_Assembly/Control_Flow>
        ## explains; so this comparison has the correct sense:
        jb 3f
        cmp $'^, %al
        ja 2f
        jmp read_constant
2:      cmp $'_, %al            # _ starts a variable
        je 2f
        cmp $'a , %al           # [a-~] also starts a variable
        jb 3f
        cmp $'~, %al
        ja 3f
2:      jmp read_var
3:      cmp $'_, %al # guaranteed to fail and clear ZF, indicating failure.
        ret

        ## Advance input pointer %esi into NUL-terminated input string
        ## to first non-whitespace character.
proc skip_whitespace
        lodsb
        cmp $32, %al
        je skip_whitespace
        dec %esi
        ret

        ## Always succeeds (possibly returning nil), doesn't set ZF.
```

```
proc read_term
        do read_factor
        jnz 1f                  # if it failed, skip ahead
        push %eax               # save returned term
        do read_term            # recursive call for tail of term
        push %eax
        do skip_whitespace
        pop %ecx
        pop %eax
        do cons                 # XXX tail call
        ret
1:      setnil %eax             # return nil if no factor found
        ret

        ## Always succeeds, sets ZF to indicate success.
proc read_constant
        do read_atom
        ## xor $3, %al is also 2 bytes, same as or $1, %al
        ## So XXX maybe one of these two should fall through into
        ## intern and the other should xor the output of intern with 3
        or $1, %al
        cmp %eax, %eax          # set ZF
        ret

        ## Always succeeds, sets ZF to indicate success.
proc read_var
        do read_atom
        or $2, %al
        cmp %eax, %eax
        ret

        ## Octal to tagged integer.  Digit count in %ecx, input starts
        ## at %esi.  19 bytes.  Not used yet.
proc o2ti
        xor %eax, %eax
        xor %ebx, %ebx          # accumulate result in %ebx
1:      lodsb
        sub $'0, %al            # convert digit from ASCII
        add %eax, %ebx
        shl $3, %ebx     # by shifting after the add instead of before,
        loop 1b
        xchg %eax, %ebx
        add $5, %eax            # we leave space for this type tag
        ret

        ## Decimal to tagged integer.  Digit count in %ecx, input
        ## starts at %esi.  22 bytes.  Not used yet.  If the
        ## difference is only like 6 bytes maybe I'll just use
        ## decimal.
proc d2ti
        xor %eax, %eax
        xor %ebx, %ebx          # accumulate result in %ebx
1:      lodsb
        sub $'0, %al            # convert digit from ASCII
        imul $10, %ebx
```

```
        add %eax, %ebx
        loop 1b
        xchg %eax, %ebx
        shl $3, %eax
        add $5, %eax
        ret

        ## Tagged integer to octal, taking integer in %eax.  Outputs
        ## to buffer at %edi.  25 bytes.  Not used yet.
proc ti2o
        xchg %eax, %ebx
        xor %eax, %eax          # clear high bytes of %eax for the loop
1:      shr $3, %ebx            # shift first to remove type tag
        mov %bl, %al            # still only 2 bytes!
        and $7, %al             # 2 bytes, shorter than `and $7, %eax`
        add $'0, %al            # convert to ASCII
        test %ebx, %ebx         # don't recurse if no digits remain
        jz 1f
        push %eax               # buffer up digit for later emission
        call 1b
        pop %eax
1:      stosb
        ret

        ## I'm thinking about adding character string literals, in a
        ## form like this maybe.  This function would be called after
        ## the open-quote.  51 bytes.
proc read_string
        xor %eax, %eax
        lodsb
        cmp $'", %al
        jne 1f
2:      xor %eax, %eax          # tagged integer 0 as list terminator
        mov $5, %al # this is 4 bytes rather than the 5 of mov $5, %eax
        ret
1:      cmp $'\n, %al
        je 2b                   # just treat this as end of string
        cmp $'\\, %al           # treat \" as embedded "
        jne 1f
        lodsb
        cmp $'\n, %al
        je 2b
1:      push %eax
        do read_string          # read rest of string
        pop %ecx
        xchg %eax, %ecx         # get saved character back in %eax
        shl $3, %eax
        or $5, %al              # add integer tag
        do cons
        xchg %eax, %ecx
        xor %eax, %eax
        mov $13, %al            # tagged integer 1
        jmp cons # XXX probably place this closer to cons so this jump is short

        ## Always succeeds.
```

```
proc read_atom
        mov %esi, %edx          # save address of start byte
1:      lodsb
        cmp $'*, %al
        jb 1f
        cmp $'~, %al
        jbe 1b
1:      dec %esi                # put back last character read
        mov %esi, %ecx          # save end address, then compute length:
        sub %edx, %ecx          # $ecx -= $edx, due to confusing AT&T syntax
        xchg %eax, %edx
        ## FALL THROUGH into intern

        ## (intern base-addr len) checks to see if a string is already
        ## in the atom table, returning it if so, or inserting it if
        ## not; either way it returns the (4-byte-aligned) address.
        ## Can't fail.  Can't use assq because it's doing a string
        ## compare.
proc intern
        push %esi
        push %edi
        mov $atoms-8, %ebx
        ## At the top of the following loop, %ebx points into (or just
        ## before) the atoms table, %eax points to the string we're
        ## trying to intern, and %ecx has its length.
1:      add $8, %ebx            # Advance to next table entry.
        mov (%ebx), %edi        # Load string pointer from table.
        test %edi, %edi         # null string pointer? indicates end of table
                                # test %edi, %edi is one byte smaller than cmp $0.
        jz 2f                   # reached end of table without finding it
        cmp %ecx, 4(%ebx)       # check to see if the lengths match
        jne 1b
        push %ecx               # repe will clobber %ecx
        mov %eax, %esi          # put pointer to needle string into %esi
        repe cmpsb
        pop %ecx
        jne 1b                  # go on to next entry unless we found it
1:      mov %ebx, %eax          # address of found entry (table pointer)
        pop %edi
        pop %esi
        ret
2:                      # not found, must insert
        mov %eax, (%ebx) # non-null pointer here says this is no longer the end
        mov %ecx, 4(%ebx)
        jmp 1b                  # now that we've inserted it, it's "found"
```

Here's the resulting 956-byte executable in base64:

f0VMRgEBAQAAAAAAAAAAAAIAAwABAAAAuIAECDQAAAAAAAAAAAAAAADQAIAADACgAAAAAAAEAAAAA
AAAAAIAECACABAiTAwAAkwMAAAUAAAAAEAAAAQAAAJQDAACUkwQIlJMECCgAAAAsIAMgBgAAAAAQ
AAAEAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAEAAAABAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAL2UkwQIv8CzBQi4uJMECLkEAAAA6FYBAADoWAEAALr/AAAAiw2OkwQIMdu4
AwAAAM2AhcB0D4t1IAHwiUUg6EkBAADrxTHbuAEAAADNgPbBAnQMkVL/VQBZ/1UAMcnDUVD2wQN1
GKgDdRiLAIsJ/1UUdQ+SWItABFmLSQTr0DnIidBZWcOoA3QBw1CLQATo8v///1lQkYsA/1UQWVeL
fRirkauHfRiXX8OLQASLGItABIP7BXULixiLQASLACnIDAXDqAN0AcPouf///4sIg/kFdNaLTRz2

wQN0AcOLEVBRiwox0kL/VRROB1lYi0kE6+WJwViLAItABFr/VQzrw6gCdAnoIwAAAItABMOoA3QB
w1BRi0AE/1UMkllYUosA/1UMWel0////i0kE9sEDdQiLETkCdfKJ0ZHDg/gBdASoA3UkULAoqlio
A3UXUIsA6Ob///9Yi0AEqAN1B1CwIKpY6+mwKarDg+D8i0gEiwBWicbzpF7DucCzBQhRifopyjHb
Q7gEAAAAzYBfw/z/VQh0AcNQ/1UErE48CnUNWP9VEOiU////sAqqw/9VCFl1DpH/VQCLTRz/VQCJ
RRzDsCGqsAqqw/9VBKxOPCh1D6zoKwAAAFD/VQSsPClYwzwqchQ8XncC6zE8X3QIPGFyBjx+dwLr
LTxfw6w8IHT7TsP/VQh1EFDo9f///1D/VQRZWP9VAMMxwEDD6IQAAAAMATnAw+h6AAAADAI5wMMx
wDHbrCwwAcPB4wPi9pODwAXDMcAx26wsMGvbCgHD4vaTweADg8AFw5MxwMHrA4jYJAcEMIXbdAdQ
6O3///9YqsMxwKw8InUFMcCwBcM8CnT3PFx1Baw8CnTuUOjh////WZHB4AMMBf9VAJExwLAN6fv9
//+J8qw8KnIEPH52906J8SnRklZXu7iTBQiDwwiLO4X/dBI5SwR18lGJxvOmWXXqidhfXsOJA4lL
BOv0AFSBBAiwggQIfYIECLeBBAh6gQQIBoEECMCzBwj/////wJMECOKGqiA=

Topics

- Performance (p. 1155) (22 notes)
- Bootstrapping (p. 1171) (12 notes)
- Lisp (p. 1174) (11 notes)
- Assembly-language programming (p. 1175) (11 notes)
- Small is beautiful (p. 1190) (8 notes)
- Programming languages (p. 1192) (8 notes)
- Higher order programming (p. 1196) (7 notes)
- Syntax (p. 1221) (5 notes)
- Bytecode (p. 1236) (5 notes)
- Dynamic dispatch (p. 1259) (4 notes)
- Term rewriting (p. 1270) (3 notes)
- Scheme (p. 1274) (3 notes)
- Qfitzah (p. 1337) (2 notes)
- Aardappel

A short list of the most useful Unix CLI tools

Kragen Javier Sitaker, 02021-09-15 (updated 02021-09-16)
(2 minutes)

A list of handy tools (mostly CLI) developed from a discussion on #scannedinavian on Libera.chat (formerly Freenode).

- `most`, `less`, `bat`: text viewing
- `vi`: text editing
- `cat`, `cut`, `join`, `comm`, `sort`, `merge`, `uniq`, `grep`, `shuf`, `random`: basic text query
- `ack`, `rg`, `awk`, ack-grep: advanced text query (sometimes filesystem too)
- `mpv`, `mplayer`, `evince`, `mupdf`, `xpdf`, `firefox`, `links`, `lynx`, `ebook-viewer`: non-text viewing
- `find`, `xargs`: filesystem query
- `fzf`, `fd`: advanced filesystem query
- `jq`, JSON query
- `history`: bunk
- `sudo`: sandwiches
- `ls`, `cd`, `mkdir`, `rmdir`, `mv`, `ln`, `rm`, `cp`, `du`: basic file manager
- dust: alternative to du
- `exa`: alternative to ls
- `xdu`, `k4dirstat`: disk usage explorers
- `ls`, `lsof`, `ps`, `w`, `who`, `ip`, `ifconfig`, `netstat`, `df`, `vmstat`, `dstat`, `top`, `htop`: system status information
- `thunar`, `bashmount`, `mkfs.vfat`: mounting and formatting removable media
- `sudo`, `apt`, `apt-get`, `apt-cache`, `ifconfig`, `dpkg`, `apt-file`, `mtr`: package management and system administration
- `ssh`, `nc`, `rsync`, `youtube-dl`, `gomuks`, `mtr`, httpdito: networking
- `rsync`, `diff`, `patch`, `cmp`: file comparison and synchronization
- `git`, `fossil`, `rcs`: source code version tracking
- `rsync`, `git`, `tar`: backups
- `%`, `bg`/`%&`, `jobs`, `kill`, `killall`, `pidof`: multitasking
- `time`, `tally`, `ulimit`, `ps`, `top`, `docker`, `lsof`, `pv`: resource monitoring and administration
- `man`, `info`, `pydoc`: online help
- `[`, `[[`, `while`, `if`, `read`, `sleep`, `$((..))`, `echo`: shell scripting
- `caesar`: toy encryption
- `units`, `bc`, `dev3/calc.py`: calculation
- `bc`, `python2`, `python3`, `awk`: programming languages
- `TZ=America/New_York date`: world clock
- `printf '\e[3gf(\eHx,\n\ty)\n'`: escape sequences
- `watch -d`: GUIfication
- `reset`: recovering a horked terminal
- `sloccount`, `cloc`: counting lines of source
- `asciinema`, `screen`, `tmux`: terminal recording, disconnecting, multiplexing
- `visidata`: data explorer
- `darkdraw`: ASCII art draw program
- `dd`: sorry, I can't, tonight I'm driving

• https://github.com/eradman/entr: run arbitrary commands when files change, turning your filesystem into an event bus (originally just a test runner)
• `w`, `who`, `write`, `wall`, `talk`: a chat system
• `xclip`, `xmessage`: sort of GUI integration

Thanks to Shae Erisson and for the discussion this arose from!

Topics

• History (p. 1153) (24 notes)
• Unix (p. 1268) (3 notes)
• Command-line interfaces (CLI) (p. 1378) (2 notes)

Three phase differential data

Kragen Javier Sitaker, 02021-09-22 (updated 02021-12-30)
(4 minutes)

Single-ended data transmission, as used in RS-232, is limited in noise immunity, so higher-speed data is commonly transmitted with differential signaling. Unfortunately, this requires twice as much wire.

AC power transmission faces a similar problem: unless you're willing to use the planet as your current return path (commonplace in parts of Brazil and for submarine cables) you need to carry the current from the power plant to the load — and then back again. In AC power transmission, this problem (as well as the problem of starting large AC motors) is solved with three-phase power transmission: three wires carry alternating voltages 120° out of phase with one another. The sum of the three phasors is zero: always approximately, and for balanced loads like a three-phase motor, exactly. This permits carrying thrice as much power over only 50% more wires than the single-phase approach used in household outlets.

If you replace your twisted-pair data transmission lines with twisted triads, for 50% more wires you can't get *thrice* the data rate, but you *can* get *twice* the data rate, with roughly the same noise-immunity benefit you get from twisted-pair differential signaling. The third wire carries a balancing voltage and current which maintains the triad's net voltage and current at 0; the mutual inductances and capacitances of the three wires remain constant along their lengths, just like in a twisted pair, and because these mutual inductances and capacitances are are all equal, crosstalk can be accurately canceled. At low speeds this balancing voltage can be arranged straightforwardly by driving the third wire with an opamp that maintains a constant reference voltage at a summing Y junction; higher speeds might require more elaborate drivers that multiplex the third wire among, in the bilevel case, four compensation voltages.

Attempts to extend this approach to differential quads, quintuplets, and so on will suffer from the fact that the balancing wire and signal wires cannot all be equidistant, at least in three-dimensional space. The result is that either data edges on one signal wire that couple equally to two other signal wires cannot be perfectly canceled on both of them by a single balancing wire, or data edges on one signal wire that couple *unequally* to two other signal wires cannot be perfectly canceled on both of them by a single balancing wire. Also, if the voltage levels on the three or more signal wires were uncorrelated, the balancing wire would be subject to much larger voltage swings. Still, MIMO approaches could yield fruit with n wires while retaining net zero voltage and current on the wire bundle; in theory it's just a question of solving a well-conditioned, very nearly linear system of n equations in n unknowns, one of which is the common-mode noise.

The problem of unequal voltage swings and complex analog line drivers can be solved, at the cost of reduced data rates, with a constant-weight m-of-n code like the 2-of-5 code. In a sense the 1-of-3 encoding SETUN used for its three-way flip-flop(-flap?)s was

a constant-weight encoding providing 1.58 bits over three “differential” wires. Decoding such a constant-weight code could be considerably simpler than a more sophisticated MIMO approach, merely amounting to comparing each wire to a threshold from a summing junction; a balanced code on 8 wires would provide 6.1 bits of data per clock (70 valid codes), and a 9-of-19 code on 19 wires would provide 15.9 bits per clock (8314020 valid codes).

More generally, if you can statically estimate (linear, memoryless) crosstalk coefficients between particular wires, you can build a summing junction that computes the “predicted” crosstalk for each wire, to be used as the comparison threshold for that wire, thus mostly canceling that crosstalk.

Topics

- Electronics (p. 1145) (39 notes)
- Protocols (p. 1206) (6 notes)
- Physical computation (p. 1208) (6 notes)
- Encoding (p. 1256) (4 notes)
- Setun (Сетунь)

Waterglass “Loctite”?

Kragen Javier Sitaker, 02021-09-22 (updated 02021-12-30) (1 minute)

Loctite polymerizes in the presence of metal ions, which is how it hardens in the screw threads but not in the plastic bottle. Guess what else solidifies in the presence of (polyvalent) metal ions? Solutions of soluble silicates: waterglass!

Suppose waterglass by itself in a screw thread doesn’t harden because the metal surface doesn’t spontaneously oxidize just because it’s wet? Even if galvanic corrosion isn’t a viable source of cations, maybe the surface can be induced to oxidize with some kind of nasty oxidizing anions. For example, maybe you can use sodium chloride, potassium sulfate, ammonium acetate, or some crossbreed of them to provoke oxidation of iron, brass, or aluminum. Only a tiny amount of oxidation is necessary, only enough iron ions to precipitate the silicate, which itself should be present in only the minimal quantity required to achieve the desired bond strength.

Another interesting possibility I haven’t tried is baking soda. Waterglass hardens in seconds upon contact with carbonic acid, and baking soda releases carbonic acid upon being heated above 50°, leaving behind its conjugate base, the more basic soda ash. Perhaps this reaction could solidify waterglass in a thread if you heat the whole assembly above 50°? I suspect that the carbonic acid solidifies the waterglass by dropping the pH, and if that’s the case, it will not work.

Topics

- Manufacturing (p. 1151) (29 notes)
- Waterglass (p. 1189) (8 notes)

Qfitzah internals

Kragen Javier Sitaker, 02021-09-24 (updated 02021-12-30)
(2 minutes)

Qfitzah represents terms as cons lists in the conventional Lisp way, exploiting the isomorphism between ordered trees and binary trees. A term may be a constant, a variable, or a list; a list is represented as a pair of the first item in the list, which may be any term, and the rest of the list, which is either a pair or nil. Additionally, variables and constants are symbols. I'm planning to add a new kind of constant that is an unboxed small integer and not a symbol.

Each of the values in a pair is represented by a 32-bit word, and the pair is represented by two adjacent 32-bit words in memory; its three low-order bits represent the type tag of the object pointed to. A pair is represented by a pointer to the beginning of that pair, and its low-order bits are ?00, 000 I think.

Variables (represented in the concrete syntax as symbols beginning with an uncapitalized letter or their ilk) are represented by words whose low-order bits are 010, and constants are represented by words whose low-order bits are 001. I'm planning to make numerical constants be represented by words whose low-order bits are 101. Nil is represented by -1, so its low-order bits are 111, so if you were expecting a term and got nil, it would look like both a variable and a constant. XXX this is a bug! There are bugs where empty lists are concerned. Shit. They match things they shouldn't, for example:

```
$ ./qfitzah
↬ () (W)
↬ (F)
Violación de segmento (`core' generado)
```

If you mask off the two low-order bits of the word that represents a symbol, you get the address of a 32-bit word containing a pointer to the characters of the symbol's name; the following 32-bit word is the length of the symbol's name. These records are allocated sequentially in the atom table, which is initially filled with zeroes.

My thought is that if you shift an unboxed integer by 3 bits you get the integer value.

Topics

- Term rewriting (p. 1270) (3 notes)
- Qfitzah (p. 1337) (2 notes)

Blowing agents

Kragen Javier Sitaker, 02021-09-29 (updated 02021-12-30)
(4 minutes)

Pentane and CFCs are conventionally used for blowing styrofoam and similar low-temperature materials; propane is used for aerosol cans. For bread, yeast produces carbonic acid gas, and heating produces water vapor, while in pancakes, cookies, and crackers, baking powder produces carbonic acid gas from something like cream of tartar (potassium bitartrate) and baking soda, or for a double-acting powder, something like monocalcium phosphate and baking soda. Salt of hartshorn (ammonium carbonate and carbamate) similarly decomposes into ammonia, water, and carbonic acid gas at around 58°.

At around 100°, water evaporates. It has the unique advantage of weighing only 18 daltons, so a given mass of water will produce more moles of vapor than just about any other condensed matter, except that ammonia is only 17 daltons. By comparison, dioxygen is 32 daltons, dinitrogen 30, and carbonic acid gas 28.

But what about higher temperatures? For blowing foams from polymers like waterglass which require higher temperatures to soften, we'd maybe benefit from solids that don't become reactive until those higher temperatures. For waterglass in particular it would be desirable that the blowing agents not release polyvalent cations, since those can raise the softening point of the waterglass to an inconveniently high temperature.

Various kinds of water-including complexes might work as fillers that can retain water to higher temperatures. Phosphoric acid retains some water up to 800°, although it's probably not very compatible with waterglass; and its melting point drops rapidly from 40° with water content. But sal mirabilis, epsom salt, blue vitriol, alabaster, and the chlorides of magnesium and calcium incorporate quite a lot of water in their crystals, some of which evaporates at temperatures somewhat above 100°. Of these, sal mirabilis is free of polyvalent cations, and alabaster is relatively insoluble in water.

Baking soda alone evolves carbonic acid gas upon heating past 50° and fairly quickly at 100°, leaving washing soda, which I believe releases a second carbonic acid molecule at above 850°. This, however, leaves behind sodium oxide, which may not be desirable.

Chalk, of course, also releases carbonic acid at 500°-850°, leaving behind quicklime.

The formate and chloride of ammonium both decompose into gas upon heating (to 180° with traces of hydrogen cyanide in the first case), but in the case of the chloride the process is reversible. I think that the oxalate also decomposes to gases at 215°-265°, producing carbon monoxide as well. Oxalates and formates of metals also tend to produce gases when heated; those of lithium, calcium, sodium, potassium, and magnesium are notable here.

Aluminum trihydroxide ("trihydrate", gibbsite) is commonly used as a flame-retardant plastic filler. It's fairly alkaline, but highly

water-insoluble, and does contain polyvalent cations. Upon heating past 220° it produces a mole and a half of water, leaving amorphous sapphire. Magnesium (di)hydroxide is similar, but doesn't decompose until 330°, leaving magnesia; and slaked lime of course does the same thing, but at 400°-600°, leaving quicklime. Slaked lime is unstable in atmospheric air because it slowly absorbs carbonic acid gas to become chalk.

Sulfur boils at 444.6°, but it melts first and then polymerizes, and the vapor consists of mostly violet disulfur (64 daltons) above 720°; at lower temperatures as much as 10% may be red trisulfur. Ignition in air happens at a much lower temperature than this boiling. Sulfur has the great advantage that it is not water soluble and pretty inert at room temperature.

Autoclaved aerated concrete uses a reaction between powdered aluminum and water to generate hydrogen (2 daltons!) but of course the total mass of the ingredients is much larger than if you were to just boil the water.

Metal hydrides might be an alternative hydrogen source, though calcium hydride (hydrolith) doesn't even melt until 816°, and even LiH (8 daltons!) doesn't melt until 688.7°. These readily produce hydrogen gas with water, forming the hydroxides. Thermal decomposition of LiH is feasible, but the resulting lithium metal doesn't boil until 1330°.

Topics

- Materials (p. 1138) (59 notes)
- Manufacturing (p. 1151) (29 notes)
- Foam (p. 1185) (9 notes)
- Azane (p. 1386) (2 notes)

Compliance spectroscopy

Kragen Javier Sitaker, 02021-09-29 (updated 02021-12-30)
(4 minutes)

As I understand it, compliance is the derivative of position with quasistatic force, and in a linear system this is a constant. Nonlinear systems have compliance that varies with position; think of a probe touching skin with subcutaneous fat over muscle and bone. Compliance is high at first because of the fat, but with larger displacements the fat is squashed out enough that now you're measuring the compliance of the muscle, which increases rapidly with further displacement. I suspect that this kind of force-displacement curve is important to the tactile sensations of softness, hardness, etc.

Generally I think that, if you want to plot the curve, you want to plot the *derivative* of force against displacement, which is to say, the compliance as a function of displacement.

Along a second dimension, we have the frequency at which the force is being applied; for some systems, resonant modes will give very large "compliances" at some frequencies and smaller ones at other frequencies, tending to the quasistatic limit as the frequency approaches zero. The degree to which the system exhibits these resonances gives you some information about lossiness in its elasticity. A steel spring might have the same compliance as a cotton pillow, and you can taper the spring to make its compliance vary with displacement (as mattress makers do routinely), but the cotton pillow will lack any sharp resonant peaks because it's very lossy, so you can easily distinguish them. This is noticeable if you just tap both systems with a coin. Other systems have other characteristic resonances. Nonlinearity will produce vibrations at other frequencies, including harmonics and subharmonics, vaguely like Raman emission but in the acoustic domain, potentially adding a third dimension.

This sort of "viscoelasticity spectroscopy" or "compliance spectroscopy" is potentially useful for a number of different purposes:

- Teledildonics. Nonlinear compliance is crucial to reproducing the tactile sensation of touching a human body.
- Somatic and haptic interfaces; by distinguishing a finger pressing a button from a palm or a floor pressing it, software can distinguish between different actions to take. Even pressing a button with the same finger at different angles can be detected. Tiny total internal reflection compound parabolic reflectors could be integrated into a button to provide a simultaneous optical interface for coupling LED illumination in and out of a finger without the discomfort occasioned by the LED bumps traditional in pulse oximeters. This, too, has obvious masturbatory uses: a vibrator can be programmed to respond to not only the pressure it's put under but also the compliance curve of the tissues it's stimulating.
- Material identification. This is somewhat trickier, because the compliance spectrum of an object depends on many things other than the material it's made of; for example, what shape it is, how firmly it's being held, and what it's being held in. But with the whole 2-D

spectrum, it might be feasible to tease out at least a guess.
• Material characterization. If you make a standard-geometry coupon of the material being tested and hold it in a standard way, the above variations go away, and you can compute quantitative viscoelastic properties of the material. Alternatively, at high enough frequencies, it should be possible to characterize just the material in the region around the probe.
• Object identification. You can use compliance spectroscopy to distinguish multiple objects, even if they are made of the same material.

Topics

• Contrivances (p. 1143) (45 notes)
• Sensors (p. 1191) (8 notes)
• Scanning probe microscopy (p. 1242) (4 notes)
• Vibrators

Planning Apples to Apples, instead of Planning Poker

Kragen Javier Sitaker, 02021-09-29 (updated 02021-12-30) (6 minutes)

Apples to Apples is a popular party game; as Wikipedia previously explained it:

> Each player is dealt seven "red apple" cards; on each is printed a noun or noun phrase (Madonna, Lightning, Socks, Mahatma Gandhi, Street Gangs, London, The Universe, A Locker Room, The San Andreas Fault, Science Fiction, etc.).
>
> One player is appointed as the first "judge". She draws a "green apple" card on which is printed an adjective (Scary, Smelly, Patriotic, Rich, Aged, etc.), and places it face-up. Each of the other players places (face down) one of his red apple cards which he feels matches the green apple card. The judge shuffles the red apple cards then turns them face up (without knowing who submitted each) and chooses the one she feels is the best match for the green apple card. The player who submitted that red apple card wins the round, and takes the green apple card to signify the win.

(by Fritzlein, Vicki Rosenzweig, Zandperl, Fsufezzik, and Brian Kendig)

Cards Against Humanity is a Creative-Commons-licensed game along the same lines with blacker humor.

In Scrum, there's a "planning poker" that is sort of structured as a game, allowing the project to be planned based on estimates of costs, but it doesn't provide incentives for providing good estimates; consequently the way a selfish player would keep their job or get promoted is to spend as little effort as possible on estimates, with the predictable result that the estimates are terrible. Perhaps "Planning Apples" or "Planning Against Humanity" could provide correct incentives, so that people who put the right amount of effort into estimation would instead keep their job or get promoted. How would that work?

The first attempt is the jellybean-jar-game approach: each person's estimate of a task is anonymously recorded, and then when the task is done, whoever had the closest estimate wins a point. The estimate used for planning is some sort of average of the estimates; perhaps the median is the best average to use, or perhaps a median weighted by people's scores.

However, this creates a perverse incentive: if you get assigned a task, you have an incentive to take the exact amount of time you estimated. There's a second-order perverse incentive: you have an incentive to pad your estimate so that you're pretty sure that, if you get the task, you can sandbag your work to hit the estimate. This can be ameliorated by disqualifying the performer's guess if it happens to be closest, assigning the point to the next-closest estimate, just as the "judge" in A2A doesn't get to play a red apple card. Then two different people have to collude (against the interests of the rest of the team) to win these points. In some social contexts that is a sufficient control.

Perhaps the perverse incentive can be solved entirely by giving a

bigger prize for beating the estimate that was used for planning. In the case where that estimate was the median, this prize should be awarded when the actual time taken was lower than the highest estimate that was less than the median. So, for example, if the estimates were 2, 4, 7, 11, and 13, the median is 7, but you only win the prize if you completed it with 4 units of work or less.

That way, if you expect to be assigned the task, and you know what the other players' estimates are, by choosing your estimate adversarially you can only push the estimated time up to the next higher estimate.

In the above example, suppose the estimates on the table are 2, 4, 11, and 13; this gives you the freedom to *set* the estimate to anything between 4 and 11 by choosing a number in that range, or to set it to 4 or 11 by picking a number lower than 4 or higher than 11. If the real difficulty of the task is in the 4–11 range, you aren't going to be able to get the fast completion prize by picking a number in that range, but you can win the estimation prize by doing the best job possible at estimating the difficulty, taking the estimation prize away from the 4 guy and the 11 guy. If you pick a number below 4, such as 3, you set the estimate to 4 and forfeit the estimation prize, but you'd have to beat your own estimate of 3 to win the early completion prize, which is unlikely. But if you pick a number over 11, like 12, you set the estimate to 11, but the early completion prize threshold to 4, which, again, you probably can't beat, and again you're forfeiting the estimation prize. So, even in that extreme case, your best strategy is to estimate as honestly as you can.

If the estimates on the table are 2, 4, 7, and 11, then the same holds, but even more strongly. You can under some circumstances nudge the median up to the next partition but you don't win anything by doing so.

How do we measure completion times? Because of course if they're entirely self-reported you could claim that every task took you 0 units or 1 unit and thus win an early-completion time for every task you complete. If the other players judge your completion time, they can try to push it closer to their estimate. Perhaps we could assign each player a fixed number of points to allocate during the iteration, or assign them in proportion to their billable hours. There are additional tricky incentive-design problems there, but they may be tractable.

Topics

- Incentives (p. 1230) (5 notes)
- Employment (p. 1370) (2 notes)

Liquid dielectrics for hand-rolled self-healing capacitors

Kragen Javier Sitaker, 02021-09-30 (updated 02021-12-30)
(3 minutes)

Suppose you wanted to hand-roll an oil capacitor out of paper, oil, and aluminum foil. The capacitance is $C = \varepsilon A/d$; $\varepsilon_0 \approx 8.8541878$ pF/m, so with 100-μm-thick paper soaked in oil with a relative permittivity of 5 (though both mineral oil and vegetable oil are closer to 2, you get 0.44 μF/m². Two typical aluminum-foil rolls of 10 m × 400 mm (see file aluminum-foil.md) with an equal amount of paper will form a 3.5 μF capacitor. Mineral oil breaks down around 10–15 MV/m so this thickness is good to about 1000–1500 V.

A more polarizable liquid like glycerin (relative permittivity 41.2–42.5) could improve capacitance and energy density by an order of magnitude, if the dielectric strength doesn't suffer; in fact it's reported to be 165 kV/cm = 16.5 MV/m at 55°, slightly higher, and higher still at lower temperatures. Propylene glycol might be another appealing alternative. Like most polar liquids, both of these are hygroscopic and would therefore need to be sealed thoroughly against water penetration if water is to be avoided. However, I'm not sure that, even if anhydrous, you wouldn't suffer the same streamer-formation problem found in water-dielectric capacitors, where upon prolonged exposure to high voltage, streamers (high-conductivity paths maintained by current flow along them) form through the fluid. Electrolysis is known to be a problem with glycerin, presumably as a result of ionic contamination, but could possibly be suppressed with a thin insulating layer of glass.

To reduce water absorption, incorporating a stronger desiccant than glycerin into the capacitor was suggested by EEVblog user Zero999; coppercone2 listed desiccant alternatives, though I suspect some of the stronger ones in their list might be able to deprotonate the glycerin!

> P_2O_5 >> BaO > $Mg(ClO_4)_2$, CaO, MgO, KOH (fused), conc H_2SO_4, $CaSO_4$, Al_2O_3 > KOH (sticks), silica gel, $Mg(ClO_4)_2 \cdot 3\ H_2O$ > NaOH (fused), 95% H_2SO_4, $CaBr_2$, $CaCl_2$ (fused) > NaOH (sticks), $Ba(ClO_4)_2$, $ZnCl_2$ (sticks), $ZnBr_2$ > $CaCl_2$ (technical) > $CuSO_4$ > Na_2SO_4, K_2CO_3

Metal hydrides beat anything on the list, I think.

An important question for high-κ dielectrics is what their electrical relaxation time is; glycerol evidently is on the order of 10 ns, much longer than water but much shorter than the relaxation time of ion-movement relaxation mechanisms. This means that at higher frequencies a glycerin-dielectric capacitor would exhibit much lower capacitances. Small amounts of water in the glycerin speed this up enormously, down into the subnanosecond range with 20% water. For non-RF uses of capacitors this is adequately fast. Other researchers report much a slower relaxation time when encapsulated in silicone, in the 10 μs range, but they didn't extend their dielectric spectroscopy to the sub-microsecond range, and it looks like the relative permittivity of their composite only dropped from about 20 to about 13.

Topics

- Materials (p. 1138) (59 notes)
- Contrivances (p. 1143) (45 notes)
- Electronics (p. 1145) (39 notes)
- Physics (p. 1157) (18 notes)

Deriving binary search

Kragen Javier Sitaker, 02021-10-01 (updated 02021-12-30)
(5 minutes)

Let's look at a binary search.

The binary-search problem

We have some array or slice A such that some predicate P is true for some possibly empty prefix of A, and then false for all following elements, and we want to define a procedure bsearch(A, P) that returns the first index for which it is false — which may be an index off the end of A, if the prefix is the whole thing.

The solution

If A is empty, then the answer is simple: it's 0. In other cases we can recurse.

We choose an element from within A, which we can do because we know it's nonempty. We can choose M = #A // 2, rounding down; this is always within the bounds of A. We test P(A[M]). If it's false, we know our return value is at most M, so we can recurse on an interval that excludes it, returning bsearch(A[:M], P). This is guaranteed to make progress toward an empty array because #A // 2 < #A for #A > 0.

On the other hand, if it's true, we know the return value is at least M+1, so we can recurse, returning M+1 + bsearch(A[M+1:], P). This also reduces the interval, possibly to size 0, but never past 0.

In Python

```
def bsearch(a, p):
    if not a: return 0
    m = len(a) // 2
    if p(a[m]):
        return m+1 + bsearch(a[m+1:], p)
    else:
        return bsearch(a[:m], p)
```

Now, while this does work, it suffers from an efficiency problem in Python: the recursive calls copy the relevant interval, which makes it take linear time instead of logarithmic time. We can solve this by representing the slice as a triple (a, i, j) to mean a[i:j]:

```
def bsearch(a, p):
    return bsearch2(a, p, 0, len(a))

def bsearch2(a, p, i, j):
    if i == j: return i
    m = i + (j-i) // 2
    if p(a[m]):
        return bsearch2(a, p, m+1, j)
    else:
```

```
    return bsearch2(a, p, i, m)
```

This allows us to define the `bisect_left` and `bisect_right` functions from Python's bisect module:

```
bisect_left = lambda a, x: bsearch(a, lambda n: n < x)
bisect_right = lambda a, x: bsearch(a, lambda n: n <= x)
```

Which brings me to the puzzle I was originally trying to solve, which took me 3 minutes with those two functions:

Given a sorted array arr[] and a number x, write a function that counts the occurrences of x in arr[]. (O(Log(N)))

```
def count(arr, x):
    return bisect_right(arr, x) - bisect_left(arr, x)
```

Humblingly, though, without using the module, it took me another 30 to get the binary-search code above right, because I was writing it at the REPL instead of deriving it in the way I derived it above.

Term-rewriting

Here's a description of the above `bsearch` procedure in terms of rewrite rules with implicit equality testing:

```
bsearch(A, P) = bsearch'(A, P, 0, #A)
bsearch'(_, _, x, x) = x
bsearch'(a, p, i, j) = bsearch''(a, p, i, j, i + (j-i) // 2)
bsearch''(a, p, i, j, m) = bsearch'''(a, p, i, j, p(a[m]))
bsearch'''(a, p, i, j, True) = bsearch'(a, p, m+1, j)
bsearch'''(a, p, i, j, False) = bsearch'(a, p, i, m)
```

This is somewhat verbose, and it might contain errors (I haven't tested it) but I think it's straightforward to see how to compile the Python version to a representation like this.

Rigor

What would it mean to make rigorous the above argument about correctness?

First, we assume (have as a precondition) that the array is "sorted", which in this case means that if we step through its elements, if at some point P becomes false, it stays false for all subsequent elements. I think I can write this as

$\forall i \in [0, \#A) \colon \ \forall j \in [i, \#A) \colon \ \neg P(A_j) \lor P(A_i)$

That is, if A_i makes P false, then any A_j after it within the array must also make P false. (I'm implicitly restricting i and j to be integers.)

What we want to prove is that `bsearch` produces the right return value: some value such that all elements before it make P true and all other elements make P false; we also want it to be either a valid array index or one past the end. That is:

$\text{bsearch}(A, P) \in [0, \#A]$
$\forall i \in [0, \text{bsearch}(A, P)):\ P(A_i)$
$\forall i \in [\text{bsearch}(A, P), \#A):\ \neg P(A_i)$

This would probably involve first proving that such a value exists, using induction and the precondition above. Such a value is necessarily unique, which might or might not be necessary to prove.

We would also need to prove that bsearch terminates, which can be done with induction on $j - i$: that difference is always nonnegative, terminating the function when it reaches 0, and recursive calls always strictly diminish it. We also need to show that it doesn't exceed the array bounds of A:

$i \in [0, \#A) \wedge j \in [0, \#A] \Rightarrow i + (j-i)\ //\ 2 \in [0, \#A)$

Topics

- Programming (p. 1141) (49 notes)
- Algorithms (p. 1163) (14 notes)
- Python (p. 1166) (12 notes)
- Term rewriting (p. 1270) (3 notes)

The sol-gel transition and selective gelling for 3-D printing

Kragen Javier Sitaker, 02021-10-03 (updated 02021-12-30)
(6 minutes)

As I was washing some dried pancake batter off a bowl, I realized that there was a phase transition between the part of the batter that had congealed into masses (stuck to the surface) and the part that hadn't and could simply be washed away with water. I think this is the percolation-threshold behavior that also governs the sol-gel phase transition: once the particles of flour are close enough together on average, instead of forming small agglomerations of particles, they form a continuous network of particles. The same thing happens in paints at the critical pigment volume concentration (CPVC), and similarly with latex paints when they're diluted: if the drops in latex or pigment particles in paint fail to form a continuous network, they fall apart.

Naturally my mind ran to how this could be used for digital fabrication. This critical threshold is a point where a very small change in composition provokes a phase change between a sol and a gel, provoked, in the case of the pancake batter, by water evaporation. If you can provoke this change in the places you want to solidify, while keeping the rest of a solution as a sol, you can then just rinse away the sol to extract a green gel object, which can provide geometry for further processing. Until this point, the sol supports the gel weightlessly, since they have essentially the same density.

The new insight here is that the gelation stimulus can be made arbitrarily small to within the precision with which you can control the state of the near-critical sol.

A wide variety of gelation stimuli can be used.

The conventional approach with stereolithography 3-D printing is to use ultraviolet lasers or spatially modulated ultraviolet illumination through an LCD to produce free radicals, which locally initiate polymerization. Two-photon polymerization, where the number of free radicals produced is proportional to the square of the light intensity rather than directly to the light intensity because two photons are needed to initiate the reaction, is becoming popular. Any of these approaches can be used; the benefit of the near-critical sol is primarily in reducing the needed amount of light.

The spatially controlled addition of small amounts of material is another possibility, either a catalyst or a limiting reagent; this can be done either at the surface of a sol bath, similar to powder-bed printing, or throughout its volume with one or more movable nozzles, similar to FDM printing, though this poses the risk of the gelated material sticking to the nozzle instead of the workpiece. Electrolysis is one appealing way to locally deposit some ions, and electrolysis can also be used to initiate other reactions.

Worth special mention here is the possibility of using inkjet nozzles or gas jets scanned over the surface of a bath in order to precisely

deposit the reagent.

Related to electrolysis, but different, is the possibility of plasma activation, where corona discharge around sharp points is used to stimulate some points on the surface of the sol but not others.

Another possibility is locally altering the temperature, either down (with a gas jet) or up (with a gas jet, flame, plasma, or laser or other light). Almost any resin system that can will polymerize on its own in enough time, such as commercial casting polyesters and epoxies, can also be convinced to polymerize much more rapidly with some heat, while a wide variety of bistable soluble gelling materials such as agar-agar will gelate upon cooling to a critical gelation temperature, but remain gelled up to a higher melting temperature.

If the sol has an extremely low vapor pressure, or an e-beam window can be moved very close to it, local gelation with an electron beam is also a possibility. This potentially provides finer spatial resolution than visible light.

In an upside-down printing vat like those used for LCD UV stereolithography resin polymerization printers, another possibility is to electrostimulate the bottom of the resin capacitively, for example by pushbroom-scanning a line of electrodes across the bottom of the vat, outside the protective membrane, while modulating an RF signal onto the electrodes. This would at least locally heat it up, which is enough to have the desired effect, but maybe even resistive heating through the membrane would be enough.

By substituting a cathode-ray tube for the vat-bottom membrane, it's possible to stimulate the sol with light, X-rays, heat, or electron beams that pass through the glass. Ion beams can be substituted for electron beams, with the usual tradeoffs. Ideally the gel would polymerize a slight distance away from the glass to avoid mechanical forces on the glass; this would be least infeasible with the electron-beam approach.

The sol can be formulated with a wide variety of functional fillers which, among other things, reduce the amount of material that needs to be gelled to form a continuous network.

The above gelation stimuli can also be used without the sol, of course, as they are in conventional stereolithography.

Topics

- Materials (p. 1138) (59 notes)
- Digital fabrication (p. 1149) (31 notes)
- Electrolysis (p. 1158) (18 notes)
- 3-D printing (p. 1160) (17 notes)
- Frrickin' lasers! (p. 1168) (12 notes)
- Plasma (p. 1339) (2 notes)

Some notes on perusing the Udanax Green codebase

Kragen Javier Sitaker, 02021-10-05 (updated 02021-10-08) (12 minutes)

Some notes on perusing the Udanax Green codebase, with particular attention to how it thinks about version tracking.

There's a slightly updated version at https://github.com/dotmpe/udanax-mpe.

Conceptual guides I've found

A conceptual overview is in `olddemo/demo_docs/Bizplan4`.

The place to start for nuts and bolts is quite likely green/man, which contains man pages that GNU man can format with, e.g., `man green/man/fex.L`, which documents the frontend UI. I suspect the "Xanadu FeBe Protocol 88.1x" document might be more informative if I could find it. `man green/man/xumain.L` offers an illuminating glimpse into how Xanadu thought of documents at the time.

Each document is evidently identified by a "docid", a dot-separated sequence of numbers called a "tumbler"; the numbers constituting the tumbler are sometimes called "digits" or "tdigits". (Sometimes a tumbler is also called an "isa" in the source.) Tumblers, like tilde-bearing WWW URLs, can contain in sequence a network node identifier, a user account (/~kragen/, represented as a number in a tumbler), a document ID within that account, and a region identifier within that document; so they can identify finer-grained or coarser-grained things than individual documents. A "span" is specified in terms of character counts. I think? Bizplan4 says:

> by combining a document's unique identifier with a character position within that document, we can uniquely address any character stored in the entire system.
>
> By designating a particular (starting) character and a length we can address any contiguous string of characters in the system. Such a string is called a "span." By clever use of tumblers instead of plain integers to represent character positions and span lengths, it is possible to have spans which contain whole documents and cross document boundaries.

There doesn't seem to be any thought given (in `green/olddemo/demodocs/ReplacingD`, anyway) to the problem of updating documents while retaining links. They do conceive of multiple versions of a document, and in `green/olddemo/demo_docs/Suespaper` it says, "Xanadu hypertext can maintain multiple versions of any given document, efficiently storing the common portions in common, ... All such editing wil [sic] be logged by Xanadu's historical trace function. Xanadu can provide historical traceback information in dated chronological order about any and all changes to a given document."; but in the status page on the WWW "Historical Trace" is in the "Needs Implementation" column.

The concern for the space required for multiple versions seems very quaint now; it's easy to forget that even when Subversion was introduced in October 02000 the fact that it kept a separate "pristine" copy of whatever source code you were editing was considered a

major concern, and in 02001 crazy old Tom Lord's arch's policy of keeping the entire version history on your development workstation was, for many, considered a showstopper. (Today almost everybody uses Git, except for people working with large binary assets, who often use Subversion.)

The source base itself

It's surprising they chose C as an implementation language so early. It's archaic K&R C formatted with 2-space tab stops (in `less`, use `-x2`), but it's C. Before 01984 it doesn't seem like it would have been an obvious choice; it seems like at some point they decided to restart from scratch on a 68000-based Sun. `green/olddemo/demo_docs/Datamation`, which seems to be from 01982, says "running under Unix on a Motorola 68000-based Onyx desktop computer," but I think Onyx's Unix machines were Z8000/Z8002-based, running the first microcomputer Unix in 01980, but even in 01981 people had a hard time finding the company. Evidently the team was foresightful enough to recognize the importance of Unix. The choice of C is a pleasant surprise; it makes the codebase a lot more readable than, say, EUMEL.

In the context of the space concerns mentioned above, it's sort of surprising that the source base *per se* is 39,000 lines of C weighing 1.07 megabytes (16000 unique lines weighing 650 KB), in significant part because they named their functions things like `klugefindisatoinsertnonmolecule`.

In the backend types are mostly named with the prefix "type", as in "typecuc" or "typetask"; in the frontend that is more typically a suffix, as in "spectype" and "cutseqtype".

The source base seems to be fairly consistent in putting the names of functions being defined (and *only* functions being defined) at the left margin without indentation. I think this is an accommodation for archaic versions of `ctags`, but it's handy as a way to navigate the source base without `ctags`.

I'm inferring that maybe documents, at least in the frontend (`fe_source/vm.c`) are referred to by "specs" (type `spectype`) and consist of "charspans" (`charspantype`). The spec contains a `specspanptr` (which I guess is a charspan?), which contains a `sizeofspan`, and a `docid`, which is a "tumbler" and can be sent to the backend. Destroying a spec is done with `specfree(&specptr)`.

They've implemented their own virtual memory system made of objects of type `vmthingtype`. There's a subtyping graph:

```
spectype* -> vmthingtype*
charspan* -> vmthingtype*
```

The frontend evidently contemplates editing documents; there's a `sendrearrange` function in `fe_source/sendtop.c` which sends a "cutseq", I guess like an EDL for text?, to the backend. There's also `sendinsert`, which makes it seem like the granularity of edits is very small.

A "link" is from a spec, to a spec, three a spec ("threesets are used to describe the intended meaning of the link, such as if it is a jump link or a footnote link", the other possibilities evidently being

“quote” and “marginal note”). But when the frontend does sendcreatelink to the backend, it also includes an additional docid, apart from those in the three specs; this is explained in olddemo/demo_docs/Bizplan4:

> Links themselves are said to reside inside documents, located in a separate logical address space from the text. A link may reside in one document and link from a second document to yet a third. The location of residence of a link is entirely independent of the contents of the link's end-sets. The fact that links are among the potential contents of documents means that links themselves may be linked from or to, just as characters may. Thus very general sorts of indirect structures may be assembled.

For the frontend they seem to have used curses, including its windowing system (well, newwin, delwin, and wrefresh, anyway), and also some kind of Sun GUI (it’s trying to #include <suntool/tool_hs.h>, call win_setcursor, etc.) This seems to be the “sunwindow” system sometimes called “SunWindows” and the “Sun Graphical User Interface,” and later called SunView, mentioned in http://homepages.rpi.edu/home/56/frankwr/afs/rpi.edu/campus/sun/lang/2.01/SC2.0.1/include/CC_413/sunwindow/win_struct.h, (which seems to be part of a huge repository of old Sun stuff that was shipped to customers, up to Solaris 2.5) and documented in http://bitsavers.trailing-edge.com/pdf/sun/sunos/4.1/800-1784-11A_SunView_System_Programmers_Guide_199003.pdf; this window system had a file descriptor per window.

The backend refers to something called a “granfilade”, which I think is one of the three possible types of enfilade: GRAN, SPAN, and POOM.

Creating new versions

The protocol documented in green/olddemo/demo_docs/Feidoc includes a CREATENEWVERSION function:

> A version is a document, the only distinction being that a version is descended from a previously existing document. A new version inherits all of its immediate ancestor's information, both textual and topological.

Its tumbler is to be nested under the document it was created from:

> For example, if there is only one version of document 23.0.17.0.256, this request will create a document with the id 23.0.17.0.256.1.

However, this isn’t always possible, because sometimes you create a new version of a document that somebody else owns, and you don’t get to inject it into their namespace like that. So, is there some way to trace the history back to the original?

The CREATENEWVERSION request type is defined in green/fe_source/requests.h and green/be_source/requests.h as #define CREATENEWVERSION 13; the backend vectors it to a createnewversion function in green/be_source/fns.c, which calls docreatenewversion in green/be_source/do1.c, which eventually calls docopyinternal with a new “isa” pointer (allocated, I think, by doopen) and vspans and vspecs it got from the old document. As mentioned above, “isa” is another name for “tumbler”, or maybe a certain kind of tumbler.

There’s a conditional at the top of docreatenewversion which checks to see if the wheretoputit argument identifies a tumbler that belongs to the user, invoking makehint differently; makehint invokes movetumbler,

which is just a struct assignment macro which copies its left argument over its right argument.

Within `docreatenewversion`, in the case where the document does not belong to the user, there is no evident dataflow from `isaptr` (the tumbler of the document being copied) to `createorglingranf` (in `green/be_source/granf1.c`, delegating to `createorglgr` in `green/be_source/granf2.c`), which I guess allocates the new tumbler with `findisatoinsertgr` (just below) and inserts it somewhere with `insertseq`.

I'm not sure what an "orgl" or a "sporgl" is but evidently it's some kind of thing that gets stored on disk, an orgl being some sort of cuc, and maybe a granfilade consists of null, texts, and orgls. In one place ORGLRANGE and SPANRANGE are explained as "wid and dsp indexes for sp", which is pretty significant; widative and dspative are the two major dimensions of enfilade theory. A diagram, however, seems to suggest that this is not respective but orthogonal: both wid and dsp have both ORGLRANGE and SPANRANGE.

After `createorglingranf`, `docreatenewversion` builds a vspec and apparently copies the original `isaptr` into it, and then invokes `doopen` with `newisaptr` and without the vspec or the original `isaptr`, and then uses `docopyinternal` to copy the old document to the new document, using the vspec. Also, the thing that it's copying is the stream of a vspan obtained from `doretrievedocvspanfoo`, which delegates to `retrievedocumentpartofvspanpm`, which sets the stream and width of the vspan from two tumblers found in the orgl: `cdsp->dsas[V]` and `cwid->dsas[V]`. I'm not quite sure what these are for; do they offer a way to navigate back to the original version?

The vspec passed to `docopyinternal` (containing the original `isaptr`) is a "specset" to `docopyinternal`; it gets converted to a "spanset" named `ispanset`. This is passed to `insertpm` and `insertspanf`. `specset2ispanset` loops over the linked list of `spanset` objects (they have a `->next`) and specifically looks at the `docisa` copied into the vspec by `docreatenewversion`, passing it as the third argument to `findorgl`, so it can use it to fetch the orgl with `fetchorglgr`. But that's all it does with the `docisa` that `docreatenewversion` copied from the `isaptr`.

However, when `docreatenewversion` is being invoked from `createnewversion`, it *also* gets the original document tumbler as its `wheretoputit` argument! So actually there is data flow from the original tumbler into *both* of the `makehint` cases, so the `hintisa` of the hint will have the original document tumbler in it.

Topics

- Programming (p. 1141) (49 notes)
- History (p. 1153) (24 notes)
- C (p. 1194) (8 notes)
- Reading (p. 1244) (4 notes)
- Hypertext (p. 1291) (3 notes)
- Namespaces (p. 1351) (2 notes)

Fung's "I can't believe it can sort" algorithm and others

Kragen Javier Sitaker, 02021-10-05 (updated 02021-12-30)
(5 minutes)

Fung published "the simplest (and most surprising) sorting algorithm ever" this year.

The algorithm from the paper is indeed an astonishingly simple sorting algorithm, and it is indeed surprising that it works, particularly since at first glance it would appear to sort the array *backwards* (see the paper):

```
void
cantbelievesort(int *p, size_t n)
{
  int tmp;
  for (size_t i = 0; i < n; i++) {
    for (size_t j = 0; j < n; j++) {
      if (p[i] < p[j]) tmp = p[i], p[i] = p[j], p[j] = tmp;
    }
  }
}
```

That's 10 lines of code according to David A. Wheeler's 'SLOCCount,' though arguably 9 would be fairer. Its cyclomatic complexity is 3, with a deepest statement nesting level of 4, and it has 5 variables: three locals plus two arguments.

```
0000000000001369 <cantbelievesort>:
    1369:    f3 0f 1e fa             endbr64
    136d:    48 89 f8                mov    %rdi,%rax
    1370:    4c 8d 0c b7             lea    (%rdi,%rsi,4),%r9
    1374:    49 39 c1                cmp    %rax,%r9
    1377:    74 23                   je     139c <cantbelievesort+0x33>
    1379:    31 d2                   xor    %edx,%edx
    137b:    48 39 f2                cmp    %rsi,%rdx
    137e:    74 16                   je     1396 <cantbelievesort+0x2d>
    1380:    8b 08                   mov    (%rax),%ecx
    1382:    44 8b 04 97             mov    (%rdi,%rdx,4),%r8d
    1386:    44 39 c1                cmp    %r8d,%ecx
    1389:    7d 06                   jge    1391 <cantbelievesort+0x28>
    138b:    44 89 00                mov    %r8d,(%rax)
    138e:    89 0c 97                mov    %ecx,(%rdi,%rdx,4)
    1391:    48 ff c2                inc    %rdx
    1394:    eb e5                   jmp    137b <cantbelievesort+0x12>
    1396:    48 83 c0 04             add    $0x4,%rax
    139a:    eb d8                   jmp    1374 <cantbelievesort+0xb>
    139c:    c3                      retq
```

With gcc -Os 9.3.0, that's 52 bytes, 19 instructions. gcc -Os has regressed; in 4.7.2, that used to be 44 bytes, 17 instructions:

```
4007cc:       31 c0                   xor    %eax,%eax
4007ce:       eb 22                   jmp    4007f2 <cantbelievesort+0x26>
4007d0:       8b 0c 87                mov    (%rdi,%rax,4),%ecx
4007d3:       44 8b 04 97             mov    (%rdi,%rdx,4),%r8d
4007d7:       44 39 c1                cmp    %r8d,%ecx
4007da:       7d 07                   jge    4007e3 <cantbelievesort+0x17>
4007dc:       44 89 04 87             mov    %r8d,(%rdi,%rax,4)
4007e0:       89 0c 97                mov    %ecx,(%rdi,%rdx,4)
4007e3:       48 ff c2                inc    %rdx
4007e6:       eb 02                   jmp    4007ea <cantbelievesort+0x1e>
4007e8:       31 d2                   xor    %edx,%edx
4007ea:       48 39 f2                cmp    %rsi,%rdx
4007ed:       75 e1                   jne    4007d0 <cantbelievesort+0x4>
4007ef:       48 ff c0                inc    %rax
4007f2:       48 39 f0                cmp    %rsi,%rax
4007f5:       75 f1                   jne    4007e8 <cantbelievesort+0x1c>
4007f7:       c3                      retq
```

However, arguably, *this* sort routine is even *simpler*. It may or may not be surprising that it works:

```
void
dumbsort(int *p, size_t n)
{
  int tmp;
  for (size_t i = 1; i < n; i++) {
    if (p[i] < p[i-1]) tmp = p[i], p[i] = p[i-1], p[i-1] = tmp, i = 0;
  }
}
```

I don't think I invented it, but I can't remember who did.

By the same metric, that's only 8 lines of code (also only 8 by my "fairer" metric), its cyclomatic complexity is only 2, its deepest statement nesting level is only 3, and it has only 4 variables (the same two arguments, but two locals instead of three). It compiles to only 15 amd64 instructions, occupying only 43 bytes:

```
  4007f8:       b8 01 00 00 00          mov    $0x1,%eax
  4007fd:       eb 1e                   jmp    40081d <dumbsort+0x25>
  4007ff:       4c 8d 04 87             lea    (%rdi,%rax,4),%r8
  400803:       48 8d 54 87 fc          lea    -0x4(%rdi,%rax,4),%rdx
  400808:       41 8b 08                mov    (%r8),%ecx
  40080b:       44 8b 0a                mov    (%rdx),%r9d
  40080e:       44 39 c9                cmp    %r9d,%ecx
  400811:       7d 07                   jge    40081a <dumbsort+0x22>
  400813:       45 89 08                mov    %r9d,(%r8)
  400816:       31 c0                   xor    %eax,%eax
  400818:       89 0a                   mov    %ecx,(%rdx)
  40081a:       48 ff c0                inc    %rax
  40081d:       48 39 f0                cmp    %rsi,%rax
  400820:       72 dd                   jb     4007ff <dumbsort+0x7>
  400822:       c3                      retq
```

Or, with more recent gcc, 55 bytes, 18 instructions:

```
000000000000139d <dumbsort>:
    139d:   f3 0f 1e fa             endbr64
    13a1:   b8 01 00 00 00          mov    $0x1,%eax
    13a6:   48 39 f0                cmp    %rsi,%rax
    13a9:   73 28                   jae    13d3 <dumbsort+0x36>
    13ab:   48 8d 14 85 00 00 00    lea    0x0(,%rax,4),%rdx
    13b2:   00
    13b3:   4c 8d 04 17             lea    (%rdi,%rdx,1),%r8
    13b7:   48 8d 54 17 fc          lea    -0x4(%rdi,%rdx,1),%rdx
    13bc:   41 8b 08                mov    (%r8),%ecx
    13bf:   44 8b 0a                mov    (%rdx),%r9d
    13c2:   44 39 c9                cmp    %r9d,%ecx
    13c5:   7d 07                   jge    13ce <dumbsort+0x31>
    13c7:   45 89 08                mov    %r9d,(%r8)
    13ca:   31 c0                   xor    %eax,%eax
    13cc:   89 0a                   mov    %ecx,(%rdx)
    13ce:   48 ff c0                inc    %rax
    13d1:   eb d3                   jmp    13a6 <dumbsort+0x9>
    13d3:   c3                      retq
```

Dylan16807 points out that if you tail-recurse to restart the function instead of resetting the index, it gets even simpler.

Topics

- Programming (p. 1141) (49 notes)
- Performance (p. 1155) (22 notes)
- Algorithms (p. 1163) (14 notes)
- Assembly-language programming (p. 1175) (11 notes)
- Small is beautiful (p. 1190) (8 notes)
- C (p. 1194) (8 notes)
- Sorting (p. 1272) (3 notes)

Spanish phonology

Kragen Javier Sitaker, 02021-10-05 (updated 02021-12-31)
(15 minutes)

I hacked together a Python script to analyze the syllabic structure and pronunciation of Spanish text at the phoneme level, which it seems to get about 97% right, about one error every 30 words. It thinks each syllable consists of a (possibly empty) onset, a (nearly always empty) liquid, a vowel nucleus (which may be a diphthong), and a (usually empty) coda.

Here's some example output, containing asterisks where it failed:

divide.^[173] /di-bi-de/ Los /los/ detalles /de-ta-ʃes/ del /del/ ciclo /si-klo/ celular /se-lu-laɾ/ solo /so-lo/ han /an/ sido /si-do/ investigados /in-bes-ti-ga-dos/ en /en/ el /el/ género /xe-ne-ɾo/ Sulfolobus, /sul-fo-lo-bus/ siendo /sien-do/ similares /si-mi-la-ɾes/ a /a/ los /los/ de /de/ bacterias /bak-te-ɾias/ y /i/ eucariontes: /e-u-ka-ɾion-tes/ los /los/ cromosomas /kɾo-mo-so-mas/ se /se/ replican /re-pli-kan/ desde /des-de/ múltiples /mul-ti-ples/ puntos /pun-tos/ de /de/ partida /paɾ-ti-da/ (origen /o-ɾi-xen/ de /de/ replicación) /re-pli-ka-sion/ usando /u-san-do/ ADN /ad-*/ polimerasas /po-li-me-ɾa-sas/ que /ke/ son /son/ similares /si-mi-la-ɾes/ a /a/ las /las/ enzimas /en-si-mas/ equivalentes /e-ki-ba-len-tes/ eucarióticas.^[174] /e-u-ka-ɾio-ti-kas/ Sin /sin/ embargo, /em-baɾ-go/ las /las/ proteínas /pɾo-tei-nas/ que /ke/ dirigen /di-ɾi-xen/ la /la/ división /di-bi-sion/ celular, /se-lu-laɾ/ como /ko-mo/ la /la/ proteína /pɾo-tei-na/ FtsZ /*-*-*-*/

It's using mostly IPA, except for č ("ch") and ñ. You can see it screwing up on a couple of words.

Based on this sort of analysis, it produces the following statistics, which I think are probably in the right ballpark, although it doesn't recognize "ai" or "ui" as valid diphthongs, and the significant figures are mostly noise:

```
  onsets:  : 19.17%  t: 11.42%  d:  9.20%  s:  8.86%  k:  8.68%  l:  7.93%
           n:  6.30%  m:  6.02%  p:  5.56%  b:  4.21%  ɾ:  4.18%  f:  2.35%
           g:  2.03%  x:  1.64%  r:  1.14%  č:  0.69%  ʃ:  0.48%  ñ:  0.14%
 liquids:  : 94.46%  ɾ:  4.26%  l:  1.28%
  nuclei: e: 28.16%  a: 26.47%  o: 19.84%  i: 12.79%  u:  5.91% io:  2.16%
         ia:  1.86% ie:  1.29% ue:  0.80% ua:  0.49% ei:  0.19% uo:  0.05%
   codas:  : 62.14%  s: 15.65%  n: 10.11%  ɾ:  5.72%  l:  2.40%  m:  1.51%
           k:  1.32% ks:  0.47%  d:  0.20%  p:  0.20% ns:  0.13%  g:  0.08%
           b:  0.05%
  phones: e: 13.14%  a: 12.44%  s: 10.84%  o:  9.52%  i:  7.90%  n:  7.14%
           ɾ:  6.11%  l:  5.01%  t:  4.93%  k:  4.52%  d:  4.06%  m:  3.25%
           u:  3.13%  p:  2.49%  b:  1.84%  f:  1.01%  g:  0.91%  x:  0.71%
           r:  0.49%  č:  0.30%  ʃ:  0.21%  ñ:  0.06%
```

This doesn't capture all the structure of Spanish syllables; for example, only some onsets can be followed by liquids. But it is at least suggestive that the syllable structure is potentially exploitable for both encoding and human input methods (as, undoubtedly, stenographers already do.) The most common nucleus is /e/, followed closely by /a/, which between them capture the majority of syllables. The majority of syllables start with a vowel (19% empty onset) or either /t/, /d/, /s/, or /k/ (usually spelled "c" or "qu"), nearly all

syllables lack a liquid following the onset, and the majority of codas are empty. So a syllable containing only "e" is 5.01 bits, one containing only "a" is 5.12 bits, the syllable "te" is 5.75 bits, and "tes" (as in *eucariontes* or *equivalentes*) is 7.74 bits. By contrast, a memoryless code represents "e" as 2.93 bits, more efficiently, but "tes" as 10.5 bits, because "t" almost never occurs at the end of a syllable, and "s" is much more common in the coda.

(Actually, my program doesn't allow it to, so it can't parse "habitat", but it only very rarely encounters such a word.)

"bes" is 9.16 bits analyzed as a syllable, or 11.90 bits as three separate letters. "ti" is 6.87 bits as a syllable or 8.00 bits as two separate letters. "ga" is 8.31 bits as a syllable or 9.79 bits as two separate letters. So I guess often you could save about 20% of the effort that way.

However, the sort of long tail of diphthongs and multi-consonant codas is crushing my hopes for a human-written "relative positional" notation in which the same symbol is used for consonants in some places and vowels in others, and only the order of the symbols distinguishes between those meanings. I feel like it would just be too hard to learn given that all of the following occur in existing, if archaic, words: "crue" ("crueza"), "cruen" ("cruenta"), "cue" ("cuero"), "cuen" ("cuenta"), "que" ("que"), "quen" ("aquende"), "cu" ("culo"), "cun" ("secundaria"), and "cuns" ("circunscribir", "circunstancia"). (I'm not totally sure bout "crueza" and "cruenta", which I think might be three syllables.)

At the end of my program's output it says:

```
bits per phone 3.879586262782223
bits per letter 3.586046123445401
```

This is 3.9 bits per phoneme (such as /ɾ/, /k/, or /e/) and 3.6 bits per letter (such as "h" or "c"), based on independently entropy-coding each letter. This is about 25% smaller than just flattening the usual Spanish alphabet, abcdefghijlmnñopqrstuvxyz, (to which we might reluctantly add k and w) because $\lg 27 \approx 4.75$ bits per letter, and the diaeresis for things like *cigüeño* and *bilingüe*; standardly we would also add áéíóúý, but I'm ignoring that here.

A pure Hamming phonetic code might assign 3-bit codes to /e/ and /a/, 4-bit codes to [soin], 5-bit codes to [ɾltkdmu], 6-bit codes to [pb], 7-bit codes to [fgx], an 8-bit code to [r] ("rr"), 9-bit codes to č [tʃ] "ch" and ʃ "ll", and an 11-bit code to ñ. This would still leave 448 11-bit codes unassigned and would use 4.32 bits per phone or 3.99 bits per letter:

```
(- 2048 (+ 1 (* 4 2) (* 8 1) (* 16 3) (* 32 2) (* 64 7) (* 128 4)
             (* 256 2)))  ; 448

(+ (* 3 .1314) (* 3 .1244) (* 4 .1084) (* 4 .0952) (* 4 .0790) (* 4 .0714)
   (* 5 .0611) (* 6 .0501) (* 5 .0493) (* 5 .0452) (* 5 .0406) (* 5 .0325)
   (* 5 .0313) (* 6 .0249) (* 6 .0184) (* 7 .0101) (* 7 .0091) (* 7 .0071)
   (* 8 .0049) (* 9 .0030) (* 9 .0021) (* 11 .0006))  ; 4.3196

(+ (* 1 .1314) (* 1 .1244) (* 1 .1084) (* 1 .0952) (* 1 .0790) (* 1 .0714)
   (* 1 .0611) (* 1 .0501) (* 1 .0493) (* 1 .0452) (* 1 .0406) (* 1 .0325)
```

```
(* 1 .0313) (* 1 .0249) (* 1 .0184) (* 1 .0101) (* 1 .0091) (* 1 .0071)
(* 1 .0049) (* 1 .0030) (* 1 .0021) (* 1 .0006))   ; 1.0001
```

However, whether for human interface stuff or for data compression, there's no reason to restrict yourself to radix 2 nowadays; IBM's arithmetic-encoding patents are long expired, and I'm typing this on a 100-key keyboard, though the home row and the keys above and below are only about radix 33. Still, it's a good ballpark that [ea] should be the easiest to write, followed by [soinɾltkdmu], and then [pb], requiring about twice as much effort as [ea], then [fgxr], then "ch" (č) requiring about three times as much effort as [ea], then finally [ʃñ].

You might omit č entirely, just spelling it out as /tʃ/, although that would be pretty rough on speakers of other dialects of Spanish, even worse than lumping [z] in with [s].

If we wanted to divide this up in some sort of rational and easy-to-learn way, we might try to express [pbfgxrčʃñ] as some kind of modified version of [soinɾltkdmu], which could be divided up into vowels [oiu], sonorants [lmnd], plosives [ɾtk], and the sibilant [s]. (Spanish /d/ has a fricative allophone [ð] and a plosive allophone [d].) There's inevitably going to be significant tension between fluent writing and ease of learning.

Here's the Python program:

```
#!/usr/bin/python3
"""Analyze phonetics and phonotactics of Spanish text.

The syllabification is somewhere around 97% accurate on the text I
tried it on, but it does have problems, for example with "mayoría"
(should be ma-yo-ri-a; same problem with "podrían", "amoníaco", and
"biotecnología") and "inusuales" (should be in-u-sua-les).  Also it
thinks "hay" and "muy" are two syllables each.  It fails on
"habitats", "atmosférico", "plancton", and "sulfhidrico".  And it
segments "aire" as "a-i-re", which I think is wrong, and similarly
"causado" as "ca-u-sa-do", "autodenominado" as "a-u-to-...",
"contribuir" as "con-tri-bu-ir", and "cuidadores" as "cu-i-da-do-res".

No attempt is made to handle numbers, initialisms, or loanwords not
spelled using Spanish orthography.

It's using a slightly deformed version of IPA.  For [tʃ] I'm using the
Esperanto "č", in the interest of having it be one letter for the
purpose of frequency tabulation, and for [ɲ] I'm using the Spanish
"ñ", in the interest of readability.  No attempt is made to select the
appropriate allophone for context (e.g., [ŋ] for /n/, [β] for /b/, [x]
for /s/).  "*" is output on error.  The pronunciation is mostly
Argentine.  Here's some sample output, containing one error:

> divide.^[173]   /di-bi-de/ Los /los/ detalles /de-ta-ʃes/ del /del/
  ciclo /si-klo/ celular /se-lu-laɾ/ solo /so-lo/ han /an/ sido
  /si-do/ investigados /in-bes-ti-ga-dos/ en /en/ el /el/ género
  /xe-ne-ɾo/ Sulfolobus, /sul-fo-lo-bus/ siendo /sien-do/ similares
  /si-mi-la-ɾes/ a /a/ los /los/ de /de/ bacterias /bak-te-ɾias/ y /i/
  eucariontes: /e-u-ka-ɾion-tes/ los /los/ cromosomas /kɾo-mo-so-mas/
```

se /se/ replican /re-pli-kan/ desde /des-de/ múltiples /mul-ti-ples/
puntos /pun-tos/ de /de/ partida /paɾ-ti-da/ (origen /o-ɾi-xen/ de
/de/ replicación) /re-pli-ka-sion/ usando /u-san-do/ ADN /ad-*/
polimerasas /po-li-me-ɾa-sas/ que /ke/ son /son/ similares
/si-mi-la-ɾes/ a /a/ las /las/ enzimas /en-si-mas/ equivalentes
/e-ki-ba-len-tes/ eucarióticas.^[174]   /e-u-ka-ɾio-ti-kas/ Sin /sin/
embargo, /em-baɾ-go/ las /las/ proteínas /pɾo-tei-nas/ que /ke/
dirigen /di-ɾi-xen/ la /la/ división /di-bi-sion/ celular,
/se-lu-laɾ/ como /ko-mo/ la /la/ proteína /pɾo-tei-na/ FtsZ
/*-*-*-*/

Here's some more, containing three errors in 110 words:

> El /el/ hecho /e-čo/ tuvo /tu-bo/ tintes /tin-tes/ mafiosos
> /ma-fio-sos/ y /i/ elementos /e-le-men-tos/ que /ke/ ya /ʃa/ se /se/
> habían /a-bian/ visto. /bis-to/ Lo /lo/ protagonizó
> /pɾo-ta-go-ni-so/ un /un/ grupo /gɾu-po/ de /de/ encapuchados
> /en-ka-pu-ča-dos/ autodenominados /a-u-to-de-no-mi-na-dos/
> "mapuches" /ma-pu-čes/ en /en/ Río /rio/ Negro, /ne-gɾo/ el /el/
> domingo /do-min-go/ a /a/ la /la/ noche. /no-če/ Primero /pɾi-me-ɾo/
> ataron /a-ta-ɾon/ a /a/ los /los/ cuidadores /ku-i-da-do-ɾes/ de
> /de/ un /un/ predio /pɾe-dio/ de /de/ Vialidad /bia-li-dad/
> provincial, /pɾo-bin-sial/ después /des-pues/ dejaron /de-xa-ɾon/
> notas /no-tas/ intimidatorias /in-ti-mi-da-to-ɾias/ con /kon/
> amenazas /a-me-na-sas/ y, /i/ antes /an-tes/ de /de/ escapar,
> /es-ka-paɾ/ incendiaron /in-sen-dia-ɾon/ un /un/
> depósito. /de-po-si-to/

> El /el/ Gobierno /go-bieɾ-no/ rionegrino /rio-ne-gɾi-no/ calificó
> /ka-li-fi-ko/ como /ko-mo/ "un /un/ acto /ak-to/ terrorista"
> /te-ro-ɾis-ta/ el /el/ ataque /a-ta-ke/ y /i/ prepara /pɾe-pa-ɾa/
> una /u-na/ presentación /pɾe-sen-ta-sion/ ante /an-te/ la /la/
> Justicia /xus-ti-sia/ federal. /fe-de-ɾal/ En /en/ paralelo
> /pa-ɾa-le-lo/ pidió /pi-dio/ al /al/ Gobierno /go-bieɾ-no/ nacional
> /na-sio-nal/ el /el/ envío /en-bio/ de /de/ fuerzas /fueɾ-sas/
> federales /fe-de-ɾa-les/ para /pa-ɾa/ controlar /kon-tɾo-laɾ/ la
> /la/ situación /si-tua-sion/ que /ke/ viene /bie-ne/ escalando
> /es-ka-lan-do/ desde /des-de/ hace /a-se/ meses. /me-ses/ Se /se/
> suma /su-ma/ el /el/ incendio /in-sen-dio/ provocado /pɾo-bo-ka-do/
> en /en/ la /la/ Oficina /o-fi-si-na/ de /de/ Turismo /tu-ɾis-mo/ de
> /de/ El /el/ Bolsón, /bol-son/ ocurrido /o-ku-ri-do/ el /el/ sábado
> /sa-ba-do/ por /poɾ/ la /la/ noche. /no-če/

<https://www.clarin.com/sociedad/mapuches-incendiaron-campamento-vialidad-rio-negro-gobernadora-pidio-apoyo-gobierno-nacional_0_ByRHc8tKZ.html>

Despite the problems mentioned earlier, I think it's accurate enough
for these statistics to be mostly right:

```
  onsets:  : 19.17%  t: 11.42%  d:  9.20%  s:  8.86%  k:  8.68%  l:  7.93%
          n:  6.30%  m:  6.02%  p:  5.56%  b:  4.21%  ɾ:  4.18%  f:  2.35%
          g:  2.03%  x:  1.64%  r:  1.14%  č:  0.69%  ʃ:  0.48%  ñ:  0.14%
 liquids:  : 94.46%  ɾ:  4.26%  l:  1.28%
  nuclei: e: 28.16%  a: 26.47%  o: 19.84%  i: 12.79%  u:  5.91% io:  2.16%
```

```
        ia:  1.86% ie:  1.29% ue:  0.80% ua:  0.49% ei:  0.19% uo:  0.05%
   codas:   : 62.14%  s: 15.65%  n: 10.11%  ɾ:  5.72%  l:  2.40%  m:  1.51%
           k:  1.32% ks:  0.47%  d:  0.20%  p:  0.20% ns:  0.13%  g:  0.08%
           b:  0.05%
  phones: e: 13.14%  a: 12.44%  s: 10.84%  o:  9.52%  i:  7.90%  n:  7.14%
           ɾ:  6.11%  l:  5.01%  t:  4.93%  k:  4.52%  d:  4.06%  m:  3.25%
           u:  3.13%  p:  2.49%  b:  1.84%  f:  1.01%  g:  0.91%  x:  0.71%
           r:  0.49%  č:  0.30%  ʃ:  0.21%  ñ:  0.06%
bits per phone 3.879586262782223
bits per letter 3.586046123445401

"""
from __future__ import print_function, division
import collections, math, re, sys

syllable = re.compile(r'''
    (?P<syllable>
        (?P<onset>b|c|ch|d|f|[gq]u(?![lr])|g|h|j|k|ll|l
                 |m|n(?![lr])|ñ|ph|p|rr|s|th|t|v|y|z|)
        (?P<liquid>l|r|)
        (?P<nucleus>a|ei|e|w|ia|io|ie|i|o|ua|ue|u|y)
        # The negative lookahead assertions are a hack to keep
        # from chomping up codas that really belong to the
        # onset of a following syllable: na ve gac ion,
        # en cic lo ped ia, etc.
        (?P<coda>(?:b|c(?!h)|d|g|l|m|ns|r|p|s|z)(?![aeiouylr])|x|n(?![aeiouy])|)
    )
  | (?P<err> .)
''', re.VERBOSE)

accents = {'á': 'a', 'é': 'e', 'í': 'i', 'ó': 'o', 'ú': 'u',
           'ü': 'u'}

def syllabize(word):
    w = ''.join(d for d in (accents.get(c, c)
                            for c in word.lower())
                if d == 'ñ' or 'a' <= d <= 'z')
    return list(syllable.finditer(w))

def pronounce(syllable, first):
    if not syllable.group('syllable'):
        yield '*'
        return

    onset = syllable.group('onset')
    liquid = syllable.group('liquid')
    nucleus = syllable.group('nucleus')
    coda = syllable.group('coda')

    mapped = {'ch': 'č', 'c': 'k', 'qu': 'ku', 'gu': 'gu', 'h': '',
              'll': 'ʃ', 'y': 'ʃ', 'ph': 'p', 'th': 't', 'v': 'b',
              'z': 's', 'rr': 'r', 'r': 'ɾ', 'j': 'x',
              }.get(onset, onset)

    if nucleus[0] in 'ei' and not liquid:
```

```
        yield {'c': 's', 'g': 'x', 'gu': 'g', 'qu': 'k'}.get(onset, mapped)
    else:
        # Treat "guas" from "aguas" as the same "ua" nucleus as "tuan"
        # as "interactuando":
        if mapped.endswith('u'):
            mapped = mapped[:-1]
            nucleus = 'u' + nucleus

        # Treat "rra" from "tierra" the same as "rra" from "rápido":
        if liquid and not onset:

            mapped = 'r' if liquid == 'r' and first else 'ɾ' if liquid == 'r' else liquid
            liquid = ''

        yield mapped

    if liquid == 'r':
        yield 'ɾ'
    else:
        yield liquid

    yield {'w': 'u', 'y': 'i'}.get(nucleus, nucleus)

    yield {'c': 'k', 'z': 's', 'x': 'ks', 'r': 'ɾ', 'j': 'x'}.get(coda, coda)

def pronounce_word(word):
    return [list(pronounce(s, i == 0))
            for i, s in enumerate(syllabize(word))]

def print_counter(name, counter):
    print("%8s:" % name, end='')
    total = sum(v for k, v in counter.items())

    nl = False
    for i, (k, v) in enumerate(counter.most_common()):
        if nl:
            print(' ' * 9, end='')

        nl = i % 6 == 5
        print('%2s: %5.2f%%' % (k, 100 * v / total),
              end='\n' if nl else ' ')

    if not nl:
        print()

def main(stdin):
    onsets = collections.Counter()
    liquids = collections.Counter()
    nuclei = collections.Counter()
    codas = collections.Counter()
    phones = collections.Counter()
    letters = 0

    for line in stdin:
```

```
        for word in line.split():
            p = pronounce_word(word)
            print(word, '/%s/' % '-'.join(''.join(s) for s in p), end=' ')

            if not any(s == ['*'] for s in p):
                for s in p:
                    onsets[s[0]] += 1
                    liquids[s[1]] += 1
                    nuclei[s[2]] += 1
                    codas[s[3]] += 1
                    for c in ''.join(s):
                        phones[c] += 1
                letters += len(word)

        print()

    print()
    print_counter('onsets', onsets)
    print_counter('liquids', liquids)
    print_counter('nuclei', nuclei)
    print_counter('codas', codas)
    print_counter('phones', phones)
    t = sum(v for v in phones.values())
    entropy = sum(-phones[c]*math.log(phones[c]/t)/math.log(2) for c in phones)
    print("bits per phone", entropy/t)
    print("bits per letter", entropy/letters)

if __name__ == '__main__':
    main(sys.stdin)
```

Topics

- Experiment report (p. 1162) (14 notes)
- Python (p. 1166) (12 notes)
- Compression (p. 1263) (4 notes)
- Natural-language processing (p. 1284) (3 notes)
- Speech synthesis (p. 1322) (2 notes)

Some notes on learning Rust

Kragen Javier Sitaker, 02021-10-06 (updated 02021-10-10)
(39 minutes)

I want to learn Rust, so I'm reading the Rust book by Steve Klabnik, Carol Nichols, et al., and I'm going to try writing an IRC bot in it. I've done a few basic Rust tutorials in previous years, and I had a Rust compiler installed in /usr/local/bin, but it's from 02016.

The Rust Programming Language book is very approachable, but it's a bit slow-paced and patronizing. Maybe it would be great if I were extremely insecure about my abilities. The *Rust Reference* is maybe closer to what I want, but the 57-page *Rust for the Polyglot Programmer* is a night-and-day improvement over either as a starting point. For example, after only 23 pages, it tells me, "There is no inheritance,", and on the next page, "this is how `for x in y` loops work: `y` must `impl IntoIterator`". These are things I've been wondering about through hundreds of pages of TRPL. However, it is very much not self-contained, so it is only a starting point.

Installing Rust was kind of a pain in the ass and needed 294–1190 MB

First, I got rustup:

```
curl https://sh.rustup.rs > rustup.sh
```

Rustup insisted I uninstall the five-years-ago Rust, so I did:

```
sudo /usr/local/lib/rustlib/uninstall.sh
```

Then I tried installing rust, but because I "only" had half a gig free, it failed:

```
$ sh rustup.sh
info: downloading installer

Warning: Not enforcing strong cipher suites for TLS, this is potentially less secure
Warning: Not enforcing TLS v1.2, this is potentially less secure

Welcome to Rust!

This will download and install the official compiler for the Rust
programming language, and its package manager, Cargo.

Rustup metadata and toolchains will be installed into the Rustup
home directory, located at:

  /home/user/.rustup
...
1) Proceed with installation (default)
2) Customize installation
```

```
3) Cancel installation
>1

info: profile set to 'default'
...
info: installing component 'rust-docs'
 10.2 MiB /  17.0 MiB ( 60 %)   7.3 MiB/s in  1s ETA:  0s
info: rolling back changes

error: failed to extract package (perhaps you ran out of disk space?): No space l
eft on device (os error 28)
$ df -h .
Filesystem              Size  Used Avail Use% Mounted on
/dev/mapper/debian-root  225G  214G  517M 100% /
```

At this point I had deleted my previous Rust installation with no way to get it back, but wasn't able to install the current Rust.

I was spending 3.2 gigs on the linux-2.6 Git repo that I hadn't updated since 02014, so I deleted that. Even if half a fucking gigabyte isn't enough space for a fucking compiler, 3.7 gigs should be. That's four times the size of my first Linux box.

This time I tried the "minimal" profile instead, too. And it "only" needed 294 megs:

```
  stable-x86_64-unknown-linux-gnu installed - rustc 1.55.0 (c8dfcfe04 2021-09-06)

Rust is installed now. Great!

To get started you may need to restart your current shell.
This would reload your PATH environment variable to include
Cargo's bin directory ($HOME/.cargo/bin).

To configure your current shell, run:
source $HOME/.cargo/env
```

Before:

```
$ df -k .
Filesystem              1K-blocks      Used Available Use% Mounted on
/dev/mapper/debian-root 235891480 220134444   3774396  99% /
```

After:

```
$ df -k .
Filesystem              1K-blocks      Used Available Use% Mounted on
/dev/mapper/debian-root 235891480 220427872   3480968  99% /
```

Hmm, maybe I'll try a fatter profile then:

```
warning: Updating existing toolchain, profile choice will be ignored
```

Hmm, maybe not? I can't find the uninstall script this time (it turns out the command is `rustup self uninstall` as explained on p. 13 of

the book, which I hadn't gotten to yet) so I'll just delete it by hand:

```
$ rm -rf ~/.rustup ~/.cargo
$ df -k .
Filesystem             1K-blocks      Used Available Use% Mounted on
/dev/mapper/debian-root 235891480 220014156   3894684  99% /
$ sh rustup.sh
...
info: profile set to 'complete'
info: setting default host triple to x86_64-unknown-linux-gnu
info: syncing channel updates for 'stable-x86_64-unknown-linux-gnu'
info: latest update on 2021-09-09, rust version 1.55.0 (c8dfcfe04 2021-09-06)
warning: Force-skipping unavailable component 'miri-x86_64-unknown-linux-gnu'

warning: Force-skipping unavailable component 'rust-analyzer-preview-x86_64-unkno
wn-linux-gnu'
...
  stable-x86_64-unknown-linux-gnu installed - rustc 1.55.0 (c8dfcfe04 2021-09-06)
...
Rust is installed now. Great!
...
$ df -k .
Filesystem             1K-blocks      Used Available Use% Mounted on
/dev/mapper/debian-root 235891480 221200928   2707912  99% /
```

So this time it's using 1.19 gigs because I set the profile to `complete`.

hello, world

But now it's working:

```
: user@debian:~/devel/dev3; . ~/.cargo/env
: user@debian:~/devel/dev3; cat hello.rs
fn main() {
    println!("hello, {}", "world");
}
: user@debian:~/devel/dev3; rustc hello.rs
: user@debian:~/devel/dev3; ./hello
hello, world
```

Hello World is Fucking Huge

Smaller runtimes have fewer features but have the advantage of resulting in smaller binaries. Smaller binaries make it easier to combine the language with other languages in more contexts. While many languages are okay with increasing the runtime in exchange for more features, Rust needs to have nearly no runtime, and cannot compromise on being able to call into C in order to maintain performance.

— *The Rust Programming Language*, §4.1 "Using threads to run code simultaneously", p. 423

```
: user@debian:~/devel/dev3; ls -l hello
-rwxr-xr-x 1 user user 3439804 Oct  6 22:50 hello
```

That's a completely unreasonable size, roughly two and a half floppy disks for "hello, world", between three and five orders of magnitude larger than is needed, but it does run. And compiling it

takes about 250 milliseconds; again, three to five orders of magnitude slower than compiling a three-line program ought to be, but tolerable.

This is mostly (>90%) debug info. Unfortunately, the remainder is still almost 300K, between two and four orders of magnitude too big:

```
: user@debian:~/devel/dev3; ls -l hello
-rwxr-xr-x 1 user user 3439804 Oct  7 23:00 hello
: user@debian:~/devel/dev3; strip hello
: user@debian:~/devel/dev3; ls -l hello
-rwxr-xr-x 1 user user 297312 Oct  7 23:01 hello
```

Different optimization levels unsurprisingly don't make much difference:

```
: user@debian:~/devel/dev3; rustc -C opt-level=s hello.rs
: user@debian:~/devel/dev3; ls -l hello
-rwxr-xr-x 1 user user 3438548 Oct  7 23:03 hello
: user@debian:~/devel/dev3; rustc -C opt-level=z hello.rs
: user@debian:~/devel/dev3; ls -l hello
-rwxr-xr-x 1 user user 3438615 Oct  7 23:04 hello
: user@debian:~/devel/dev3; rustc -C opt-level=3 hello.rs
: user@debian:~/devel/dev3; ls -l hello
-rwxr-xr-x 1 user user 3438552 Oct  7 23:04 hello
: user@debian:~/devel/dev3; strip hello
: user@debian:~/devel/dev3; size hello
   text    data     bss     dec     hex filename
 281780   11288     576  293644   47b0c hello
```

Apparently I'd have to not use the prebuilt libstd to fix this, which requires nightly Rust, but that still leaves a 51-kilobyte executable, or use `#![no_std]` to not use libstd at all. Dynamically linking libstd *by default* isn't an option because Rust doesn't have an ABI, but you *can* dynamically link with `-C prefer-dynamic`, which gives you a 10-kilobyte stripped binary which by default doesn't work because it doesn't know where to find Rust's libstd:

```
: user@debian:~/devel/dev3; rustc -C prefer-dynamic hello.rs
: user@debian:~/devel/dev3; strip hello
: user@debian:~/devel/dev3; ls -l hello
-rwxr-xr-x 1 user user 10456 Oct  7 23:26 hello
: user@debian:~/devel/dev3; ldd hello
        linux-vdso.so.1 =>  (0x00007fff26df3000)
        libstd-008055cc7d873802.so => not found
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f05ead14000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f05ea987000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f05eb12d000)
: user@debian:~/devel/dev3; ./hello

./hello: error while loading shared libraries: libstd-008055cc7d873802.so: cannot
open shared object file: No such file or directory

: user@debian:~/devel/dev3; LD_LIBRARY_PATH=/home/user/.rustup/toolchains/stable-
x86_64-unknown-linux-gnu/lib ./hello
```

```
hello, world
```

That seems pretty reasonable.

There are, however, some other reasons that Rust compilation output is bloated:

```
0000000000003b00 <_ZN4core3ptr9const_ptr33_$LT$impl$u20$$BP$const$u20$T$GT$4cast17h2979c04ce50f48ccE>:
    3b00:       48 89 f8                mov    %rdi,%rax
    3b03:       c3                      retq
    3b04:       90                      nop
    3b05:       90                      nop
    3b06:       90                      nop
    3b07:       90                      nop
    3b08:       90                      nop
    3b09:       90                      nop
    3b0a:       90                      nop
    3b0b:       90                      nop
    3b0c:       90                      nop
    3b0d:       90                      nop
    3b0e:       90                      nop
    3b0f:       90                      nop
```

Holy shit, thirty thousand HTML files?

For some reason the `rustup doc` command just opens some kind of Wine error dialog telling me how to install Wine. But it looks like the docs are here:

```
: user@debian:~/devel/dev3; find /home/user/.rustup/ -name '*.html' | random 5000

/home/user/.rustup/toolchains/stable-x86_64-unknown-linux-gnu/share/doc/rust/html/core/arch/x86_64/fn._pdep_u32.html

/home/user/.rustup/toolchains/stable-x86_64-unknown-linux-gnu/share/doc/rust/html/core/arch/x86_64/fn._mm512_mask_reduce_add_pd.html

/home/user/.rustup/toolchains/stable-x86_64-unknown-linux-gnu/share/doc/rust/html/core/arch/aarch64/fn.vaddv_s32.html

/home/user/.rustup/toolchains/stable-x86_64-unknown-linux-gnu/share/doc/rust/html/core/arch/aarch64/fn.vaddl_s32.html

/home/user/.rustup/toolchains/stable-x86_64-unknown-linux-gnu/share/doc/rust/html/core/arch/aarch64/fn.vmlsl_u32.html

/home/user/.rustup/toolchains/stable-x86_64-unknown-linux-gnu/share/doc/rust/html/core/core_arch/arm_shared/neon/generated/fn.vqrdmlahq_laneq_s32.html

/home/user/.rustup/toolchains/stable-x86_64-unknown-linux-gnu/share/doc/rust/html/core/core_arch/x86/avx512vbmi2/fn._mm512_mask_compress_epi16.html

: user@debian:~/devel/dev3; find /home/user/.rustup/toolchains/stable-x86_64-unkn
```

```
own-linux-gnu/share/doc/rust/html/ | wc
  30223   30223 3819176
```

Thirty. Thousand. Files. *Of documentation alone*. Evidently, it's mostly one file per assembly-language instruction on any of the supported architectures. What have I done?

```
: user@debian:~/devel/dev3; find ~/.rustup ~/.cargo | wc
  31994   31994 4029266
```

Oh, I guess that's not so bad, then.

```
: user@debian:~/devel/dev3; firefox /home/user/.rustup/toolchains/stable-x86_64-u
nknown-linux-gnu/share/doc/rust/html/index.html
```

Well, that works. Nice comprehensive and polished documentation, too, looks like.

My very first crate

Let's try making a crate. I don't want to proliferate Git repos, because then I can forget to check things in or push them, so I'm using my standard hellbox repo:

```
: user@debian:~/devel/dev3; cargo new --vcs none hello_cargo
     Created binary (application) `hello_cargo` package
```

(The Rust book says to use `--bin` but `cargo new --help` says that's the default.)

```
: user@debian:~/devel/dev3; cd hello_cargo/
: user@debian:~/devel/dev3/hello_cargo; cat > hello.rs
fn main() {
        println!("hello, world");
}
: user@debian:~/devel/dev3/hello_cargo; ls
Cargo.toml  hello.rs  src
: user@debian:~/devel/dev3/hello_cargo; mv hello.rs src/.
: user@debian:~/devel/dev3/hello_cargo; mv src/hello.rs src/main.rs
: user@debian:~/devel/dev3/hello_cargo; cat Cargo.toml
[package]
name = "hello_cargo"
version = "0.1.0"
edition = "2018"


# See more keys and their definitions at https://doc.rust-lang.org/cargo/referenc
e/manifest.html

[dependencies]
: user@debian:~/devel/dev3/hello_cargo; cargo build
   Compiling hello_cargo v0.1.0 (/home/user/devel/dev3/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 0.72s
: user@debian:~/devel/dev3/hello_cargo; ls
```

```
Cargo.lock  Cargo.toml  src  target
: user@debian:~/devel/dev3/hello_cargo; find target/
target/
target/.rustc_info.json
target/debug
target/debug/.fingerprint
target/debug/.fingerprint/hello_cargo-cb8f156fc8def340
target/debug/.fingerprint/hello_cargo-cb8f156fc8def340/bin-hello_cargo
target/debug/.fingerprint/hello_cargo-cb8f156fc8def340/invoked.timestamp
target/debug/.fingerprint/hello_cargo-cb8f156fc8def340/bin-hello_cargo.json
target/debug/.fingerprint/hello_cargo-cb8f156fc8def340/dep-bin-hello_cargo
target/debug/incremental
target/debug/incremental/hello_cargo-3fhio3llrdrxv
target/debug/incremental/hello_cargo-3fhio3llrdrxv/s-g31wqj1itv-18kv17w.lock

target/debug/incremental/hello_cargo-3fhio3llrdrxv/s-g31wqj1itv-18kv17w-3u24lffwq
59u4

target/debug/incremental/hello_cargo-3fhio3llrdrxv/s-g31wqj1itv-18kv17w-3u24lffwq
59u4/3n8baekyl6jfd1zt.o

target/debug/incremental/hello_cargo-3fhio3llrdrxv/s-g31wqj1itv-18kv17w-3u24lffwq
59u4/dep-graph.bin

target/debug/incremental/hello_cargo-3fhio3llrdrxv/s-g31wqj1itv-18kv17w-3u24lffwq
59u4/5893w20ken98e8mr.o

target/debug/incremental/hello_cargo-3fhio3llrdrxv/s-g31wqj1itv-18kv17w-3u24lffwq
59u4/15pcyh12hnx9h9yu.o

target/debug/incremental/hello_cargo-3fhio3llrdrxv/s-g31wqj1itv-18kv17w-3u24lffwq
59u4/work-products.bin

target/debug/incremental/hello_cargo-3fhio3llrdrxv/s-g31wqj1itv-18kv17w-3u24lffwq
59u4/2zukcvf9271rij44.o

target/debug/incremental/hello_cargo-3fhio3llrdrxv/s-g31wqj1itv-18kv17w-3u24lffwq
59u4/3jf4rvldk0nwopmj.o

target/debug/incremental/hello_cargo-3fhio3llrdrxv/s-g31wqj1itv-18kv17w-3u24lffwq
59u4/query-cache.bin

target/debug/incremental/hello_cargo-3fhio3llrdrxv/s-g31wqj1itv-18kv17w-3u24lffwq
59u4/3vvwo10tkawer2dj.o

target/debug/incremental/hello_cargo-3fhio3llrdrxv/s-g31wqj1itv-18kv17w-3u24lffwq
59u4/497974iq30wb32q0.o

target/debug/incremental/hello_cargo-3fhio3llrdrxv/s-g31wqj1itv-18kv17w-3u24lffwq
59u4/2pe66p99jtgk2gt2.o

target/debug/incremental/hello_cargo-3fhio3llrdrxv/s-g31wqj1itv-18kv17w-3u24lffwq
59u4/4moexls4ruzzyfmb.o

target/debug/incremental/hello_cargo-3fhio3llrdrxv/s-g31wqj1itv-18kv17w-3u24lffwq
```

```
59u4/56vc5hmppbu1ww1q.o
target/debug/build
target/debug/.cargo-lock
target/debug/hello_cargo.d
target/debug/deps
target/debug/deps/hello_cargo-cb8f156fc8def340
target/debug/deps/hello_cargo-cb8f156fc8def340.d
target/debug/examples
target/debug/hello_cargo
target/CACHEDIR.TAG
: user@debian:~/devel/dev3/hello_cargo; ./target/debug/hello_cargo
hello, world
```

Hmm, seems okay. A bit voluminous, I guess, but that's a small price to pay if it speeds up builds and/or makes them more reliable.

Because I said `--vcs none` it didn't create a .gitignore, so I do:

```
: user@debian:~/devel/dev3/hello_cargo; echo target > .gitignore
```

Then I can add it to git, which I do, and then I can clone:

```
: user@debian:~/devel/dev3/hello_cargo; cd ../..
: user@debian:~/devel; time git clone dev3 dev3.copy
...
real    0m2.890s
...
: user@debian:~/devel; cd dev3.copy
: user@debian:~/devel/dev3.copy; cd hello_cargo/
: user@debian:~/devel/dev3.copy/hello_cargo; cargo build
   Compiling hello_cargo v0.1.0 (/home/user/devel/dev3.copy/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 0.72s
: user@debian:~/devel/dev3.copy/hello_cargo; ./target/debug/

build/       deps/         examples/     .fingerprint/ hello_cargo   incremental
/
: user@debian:~/devel/dev3.copy/hello_cargo; ./target/debug/hello_cargo
hello, world
```

Good enough. And it's nice that it records the versions of dependencies I'm building with in `Cargo.lock` by default.

There's a `cargo run`:

```
: user@debian:~/devel/dev3/hello_cargo; cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/hello_cargo`
hello, world
: user@debian:~/devel/dev3/hello_cargo; rm -rf target
: user@debian:~/devel/dev3/hello_cargo; cargo run
   Compiling hello_cargo v0.1.0 (/home/user/devel/dev3/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 0.71s
     Running `target/debug/hello_cargo`
hello, world
```

Oof, 710 ms to build a three-line program. Four lines of code

compiled per second. This is *not* going to be fun. Oddly, the release build happens faster, so possibly that was just a measurement error:

```
: user@debian:~/devel/dev3/hello_cargo; cargo run --release
   Compiling hello_cargo v0.1.0 (/home/user/devel/dev3/hello_cargo)
    Finished release [optimized] target(s) in 0.30s
     Running `target/release/hello_cargo`
hello, world
```

Cross-compiling depends on the C toolchain (and other things)

It seems like the Rust compiler I installed includes every Rust backend known to history or myth:

```
: user@debian:~/devel/dev3; rustc --print target-list| wc
    166     166    4022
: user@debian:~/devel/dev3; rustc --print target-list| random 32
mips64-unknown-linux-muslabi64
mipsisa64r6el-unknown-linux-gnuabi64
powerpc64-wrs-vxworks
x86_64-unknown-illumos
```

But because the binaries link with libc, you need to have a GCC or similar toolchain installed for the target platform:

```
: user@debian:~/devel/dev3; rustc --target s390x-unknown-linux-gnu hello.rs
error[E0463]: can't find crate for `std`
  |
  = note: the `s390x-unknown-linux-gnu` target may not be installed

  = help: consider downloading the target with `rustup target add s390x-unknown-l
inux-gnu`

error: aborting due to previous error

For more information about this error, try `rustc --explain E0463`.
: user@debian:~/devel/dev3; rustup target add s390x-unknown-linux-gnu
info: downloading component 'rust-std' for 's390x-unknown-linux-gnu'
info: installing component 'rust-std' for 's390x-unknown-linux-gnu'
 22.9 MiB /  22.9 MiB (100 %)  11.2 MiB/s in  1s ETA:  0s
: user@debian:~/devel/dev3; rustc --target s390x-unknown-linux-gnu hello.rs
error: linking with `cc` failed: exit status: 1
  |
  = note: "cc" "hello.hello.996e1e6f-cgu.0.rcgu.o" "hello.hello.996e1e6f-
...
e.rlib" "-Wl,-Bdynamic" "-lgcc_s" "-lutil" "-lrt" "-lpthread" "-lm" "-ldl" "
-lc" "-Wl,--eh-frame-hdr" "-Wl,-znoexecstack" "-L" "/home/user/.rustup/toolc
hains/stable-x86_64-unknown-linux-gnu/lib/rustlib/s390x-unknown-linux-gnu/li
b" "-o" "hello" "-Wl,--gc-sections" "-pie" "-Wl,-zrelro" "-Wl,-znow" "-nodef
aultlibs"

  = note: /usr/bin/ld: hello.hello.996e1e6f-cgu.0.rcgu.o: Relocations in generic
ELF (EM: 22)
```

```
    hello.hello.996e1e6f-cgu.0.rcgu.o: could not read symbols: File in wron
g format
    collect2: error: ld returned 1 exit status

error: aborting due to previous error
```

This failure left a debris of 11 hello.*.rcgu.o files built for the S/390, perhaps as a debugging aid.

Among the more exciting targets included are x86_64-fuchsia, wasm32-wasi, wasm32-unknown-emscripten, riscv32i-unknown-none-elf, riscv64gc-unknown-linux-gnu, nvptx64-nvidia-cuda, mipsel-sony-psp, arm-linux-androideabi, and avr-unknown-gnu-atmega328. I actually have the cross-compiling toolchain for the AVR, but trying to get Rust working for it fails in an excitingly different way:

```
: user@debian:~/devel/dev3; rustup target add avr-unknown-gnu-atmega328

error: toolchain 'stable-x86_64-unknown-linux-gnu' does not contain component 'ru
st-std' for target 'avr-unknown-gnu-atmega328'

note: not all platforms have the standard library pre-compiled: https://doc.rust-
lang.org/nightly/rustc/platform-support.html
```

(Here by "excitingly" I mean "disappointingly".)

Notes on things that surprised me about the language

I'd say "notes about the language" but I'm not going to attempt to describe the whole language, except very cursorily: the atomic ("scalar") types are {u,i}{8,16,32,64,size}, Unicode codepoints ("char"), f{32,64}, and boolean. Built-in aggregate types ("compound types" — oddly not "vector", which is a standard library growable array, as in the STL) are tuples, strings, arrays, structs (chapter 5), enums (ADTs, chapter 6), plus references, mutable references, and, rarely, pointers. Hmm, what about traits and functions? Looks like closure types are trait types (Fn, FnMut, FnOnce).

Some of what follows probably sounds critical and might inspire Rustaceans to feel defensive. I'd suggest they don't read it, because it's not about Rust; it's about me.

It's nice to be able to use underscores in numbers. Binary literals (0b101) are nice. Array literals [x, y, z] are nice. String formatting with println! (and format!, and even panic!) is nice. Snake case is nice. Array indexes are checked at runtime, panicking like .expect() when out of bounds. Type inference is nice, but unfortunately it doesn't extend to formal parameters or function return types, making the subroutine mechanism a more costly form of generalization than it would be. Implicit return and closure syntax, OTOH, reduce the cost of the subroutine mechanism, and it's nice that implicit return is just a special case of a more general progn mechanism. (Closure syntax

does receive the benefit of type inference.) Unparenthesized conditions in `if` and `while` are nice. Conditional expressions are nice, even if they do have to be made out of blocks. Not sure I like the `else if` special-case syntax, but I guess it's easy to read and remember. `for-in` is nice; not sure about the explicit `.iter()`. The `(1..4).rev()` syntax for a `Range` is nice.

I was thinking that maybe the cmp method from std::cmp::Ordering implied that there was no operator overloading, but evidently that's not true; `std::ops::Add<T>` is the trait of things that overload `+`. And `Vec` overloads `[]`, which is even better news for nefarious EDSL purposes. (Though Rust's macro system is probably a more capable way of doing EDSLs.)

In general the error messages are really excellent:

```
: user@debian:~/devel/dev3; rustc add.rs
error: return types are denoted using `->`
 --> add.rs:1:13
  |
1 | fn f(i: i32): i32 {
  |             ^ help: use `->` instead
```

Though not always:

```
thread 'main' panicked at 'index out of bounds: the len is 1
but the index is 1', /stable-dist-rustc/build/src/libcollections/
vec.rs:1307
```

That's... not a useful error location.

This is a very groovy way to *almost* implicitly propagate an exception:

```
fn run(config: Config) -> Result<(), Box<Error>> {
    let mut f = File::open(config.filename)?;
```

That sneaky little byte `?` means "return the result if it's an error".

I like the fact that each file forms a namespace of its own by default. I dislike the fact that apparently the crate name has nothing to do with the filename.

I wonder if instead of a `&` sigil for borrowing an immutable reference and no sigil for consumption or copying (the difference between them being only whether the object has the `Copy` trait) the unmarked case should be borrowing an immutable reference, while copying and consumption each have their own sigils. Mina suggested that consumption should use an arrow; instead of `let s2 = s1` you could say `let s2 ← s1` to emphasize the "movement" aspect of the value; in other consumption contexts (arguments, returns) that wouldn't quite work, but `let s2 = ←s1` would.

Syntactically, I am not a fan of the paamayim nekudotayim, but I guess it could be worse; VMS used $.

It's interesting that library functions are private (like C file `static`, I guess?) by default, if you don't prefix them with `pub`. `pub fn foo`, etc.

Recursive deref coercion for arguments surprised me.

Unhandled results are just a warning

Unhandled result failures warn by default, which is nice:

```
: user@debian:~/devel/dev3; rustc greet.rs
warning: unused `Result` that must be used
 --> greet.rs:6:4
  |
6 |    io::stdin().read_line(&mut s);
  |    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: `#[warn(unused_must_use)]` on by default
  = note: this `Result` may be an `Err` variant, which should be handled

warning: 1 warning emitted
```

But it was only a warning:

```
: user@debian:~/devel/dev3; ./greet
hi, what is your name?
bob
hello, bob
!
```

There's a linter standardly installed with the compiler called "Clippy"; I'm not sure if this is a Clippy warning or not.

Unicode handling? Well, shit, at least it's not Python

The Rust book says:

Note: `std::env::args` will panic if any argument contains invalid Unicode. If you need to accept arguments containing invalid Unicode, use `std::env::args_os` instead. That function returns OsString values instead of String values. We've chosen to use `std::env::args` here for simplicity because `OsString` values differ per-platform and are more complex to work with than `String` values.

So, on the plus side, at least command-line argument handling isn't *completely* broken in order to enable portability to broken operating systems. On the other hand, the easiest interface to command-line argument apparently *is* broken. I don't understand why I should suffer because other people use Microsoft Windows.

But hey. At least I think my Rust programs won't crash while attempting to print a crash traceback because the traceback contains a non-ASCII character, which is actually a thing that has happened to me with Python 3. And probably there won't be files I can't open in Rust because their names aren't UTF-8. And I'm pretty sure my Rust programs won't stop compiling if I put curly quotes inside my comments, which happened to me a lot in Python 2.

Strings

The str/String distinction is a bit of a hassle. Nice that str (and maybe String?) has .lines() iterator and a .contains() method, and that String (and maybe str?) can be sliced by bytes. .to_lowercase() is a longish method name but not commonly enough used to merit a more ambiguous name.

Iterators

Python's iterator design is one of its strongest points, and Rust's iterator design is one of *its* strongest points. Both are external iterators (they don't receive a closure to evaluate on each item, so you can build fairly general converging dataflow trees from them). And they are very similar, consisting of only a `next()` method (renamed `__next__()` in Python 3 for dubious reasons) that either returns the next item or fails (with None in Rust, StopIteration in Python) and implicitly mutates the iterator.

It's interesting that `Vec` "is an iterator" (you can directly iterate over it with for-in) but the book implies some built-in collections aren't; you need to call .iter() on them. Though, which built-in collections were they? Arrays evidently *can* be directly iterated over. The default way of iterating over `Vec` is I think its .iter() method; it also has `.into_iter()`, which consumes the vec, and `.iter_mut()`, which returns an iterator of mutable references.

The `next()` method in the `Iterator` trait takes a mutable self reference, which makes it surprising that you can usefully make an immutable iterator reference.

Python added generators fairly soon after iterators, allowing you to implement an iterator as a coroutine, which greatly improved the clarity of iterator transformation. Soon after that it added generator expressions, which are still terser.

Python's iterators are somewhat bug-prone because you can confuse them with collections and attempt to use them again after they've already been fully consumed, in which case they will generally appear to be empty. Java's iterator design solved this by not treating iterators themselves as iterable, at the cost of not being able to deal with sequences like lines from an input file. I think this bug-proneness is less of a concern in Rust because normally anything that iterates over an iterator will consume and drop the iterator; it won't be satisfied with a borrowed mutable reference.

Another bug that Rust is better at detecting than Python is creating a lazy iterator and then never consuming it, because at least iterator adaptors are `#[warn(unused_must_use)]`, like Err.

Because Rust has traits instead of just protocols, things like map(), filter(), enumerate(), zip(), sum(), reduce() (called .fold()), collect() (like Python `list()`, `dict()`, etc.), and skip() (like APL drop I guess) are methods on the iterator trait with default implementations, not functions in a global namespace. This helps to reduce nesting compared to Python, though a Python genex is still usually shorter and clearer.

Interestingly, both .collect() and .sum() have ad-hoc polymorphism on their return type, similar to Perl's scalar vs. list context, but generalized. Any type that implements the FromIterator trait can be returned from .collect(); any type that implements Sum can be returned from .sum() (and similarly for Product and .product()).

There is some implicit lifting into the Result and Option monads for, e.g., .sum() and .product().

In addition to what STL calls input and output (see below!) iterators, I think Rust iterators can be forward iterators (by implementing Copy or Clone) and random-access iterators (with the

`Step` trait).

Writing through iterators

Because Rust iterators can yield mutable references, you can use them as cursors into data structures you're mutating as well, like C++ forward iterators. This is something Python iterators can't do. This took me 20 minutes of struggling through compiler errors, but I did finally get it to work:

```
fn copy_iter<T: Copy>(src: &mut dyn Iterator<Item=&T>,
                      dest: &mut dyn Iterator<Item=&mut T>) {
    loop {
        match (src.next(), dest.next()) {
            (Some(s), Some(d)) => *d = s.clone(),
            (_, _) => return,
        }
    }
}

fn main() {
    let mut v1 = vec![3, 4, 1];
    let v2 = vec![10, 20, 100];
    let mut i = v1.iter_mut();
    i.next();

    copy_iter(&mut v2.iter(), &mut i);
    println!("Now it's {:?}", v1);      // outputs: Now it's [3, 10, 20]
}
```

That, uh, doesn't really emit reasonable code for `copy_iter`, though. It *does* get specialized for the `i32` integers it's being invoked with, but, oddly enough, not for vector iteration, presumably because of `dyn`:

```
0000000000002cb0 <_ZN4iter9copy_iter17h5fd7a53461d29648E>:
    2cb0:       48 83 ec 58             sub    $0x58,%rsp
    2cb4:       48 89 7c 24 28          mov    %rdi,0x28(%rsp)
    2cb9:       48 89 74 24 30          mov    %rsi,0x30(%rsp)
    2cbe:       48 89 54 24 38          mov    %rdx,0x38(%rsp)
    2cc3:       48 89 4c 24 40          mov    %rcx,0x40(%rsp)
    2cc8:       48 8b 44 24 30          mov    0x30(%rsp),%rax
    2ccd:       48 8b 7c 24 28          mov    0x28(%rsp),%rdi
    2cd2:       ff 50 18                callq  *0x18(%rax)
    2cd5:       48 89 44 24 20          mov    %rax,0x20(%rsp)
    2cda:       48 8b 44 24 40          mov    0x40(%rsp),%rax
    2cdf:       48 8b 7c 24 38          mov    0x38(%rsp),%rdi
    2ce4:       ff 50 18                callq  *0x18(%rax)
    2ce7:       48 89 44 24 18          mov    %rax,0x18(%rsp)
    2cec:       48 8b 44 24 18          mov    0x18(%rsp),%rax
    2cf1:       48 8b 4c 24 20          mov    0x20(%rsp),%rcx
    2cf6:       48 89 4c 24 48          mov    %rcx,0x48(%rsp)
    2cfb:       48 89 44 24 50          mov    %rax,0x50(%rsp)
    2d00:       b8 01 00 00 00          mov    $0x1,%eax
    2d05:       31 c9                   xor    %ecx,%ecx
    2d07:       48 83 7c 24 48 00       cmpq   $0x0,0x48(%rsp)
    2d0d:       48 0f 44 c1             cmove  %rcx,%rax
```

```
    2d11:       48 83 f8 01             cmp    $0x1,%rax

    2d15:       75 17                   jne    2d2e <_ZN4iter9copy_iter17h5fd7a53○
○461d29648E+0x7e>
    2d17:       b8 01 00 00 00          mov    $0x1,%eax
    2d1c:       31 c9                   xor    %ecx,%ecx
    2d1e:       48 83 7c 24 50 00       cmpq   $0x0,0x50(%rsp)
    2d24:       48 0f 44 c1             cmove  %rcx,%rax
    2d28:       48 83 f8 01             cmp    $0x1,%rax

    2d2c:       74 05                   je     2d33 <_ZN4iter9copy_iter17h5fd7a53○
○461d29648E+0x83>
    2d2e:       48 83 c4 58             add    $0x58,%rsp
    2d32:       c3                      retq
    2d33:       48 8b 7c 24 48          mov    0x48(%rsp),%rdi
    2d38:       48 8b 44 24 50          mov    0x50(%rsp),%rax
    2d3d:       48 89 44 24 08          mov    %rax,0x8(%rsp)

    2d42:       e8 09 0e 00 00          callq  3b50 <_ZN4core5clone5impls52_$LT$i○
○mpl$u20$core..clone..Clone$u20$for$u20$i32$GT$5clone17h4244c5f4dce8d8e8E>
    2d47:       89 44 24 14             mov    %eax,0x14(%rsp)
    2d4b:       48 8b 44 24 08          mov    0x8(%rsp),%rax
    2d50:       8b 4c 24 14             mov    0x14(%rsp),%ecx
    2d54:       89 08                   mov    %ecx,(%rax)

    2d56:       e9 6d ff ff ff          jmpq   2cc8 <_ZN4iter9copy_iter17h5fd7a53○
○461d29648E+0x18>
    2d5b:       90                      nop
    2d5c:       90                      nop
    2d5d:       90                      nop
    2d5e:       90                      nop
    2d5f:       90                      nop
```

I mean, reading through the code, it's not *totally* appalling, but does this function really need an almost-90-byte stack frame? And what's going on here?

```
    2ce7:       48 89 44 24 18          mov    %rax,0x18(%rsp)
    2cec:       48 8b 44 24 18          mov    0x18(%rsp),%rax
```

And this in particular is kind of an embarrassing way to compile `*d = s.clone()` in a production compiler optimizing for size:

```
    2d33:       48 8b 7c 24 48          mov    0x48(%rsp),%rdi
    2d38:       48 8b 44 24 50          mov    0x50(%rsp),%rax
    2d3d:       48 89 44 24 08          mov    %rax,0x8(%rsp)

    2d42:       e8 09 0e 00 00          callq  3b50 <_ZN4core5clone5impls52_$LT$i○
○mpl$u20$core..clone..Clone$u20$for$u20$i32$GT$5clone17h4244c5f4dce8d8e8E>
    2d47:       89 44 24 14             mov    %eax,0x14(%rsp)
    2d4b:       48 8b 44 24 08          mov    0x8(%rsp),%rax
    2d50:       8b 4c 24 14             mov    0x14(%rsp),%ecx
    2d54:       89 08                   mov    %ecx,(%rax)
```

I'd think something like this would be more reasonable:

```
    mov 0x48(%rsp), %rdi    # s

    callq _ZN4core5clone5impls52_$LT$impl$u20$core..clone..Clone$u20$for$u20$i32$
GT$5clone17h4244c5f4dce8d8e8E #WTAF
    mov 0x50(%rsp), %rcx    # d
    mov %eax, (%rcx)        # *d = ...
```

That's with `-C prefer-dynamic -C opt-level=s`. Without the optimization the executable is three times the size. opt-level=3 doesn't help but opt-level=1 is actually a little better, except that its invocation of the `next()` method is much worse:

```
00000000000013a0 <_ZN4iter9copy_iter17h5fd7a53461d29648E>:
    13a0:       41 57                   push   %r15
    13a2:       41 56                   push   %r14
    13a4:       41 54                   push   %r12
    13a6:       53                      push   %rbx
    13a7:       50                      push   %rax
    13a8:       49 89 f6                mov    %rsi,%r14
    13ab:       49 89 ff                mov    %rdi,%r15
    13ae:       66 90                   xchg   %ax,%ax

    13b0:       e8 8b ff ff ff          callq  1340 <_ZN91_$LT$core..slice..iter.
.Iter$LT$T$GT$$u20$as$u20$core..iter..traits..iterator..Iterator$GT$4next17h074db
47cc7af8891E>
    13b5:       49 89 c4                mov    %rax,%r12
    13b8:       4c 89 f7                mov    %r14,%rdi

    13bb:       e8 b0 ff ff ff          callq  1370 <_ZN94_$LT$core..slice..iter.
.IterMut$LT$T$GT$$u20$as$u20$core..iter..traits..iterator..Iterator$GT$4next17h50
a12d7708b22495E>
    13c0:       4d 85 e4                test   %r12,%r12

    13c3:       74 17                   je     13dc <_ZN4iter9copy_iter17h5fd7a53
461d29648E+0x3c>
    13c5:       48 89 c3                mov    %rax,%rbx
    13c8:       48 85 c0                test   %rax,%rax

    13cb:       74 0f                   je     13dc <_ZN4iter9copy_iter17h5fd7a53
461d29648E+0x3c>
    13cd:       4c 89 e7                mov    %r12,%rdi

    13d0:       e8 ab fd ff ff          callq  1180 <_ZN4core5clone5impls52_$LT$i
mpl$u20$core..clone..Clone$u20$for$u20$i32$GT$5clone17hb0e95370c1e5efa8E>
    13d5:       89 03                   mov    %eax,(%rbx)
    13d7:       4c 89 ff                mov    %r15,%rdi

    13da:       eb d4                   jmp    13b0 <_ZN4iter9copy_iter17h5fd7a53
461d29648E+0x10>
    13dc:       48 83 c4 08             add    $0x8,%rsp
    13e0:       5b                      pop    %rbx
    13e1:       41 5c                   pop    %r12
    13e3:       41 5e                   pop    %r14
```

```
    13e5:       41 5f                   pop    %r15
    13e7:       c3                      retq
```

(Maybe all those extra `movs` disappear into register renaming in early stages of execution, though.)

(On the plus side, compiling this 17-line program at any optimization level takes 280–290 ms, barely longer than the 230 ms to compile the three-line hello-world program. So it's compiling... about 300 lines a second? Probably that's just happenstance and the actual amount of code is a minimal factor here.)

The explicit call to .iter() is necessary; maybe coercion to iterators happens automatically in for-in loops for `Vec`, but not here:

```
error[E0277]: `Vec<{integer}>` is not an iterator
  --> iter.rs:17:15
   |
17 |     copy_iter(&mut v2, &mut i);
   |               ^^^^^^^ `Vec<{integer}>` is not an iterator
   |
   = help: the trait `Iterator` is not implemented for `Vec<{integer}>`
   = note: required for the cast to the object type `dyn Iterator<Item = &_>`
```

The body of the loop is stupid, though, because it's explicitly calling .clone() on a `Copy` instance; it should instead say

```
match (src.next(), dest.next()) {
    (Some(s), Some(d)) => *d = *s,
    (_, _) => return,
}
```

and, with this fix, the function is inlined into main() as it should be, and fully unrolled, and I think maybe dead-store-eliminated as well. This also works:

```
while let (Some(s), Some(d)) = (src.next(), dest.next()) {
    *d = *s;
}
```

And so does this:

```
for (s, d) in src.zip(dest) {
    *d = *s;
}
```

FFI Callability

One of the major draws of Rust for me is interoperability: being able to call code from other languages and being able to call code in other languages.

It's not obvious how you invoke the Rust compiler to build a .o file you can link with C, though. All in all this seems like an underdocumented aspect of Rust.

The following seems to work (see SO question), but involves compiling four lines of code into a 20-megabyte library which adds 4.7 megs to the binary, and adds dependencies on libpthreads, libdl, libm, and librt to the C code:

```
: user@debian:~/devel/dev3; cat add2.rs
#[no_mangle]
pub extern "C" fn add(a: i32, b: i32) -> i32 {
    a + b
}
: user@debian:~/devel/dev3; rustc --crate-type=staticlib add2.rs
: user@debian:~/devel/dev3; ls -l libadd2.a
-rw-r--r-- 1 user user 19493732 Oct  7 23:51 libadd2.a
: user@debian:~/devel/dev3; cat calladd2.c
#include <stdio.h>

int add(int a, int b);          /* prototype for function written in Rust */

int main(int argc, char **argv) {
  printf("3 + 4 = %d\n", add(3, 4));
  return 0;
}
: user@debian:~/devel/dev3; cc -L. calladd2.c -ladd2 -lpthread -ldl -lm -lrt
: user@debian:~/devel/dev3; ls -l a.out
-rwxr-xr-x 1 user user 4689773 Oct  7 23:52 a.out
: user@debian:~/devel/dev3; ./a.out
3 + 4 = 7
```

(It sort of works with `cc -static` but gives terrifying warnings.)

So it seems like doing this in practice would involve doing some of the things mentioned in the “Hello World is Fucking Huge” section above. Until your library is hundreds of thousands of lines of code, anyway.

Fontdue is a TrueType rasterizer written this way (a `no_std` crate) to facilitate calling from C. It seems like I could probably learn a lot from things like that about how to pull this off.

However, it’s notable that building libraries like this evidently doesn’t rely on having a working GCC toolchain, so cross-compiling is easier for building C-callable libraries than for building executables:

```
: user@debian:~/devel/dev3; rustc --crate-type=staticlib \
    --target s390x-unknown-linux-gnu add2.rs
: user@debian:~/devel/dev3; ls -l libadd2.a
-rw-r--r-- 1 user user 37002666 Oct  8 00:13 libadd2.a
: user@debian:~/devel/dev3; ar tv libadd2.a
rw-r--r-- 0/0   1640 Dec 31 21:00 1969 add2.add2.a3d9fba4-cgu.0.rcgu.o
rw-r--r-- 0/0   2288 Dec 31 21:00 1969 add2.1o36m3z73gy3kp52.rcgu.o
...[188 lines omitted]...
: user@debian:~/devel/dev3; ar x libadd2.a add2.add2.a3d9fba4-cgu.0.rcgu.o
: user@debian:~/devel/dev3; ls -l add2.add2.a3d9fba4-cgu.0.rcgu.o
-rw-r--r-- 1 user user 1640 Oct  8 00:14 add2.add2.a3d9fba4-cgu.0.rcgu.o
: user@debian:~/devel/dev3; file add2.add2.a3d9fba4-cgu.0.rcgu.o
```

```
add2.add2.a3d9fba4-cgu.0.rcgu.o: ELF 64-bit MSB relocatable, IBM S/390, version 1
(SYSV), not stripped
```

I don't have cross-platform binutils installed, though:

```
: user@debian:~/devel/dev3; objdump -d add2.add2.a3d9fba4-cgu.0.rcgu.o

add2.add2.a3d9fba4-cgu.0.rcgu.o:     file format elf64-big

objdump: can't disassemble for architecture UNKNOWN!
```

Creature comforts and affordances

I'd really like to have Hypothesis. Rik de Kort has ported minithesis but doesn't recommend using it; he recommends the Hypothesis-inspired proptest (docs) or quickcheck instead, which latter is by BurntSushi (Andrew Gallant, the ripgrep guy) and also comes recommended by DRMacIver. There are efforts to provide proptest via symbolic execution in KLEE.

It's nice that there's a standard test setup: the `#[cfg(test)]` attribute on a `mod`, the `#[test]` attribute on each test function, the `assert!` macro (or just `panic!`), and `cargo test` to run the lot (implicitly all in parallel!). I don't think the Rust book's recommendation to put the tests `mod` in `src/lib.rs` is optional or not; XXX try it. I like the recommendation to put unit tests in the same file as the implementation; I guess Cargo enforces the putting of integration tests in a `tests/` directory and `extern crate` importing your library module? XXX try a different directory.

Deep equality and deep printing

One of the big advances in Python over Perl for me was deep equality and printing by default (for lists, tuples, and dicts), The semantics of equality used by `assert_eq!` are those of ==, which comes from the `PartialEq` trait. As with printing, Rust doesn't do the deep comparison thing for structs and enums unless you opt into it with `#[derive(..., PartialEq)]`. Not sure yet about the semantics of these with built-in arrays, slices, tuples, and hash maps. XXX try it. Vec evidently has a useful debug print format.

Vec and std::collection::HashMap at least do the deep equality thing by default. Given this code:

```
let xs = vec![3, 8, 12];

let mut ys = vec![3, 8];
ys.push(13);
assert_eq!(xs, ys);
```

We get this behavior:

```
: user@debian:~/devel/dev3; ./veciter
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `[3, 8, 12]`,
 right: `[3, 8, 13]`', veciter.rs:6:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

And similarly for HashMap. It formats okay with {:?} too.

Backtraces

It'd be nice to have a stack data dump like Python cgitb, but I'm not sure to what extent that's implementable in Rust. With RUST_BACKTRACE=1 in the environment, you do get some kind of backtrace, but it doesn't display the values of local variables, and if you compile without -g it won't even show you the line number in your code where it failed:

```
: user@debian:~/devel/dev3; RUST_BACKTRACE=1 ./veciter
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `[3, 8, 12]`,
 right: `[3, 8, 13]`', veciter.rs:6:5
stack backtrace:
   0: rust_begin_unwind

             at /rustc/c8dfcfe046a7680554bf4eb612bad840e7631c4b/library/std/src/p
panicking.rs:515:5
   1: core::panicking::panic_fmt

             at /rustc/c8dfcfe046a7680554bf4eb612bad840e7631c4b/library/core/src/
panicking.rs:92:14
   2: core::panicking::assert_failed_inner
   3: core::panicking::assert_failed
   4: veciter::main
   5: core::ops::function::FnOnce::call_once

note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose back
trace.
```

The suggested RUST_BACKTRACE=full gives you more stack frames, machine-code addresses, and compilation hashes, but not more variables, e.g.,

```
  17:     0x7ff3987a1c61 - veciter::main::hdfeb52505aea83ac
                               at /home/user/devel/dev3/veciter.rs:6:5
```

This would be useful if I were debugging the compiler or build system but not if the bug is in my code.

Etc.

Failing full backtraces, what does the debugger look like? Evidently (Rust's fork of) GDB and (Rust's MacOS-only fork of) LLDB are supported, and Tom Tromey has been working on it, but DWARF can't represent traits yet.

There are some wrappers installed by Rustup (or Cargo?) that don't work:

```
: user@debian:~/devel/dev3; rust-gdb
gdb: unrecognized option '-iex'
Use `gdb --help' for a complete list of options.
: user@debian:~/devel/dev3; rust-lldb
```

```
lldb not found! Please install it.
```

You'd think there would be a `rustfilt` analogous to `c++filt` for the name mangling, but there doesn't seem to be.

There's a crate called coredump to dump core on panic, which is potentially a useful alternative to full backtraces, if you have a working debugger, anyway.

Printf debugging in tests is feasible but requires `cargo test -- --nocapture`.

A lot of the things I'm accustomed to in the Python standard library (JSON, XML, HTTP) aren't in the Rust standard library; you're supposed to get them from Cargo. But which crates (packages) do I use in Cargo for these things? For example, apparently ureq is a lot smaller than reqwest for HTTP.

I guess one possibility is to look at exemplary Rust projects and see what dependencies *they* use. ripgrep, for example, has 46 dependencies (!). Among them are the FNV hash function used by the Rust compiler, `atty` (which provides various OS-specific cversions of `isatty`), `libc` (a wrapper around libc), `itoa` (a faster version than the `fmt::Formatter` version), `memmap2` (a fork of memmap-rs, supporting mmap and similar facilities on other OSes), `ryu` (for float-to-string conversion), and `serde` (similar to pickle). This gives something of a flavor of the stuff left out of the standard library.

Rust for the Polyglot Programmer recommends crates called `slab`, `slotmap`, and `generational_arena` for memory management; `itertools`; the locking-primitives crate `parking_lot`; the `tokio` runtime for async programs; the alternative `smol`; `pin-project` and `pin-project-lite` for dealing with some obscure async problems; `futures`; `cxx`, for calling C++; `inline-python` and `pyo3` for calling Python; `wasm-bindgen`, `web-sys`, and `rusty_v8` for WASM and the web; `j4rs` and `jni` for calling Java; `fehler`, `thiserror`, `eyre`, and/or `anyhow` for error handling; `num`, `num-traits`, and `num-derive` for numerical code and integer conversion; `index_vec`, `arrayvec`, and `indexmap` for containers; `easy-ext`; `rayon` and `crossbeam` for multi-thread parallelism; `chrono` and `chrono-tz` for datetime; `libc` or `nix`; `lazy_static` and `once_cell`; `log`; `tracing`; `regex`; `lazy-regex`; `glob`; `tempfile`; `rand` (also recommended by TRPL); `either`; `void`; `ndarray`; `ndarray-linalg`; `ring`; `rustls`; `bstr`; `bytemuck`; `serde`, mentioned above, but also with the objective of data *interchange with other languages*, saying they are "considerably better for many tasks than anything available in any other programming environment"; `reqwest` or `ureq`; `hyper` for raw HTTP; `rocket`, `actix-web`, `rouille` (sync), or `warp` as a web server framework; `structop` and `clap` or `argparse` for command-line parsing; etc. It also suggests looking at "recent downloads" on crates.io to see what other people are using. It specifically recommends avoiding `wasm-pack` and `stdweb`.

I think the easiest way to make Cargo get the source for a package is to add it as a dependency to a project.

Topics

- Programming (p. 1141) (49 notes)
- Rust

PBKDF content addressing with keyphrase hashcash: a non-blockchain attack on Zooko's Triangle

Kragen Javier Sitaker, 02021-10-08 (24 minutes)

This note provides a cryptographically secure, moderately decentralized way to link moderately human-memorable strings like b8://grain-more-state-court/GenevievesDeliciousPies or b8://lucy-cure-tommy-rinse/Francisco.H.Walz to public keys or other data, without a blockchain or other consensus-arbitration mechanism, at the cost of a few thousand dollars of proof-of-work for each four-word keyphrase such as b8://grain-more-state-court/ or b8://lucy-cure-tommy-rinse/, which creates a namespace of unlimited size that can be securely delegated. Brute-force attacks on the system's cryptographic integrity are calculated to have infeasibly large costs under standard assumptions, while intermediaries can efficiently verify the integrity of data they convey, preventing the propagation of malicious or accidentally corrupted data.

I was writing Wordlists for maximum drama (p. 904), and I was thinking about how a password using a reasonable PBKDF can be a lot shorter than a secure content hash, even one that's only secure against second-preimage attacks. For example, in "How you should set up a full-disk-encryption passphrase on a laptop", I recommended encrypting your disk with 2^{16} iterations of PBKDF2 (because LUKS doesn't support scrypt, though apparently LUKS2 has since added argon2i support) and a 72-bit-of-entropy six-word passphrase.

This suggests the idea: can we use a PBKDF for content-addressing by hash?

It turns out that not only can you use a PBKDF like scrypt to expand a human-memorable password into a larger, strong hash; you can also use a PBKDF to compress a larger, strong hash into a human-memorable keyphrase. This gives you a decentralized, human-*memorable*, secure naming system, but the resulting names are generally not human-*meaningful*.

It turns out that a hashcash-like approach to keyphrase stretching is in some ways much better than an approach that relies only on the cost-factor parameter of PBKDFs.

Using a PBKDF for content-addressing by hash

You hash a massive blob of content, or maybe somebody's public key, using something fast like SHA-256 or BLAKE3, then run the result through a memory-hard PBKDF like scrypt or Argon2, then encode the result with some kind of human-readable word list, like the S/Key or Diceware word lists. This could potentially greatly reduce the human effort involved in using such hashes, at the expense

of computer effort.

Memory-hard key derivation functions are designed to increase the cost of a brute-force attack by requiring large, fast memory. Right now a 4 GiB RAM stick costs US$15 and is normally depreciated over 3 years, or US$5 per year, which works out to about 160 nanodollars per second for 4 GiB or 4.6 attodollars per bit-second. Probably in practice the cost of using this memory is around four times this because of the cost of the non-RAM parts of the computer and of the electrical energy to run it, but this conservative 4.6 attodollars per bit-second is the estimate I'll use below.

Suppose we assume that the crypto works according to the standard conjectures, and we're willing to demand that a legitimate participant in the system employ 4 GiB of RAM for 128 seconds to verify a hash, which is 20 microdollars for an attacker with warehouse-scale computers stuffed full of RAM cracking hashes. Suppose we are content with resisting up to a quadrillion dollars' worth of attack (many years of current human economic productivity), a quadrillion dollars buys you 2.2×10^{32} bit-seconds of RAM, or 4.9×10^{19} of these 128-second-4-GiB hashes. That's about $2^{65.4}$ hashes. So to resist a second-preimage attack, a hash would only need 66 bits.

Therefore, 2^{72} possible human-readable hash identifiers would be adequate to resist a quadrillion-dollar attack. A random 72-bit number like 3943234810267769014593 can be represented in many different ways, including the following:

```
In hex: d5c3 603f c8fd 3889 41

In the first half of the alphabet to avoid cusswords, like Chrome: cjakmgljchceddegilkk
In lowercase letters, ideal for a cellphone keyboard: cjdhebmuvttqrjlx
In lowercase letters and digits: n46ukw493t154x
In letters and digits: 1dMffgtrOWF5T
In letters, digits, hyphen, and underscore, like YouTube: RsdwfYzZe8B1
In printable ASCII: j41'/~4%s6t
In 12-bit words of 5 letters or less: todd guide port ros iron tubes
In the S/Key word list: BEE DEAF MULL WIRE FEAT SONG MEN
```

I think that, of these, at least "Todd guide port ROS iron tubes" and "bee deaf mull wire feat song men" are things that a person could reasonably memorize, though not without effort.

If instead of a quadrillion dollars we're willing to succumb to single-target second-preimage attacks of a billion dollars or more, we can drop 20 bits. 46 bits of hash instead of 66 bits: "pork bring extra asset" or "ni death col batty", say.

I'm assuming here that the work factor of the PBKDF in question (the number of iterations or whatever) is fixed up front and doesn't have to be incorporated into the human-memorized key. But it wouldn't have to be; you could reserve, for example, 4 bits of the human-memorable key for the natural logarithm of the amount of work required in gigabyte-seconds, thus permitting adjustment of the time required to try a single hash from 1 gigabyte-second to 3269000 gigabyte-seconds. This involves making the phrases longer on average, but note that my "46-bit" phrases above already have 2 bits

of slack, and my 72-bit phrases have 6 bits of slack.

In many scenarios, the document being identified by such an identifier is necessarily public, and so is the algorithm, so that anyone can spend 128 seconds to verify that the document produces the correct hash. This means that an attacker knows not only the desired (truncated) PBKDF *output* but also the PBKDF *input*, which is the file's hash using BLAKE3 or SHA-256 or whatever. If the attacker can find a second preimage of *that* hash, they don't need to run the PBKDF even once — they just run the other, much faster hash function, potentially a much larger number of times. But the 256-bit output size of BLAKE3 and SHA-256 is large enough to make that infeasible, more than compensating for the enormously higher speed of these hash functions. (This is the same property that makes client offload secure for password authentication.)

A useful aspect of content-addressable schemes in general is that network intermediaries providing a storage service can authenticate that files they cache are valid — they haven't been corrupted either by a malicious attacker or by, for example, cosmic rays hitting RAM. If such authentication is infeasible or too expensive, they won't do it, and so a querent requesting a file by content-hash from an intermediary is likely to get useless data that they eventually reject.

With the scheme described above, verifying the hash costs 512 GiB-seconds of work, so we'd like the granularity of hashed files to be fairly large. If there are 8192 intermediaries and 524288 end-users in the world, and you do 512 GiB-seconds of work to compute an independent phrase for a file you want to publish on that network, instead of just getting someone else to include its hash in their own file, in some sense you're imposing 512×(8192+524288) GiB-seconds of work on the rest of the world. Maybe it would be nice if there was some way the original publisher could shoulder a disproportionate share of that security burden. As it turns out, there is; see below about leading-zeroes phrase stretching.

As explained later, having a large number of files identified by keyphrases in this way would make the system more vulnerable to brute-force attacks, so it is desirable to use the human-memorable hash for only the root of some kind of large Merkle DAG. It might contain, for example, a public key for everyone in your city, or the entire corpus of Project Gutenberg or Wikipedia. This property gives the protocol a flavor more anarcho-syndicalist than individualist.

(Of course, the hashes connecting the nodes in the Merkle DAG should be normal, full-sized hashes of 256 bits or so, not these expensively stretched hashes.)

If the hashed data is some kind of efficiently traversable namespace, it can assign human-memorable names to other particular blobs, including public keys, IP addresses, and other identifiers. So instead of identifying someone as b8://ni-death-col-batty/ we identify them as b8://ni-death-col-batty/paul-hannigan, then Paul can share the cost of setting up ni-death-col-batty with everyone else in that namespace.

If the hashed data is a public key, the retrieve-by-content-hash storage network can be supplemented by a retrieve-by-public-key-hash storage network, which distributes

streams of blobs that have been signed by a given private key. In this way, such stable, unchanging keyphrases can securely identify a time-varying data resource.

To some extent this reintroduces the potential problem that decentralized naming systems are supposed to guard against: participants in ni-death-col-batty may disagree about which "Paul Hannigan" should get the right to determine the binding of b8://ni-death-col-batty/paul-hannigan. Wherever there is community property, there is conflict. But it's not as bad as in, for example, the DNS, where if ICANN is persecuting Paul, no domain name for him will remain stable for long.

Hashcash-like leading-zeroes phrase stretching

I think there *is* a way to get the original publisher to shoulder a disproportionate share of the security burden, with an approach similar to Hashcash or Bitcoin's proof-of-work function. Suppose that, to *create* a valid phrase, you have to do not *one* PBKDF verification, but a probabilistically large number of them, say, 8192 — possibly in parallel, so this need not necessarily add latency, just cost. This can be done by only accepting as valid PBKDF outputs that begin with, say, 13 zero bits. As with Hashcash and Bitcoin, you can do this by varying a nonce that you include in the hashed data, thus producing 8192 different PBKDF outputs. If one of them begins with a leader of 13 zero bits, you take its last, say, 48 bits, and encode them as above as a phrase; any with a 1 bit in its first 13 bits are rejected.

The procedure for verifying a file that purportedly matches a keyphrase such as "thorn foyer debut arson" is to hash the file with your fast file-hashing algorithm such as BLAKE3, run your PBKDF such as Argon2 on the hash for 512 GiB-seconds, and then verify that the leading 13 bits are 0 and the trailing 48 bits encode to "thorn foyer debut arson". This costs the verifier only 512 GiB-seconds (0.002¢), but costs the original publisher 4096 GiB-seconds (0.016¢), and increases the cost of finding a second preimage by brute force by that same factor of 8192 (to 2^{61} runs of the PBKDF, 2^{70} GiB-seconds, 2^{100} byte-seconds, which works out to US$47 trillion), without lengthening the human-memorable phrase.

That's probably far too easy, though. Maybe a more reasonable phrase-stretching difficulty factor would be a week on this old 16-GiB laptop: 2^{23} GiB seconds = 2^{53} byte-seconds (33¢). We could reduce the verification effort to, say, 8 GiB seconds (0.32 microdollars), so we'd need 2^{20} tries to produce a valid phrase: 20 leading zeroes in the PBKDF output. If we keep using 48-bit phrases like "spade will of force", then a brute-force second-preimage attack would take on average 2^{67} PBKDF invocations, each costing 8 GiB seconds, which is also 2^{100} byte-seconds as in the example above.

A larger hashcash-like difficulty factor might be desirable, both to reduce the number of trusted keyphrases floating around out there by incentivizing people to band together a bit more (thus reducing the number of targets for a shotgun bruteforce attack, as explained below), and also to shorten the keyphrases. For example, if we

reduced the keyphrase to 36 bits of entropy ("grips halt seed", "spelt spear pose", "thank graph ready") and increased the difficulty by another 13 bits (8192×, to 2^{36} GiB seconds, 2^{66} byte seconds), then publishing a file would cost US$2700, while a brute-force second-preimage attack would cost 2^{36} times as much: US$190 trillion. US$2700 of computation would be a significant incentive to pay someone else to include your key in their file rather than publish your own file, but it's still low enough to ensure dissidents could establish a memorable identity.

Such a small keyspace of course implies that unless the total number of keyphrases ever created is small compared to $2^{18} = 262144$, there are likely to be collisions; this suggests that perhaps three words is too optimistic, since that's only US$700 million (of legitimate users creating keyphrases) at current prices. If it costs US$2700 to create a four-word keyphrase, then the 2^{24} or so that would make a collision likely would amount to an investment of US$45 billion from the system's legitimate users. The fourth word would raise the cost of a brute-force second-preimage attack to the cost of creating 2^{48} keyphrases: US$760 quadrillion.

(2^{36} GiB seconds would be 2^{32} seconds for this laptop, roughly 136 years. Or 136 such laptops could do it in a year.)

Leading-zeroes phrase stretching *instead of* a PBKDF cost factor

Is there any reason to use *both* a PBKDF *and* a phrase-stretching leading zero count? Well, it's desirable to use a memory-hard PBKDF like scrypt or Argon2 to make the attack memory-hard instead of only CPU-hard, thus preventing vulnerability to future ASICs like Deep Crack.

But, suppose that, while still doing 2^{53} byte-seconds of work to compute the keyphrase, we reduce the PBKDF work factor to, say, 10 milliseconds on this laptop, so that converting a file hash to the PBKDF hash from which the phrase is extracted takes only 10 milliseconds (0.16 GiB-seconds) instead of 500 milliseconds, as in my example above. To compensate, we increase the difficulty from 20 bits to 26 bits: instead of an average of 2^{20} nonces, my laptop will have to try 2^{26} nonces before hitting on one with the requisite pattern of 26 leading zeroes in the output from its PBKDF (Argon2 or whatever).

That makes things easier for users and intermediary nodes, because it's 50 times easier to verify that a file matches a keyphrase; does it make things any easier for attackers?

Well, attackers also have to try 2^{26} nonces for every one that is a valid keyphrase, and, as before, they will have to compute on average 2^{48} such keyphrases before they finally find a second preimage. So the situation for the attacker has not improved at all: they still have to do the same amount of work, it's just that it's divided into 50 times or 64 times as many separate nonces.

Weaknesses

Attacking many hashes at once

If there are many unsalted hashes, the attacker might attack them all simultaneously. For example, if there are 2^{40} public keys in use, each retrievable by its own keyphrase, or 2^{40} documents, and the attacker wins if they succeed in counterfeiting any one of them, their attack speeds up by a factor of 2^{40}. And salting is not really a solution, since the salt would have to be part of the human-memorized authenticator; it's equivalent to just using more hash bits.

I think the best defense for this is to have relatively few human-memorable keyphrases, each identifying a large namespace, as explained above, by increasing the cost to create each keyphrase.

Consider the parameters suggested above: the difficulty is set to US$2700 to create each passphrase, and 48 bits of keyphrase are used, for a cost of US$760 quadrillion to find a particular second preimage. If there are 1000 known keyphrases and an attacker would be content to find any of them, then the cost plummets to US$760 trillion; if a million keyphrases exist, then it totally collapses to only US$760 billion, which is still more than a casual attacker will spend. As noted above, though, soon after that point, at only around 2^{24} keyphrases created, accidental collisions begin to occur, at which point the cost of an intentional shotgun attack is still US$45 billion.

So, with those parameters, there's still a reasonable degree of attack resistance up to the limit of the system being able to function at all.

Collision attacks

The birthday paradox pops up in another context as well.

In some circumstances, second-preimage attacks are not the only attack; birthday-paradox collision attacks have been demonstrated in practice against TLS, for example, where an attacker generates two certificate signing requests: one for their legitimate domain, and one for the domain they want to spoof, which hash to the same hash. The certifying authority signs the legitimate one, but the CA signature enables them to masquerade as the other domain.

Resisting such collision attacks would require twice as many bits, and would thus double the length of the passphrase, so this scheme should probably not be used to defend against a second-preimage attack. Alternatively, you could use keyphrases of 6–12 words instead of 3–6.

Quantum attacks

As I understand it, Grover's algorithm can find a second preimage in the same $O(\sqrt{N})$ time required to find a birthday attack, so if quantum computers scale up, this system would require keyphrases of 6–12 words instead of 3–6.

Although Merkle graphs are in general relatively postquantum-safe if the hashes are big enough, any public-key cryptosystems used with this system would need to be postquantum-safe for the system as a whole to resist quantum computation attacks.

Parameter downgrade attacks

If the difficulty is encoded in a keyphrase — as in my above example with encoding the PBKDF cost parameter in 4 of its bits, or just by virtue of its length — attackers who want to mount collision

attacks could use intentionally weak keyphrases. This is not a concern for second-preimage attacks, but it's a concern anytime producing two colliding files would be a concern.

Possible extensions inspired by time-lock encryption

Someone, maybe Gwern, proposed a one time-lock encryption protocol that I will inaccurately summarize as follows. The speaker computes 1024 random encryption keys for a symmetric cryptosystem and computes 1024 truncated versions by, for example, zeroing the last 20 bits of each key. Then, in parallel, they encrypt each truncated key with the non-truncated version of the previous key (except for the first truncated key, which has no previous key). They concatenate the unencrypted first truncated key with the 1023 encrypted keys, followed by a payload, encrypted with the last key; this is what they publish.

This takes them 1024× the work of encrypting a single key, but since they do it in parallel on 1024 processors, it doesn't add latency.

To decrypt the payload, the recipient must try (on average) 2^{19} possible completions for the first key in order to decrypt the second key, 2^{19} possible completions for the second key in order to decrypt the third key, and so on. We assume they can distinguish a correct decryption from a decryption attempt with the wrong key. In the end they have done 2^{29} decryption attempts and can finally decrypt the payload; if they have 2^{19} processors they can do this in only 1024 times the time to perform a single decryption. Unless they have enough processors to search the whole keyspace, this is necessarily a serial process: they cannot do it faster because they don't know the $N - 20$ bits of the second key until they have finished guessing the first key, etc.

(Of course 1024 can be expanded to a larger number such as 1048576, and you can use a deliberately slow cryptosystem such as a quadrillion rounds of AES-ECB with the same key, or a quadrillion rounds of SHA-256 on a "key" to derive a key that's used for AES-ECB.)

It would be nice to somehow apply this idea in reverse to the hashcash problem: we would like attackers to have to work serially to create a second-preimage key, but listeners and possibly legitimate publishers to somehow be able to do whatever processing they need to do in parallel. If there's a way to do this, it might add another 10–30 effective bits to the keyphrase.

Gwern also mentions Gregory Maxwell's suggestion, possibly anticipated by Ke et al., to do something analogous with ECC cryptography. In many ECC systems, including Curve25519, public keys are in an almost 1-1 correspondence with bitstrings of a certain length, so you can generate a random bitstring that is a valid public key and encrypt something to it. If someone takes the time to bruteforce the private key — not feasible for Curve25519, but of course feasible for elliptic curves over small enough Galois fields — they can then read the message.

Maxwell suggested a Bitcoin-like blockchain in which miners

compete to publish the next private key for a predetermined infinite sequence of random bitstrings taken as public keys; this provides an incentive structure that makes it likely for a predetermined sequence of certain private keys to be published at certain times in the future, and the only way to get all the keys needed to unlock something sooner than the miners do is to have more ECDLP computrons than they do.

This is vaguely similar to the chained-symmetric-keys approach. Could it be applied to this namespace-claiming problem, maybe not using a predetermined infinite sequence?

Generically I think the answer is that the publisher needs to publish some kind of noninteractive, easily verifiable zero-knowledge proof that demonstrates that they know something "about" that keyphrase that is computationally difficult to discover. They start with some secret information, which they never reveal, from which the keyphrase is derived, and which is used to sign the file the keyphrase names. There's an obvious approach that doesn't work: you generate a private key, compute the public key from it, append a signature and the public key to the file, append a nonce, hash it, and if the result isn't a valid passphrase try a new nonce, for 150 CPU-years. This doesn't work because a different public key would work just as well, so it doesn't make a second-preimage attack any harder.

Topics

- Programming (p. 1141) (49 notes)
- Pricing (p. 1147) (35 notes)
- Security (p. 1224) (5 notes)
- Hashing (p. 1293) (3 notes)
- Passwords (p. 1344) (2 notes)
- Namespaces (p. 1351) (2 notes)
- Decentralization (p. 1374) (2 notes)
- Ccn (p. 1380) (2 notes)
- Quantum computing
- Crypto

Wordlists for maximum drama

Kragen Javier Sitaker, 02021-10-08 (updated 02021-12-30)
(16 minutes)

I composed some lists of words for maximum vividness of random imagery in minimal letters, then wrote shell scripts to generate random combinations of them, vaguely like *ars magna Lulli*.

This is potentially useful for encoding passwords or secure content hashes, or naming novel concepts in a terse way that collides with minimal existing terminology; "ehate" or "godfat" may not be a good choice, but "medry" or "linhag" might be okay, especially if you work a digit in there somewhere. "Linhag" is just an encoding of the random number 322442 in base 661, but I'm probably not alone in having an easier time attaching a novel meaning to "linhag" than to "322442". Plus, it's easier to type.

This is a potentially serious pareidolia hazard for anyone who has taken a lot of psychedelics or just has natural psychotic tendencies. Readers in these categories should abstain from reading this in order to avoid delusions that this document contains secret messages from God, extraterrestrials, transdimensional machine elves, or the CIA. Selecting short words uniformly with a PRNG can't compete for pareidolia with ELIZA or GPT-2, but it's definitely good enough to evoke vivid mental imagery.

Try-hat oat-egg lin-hag apt-zen hip-ale sag-pry if-egg way-ho out-neo poo-bud ask-emo tip-poo neo-fry flu-cis bye-kev dog-pug cab-try sun-len cod-foe ex-won wye-dad ash-art or-par lit-oaf min-met lad-tab fop-dub ack-hit elf-run keg-ha few-pun nor-git!

3-letter words

Here's a list of 661 of the 4031 three-letter words that occurred 5 times or more in the British National Corpus. I've tried to remove non-words, abbreviations and acronyms not usually pronounced (including "mrs"), and things that aren't in most native English speakers' vocabularies, as well as the most offensive words, like "fag", "wog", "nig", and "gyp". However, there are still lots of things in there that might offend someone, including but not limited to "god", "ass", "tit", "vag", "goy", "rim", "gag", "fat", "fop", "pus", "gay", and "fap". I've left in many proper nouns (unfortunately not capitalized, due to the source) as well as some morphemes that are especially productive in modern English, like "neo", "eco", and "tri".

This is not suitable as a word-game list for ruling words in or out. The words do not all have distinct pronunciations, so it is not a good alternative to the various "biometric word lists" out there like the S/Key word list.

Selecting one word from this list with uniform probability encodes 9.37 bits of entropy; if we follow it with a space, that's 4 bytes, and thus 2.34 bits per character.

Example three-word sequences:

```
$ for i in $(seq 10); do echo $(shuf -n 3 3-letter-words | awk '{print $2}'); don
e
key hem sun
fly tag tom
wad bay dow
emo ned boo
vie fen kev
tri fen ben
jug mat mid
ben sag ill
pan sim tit
bic doh one
```

Not all of thes are hits, but it's presumably easy to imagine Emo Ned booing, Ben of three swamps, a jug in the middle of a mat, or a pan that simulates a breast.

Pairs of these words usually form a pronounceable nonexistent English word:

```
$ echo $(for i in $(seq 10); do echo $(shuf -n 2 3-letter-words |
    awk '{print $2}') | tr -d ' '; done)
gagkim godfat kinoff yenflu cadice forvex gunark tonroc legten negeco
```

Here's the list:

```
6187927 the
2682878 and
923975 was
836687 for
695595 you
470949 are
462777 not
455906 but
445396 had
433599 his
380284 she
327014 her
262447 all
259443 has
236364 one
236165 can
206627 who
165016 him
163106 its
156114 two
151629 out
143429 did
137814 now
128513 may
125206 new
124108 any
118641 see
101517 how
```

99506 get
96280 way
95006 our
81601 got
71143 own
70169 too
67862 say
60680 day
60612 yes
60358 man
59364 use
55001 put
54610 old
50877 why
48851 off
48189 end
38892 men
38769 set
33752 yet
30379 six
27882 war
27873 car
26734 saw
25918 let
25486 far
25346 law
24712 big
22944 act
22803 job
21803 age
21377 run
20935 try
20419 pay
20274 ten
19810 ago
19401 ask
19161 few
19047 air
18885 god
18820 sir
18483 lot
15873 bed
15695 tax
15087 top
14816 art
14792 cut
14247 bad
13539 per
13290 boy
12985 bit
12857 son
12776 sea
12766 red
12470 nor
12394 low

12329 buy
10950 sat
10817 met
10520 cup
10243 oil
9930 led
9690 lay
9552 eye
9166 arm
9157 win
8859 hot
8766 sun
8742 ran
8710 box
8703 sit
8619 tea
8497 won
8316 sex
8238 add
8165 aid
8063 dog
7939 key
7882 mum
7750 bar
7549 eat
7328 gas
7169 hit
6846 dad
6623 dry
6499 fit
6101 aim
6040 due
5463 die
5343 leg
5299 bus
5166 aye
5063 tom
4984 sky
4909 bag
4762 net
4647 via
4630 row
4610 lie
4567 ooh
4482 odd
4481 bob
4341 sum
4337 joe
4323 jim
4242 van
4009 guy
3933 ref
3862 cat
3839 ice
3821 pub

3783 map
3686 lee
3491 gun
3443 sad
3422 bid
3401 tim
3335 gap
3270 fly
3228 wet
3163 ken
3150 mad
3150 ben
3137 hat
3131 sam
3106 bay
2990 pop
2987 cry
2876 ear
2837 fee
2781 joy
2770 wan
2745 fun
2717 ban
2678 bet
2616 rid
2583 aha
2582 ill
2509 egg
2472 raw
2425 san
2395 tie
2391 sue
2390 fat
2382 vat
2239 mix
2222 tip
2155 era
2120 pen
2020 pot
2011 tin
1985 ray
1950 ann
1921 mud
1895 lad
1884 pat
1864 gay
1806 cap
1784 roy
1767 fox
1765 bye
1760 ira
1759 hey
1756 ate
1720 pan
1717 don

1716 les
1688 kid
1686 liz
1684 tap
1648 fed
1646 fig
1535 pit
1511 kit
1480 rob
1466 lit
1445 yer
1430 fan
1425 cab
1420 jan
1410 eve
1392 inn
1375 max
1369 lip
1369 ham
1355 ali
1352 fix
1320 sin
1317 yep
1314 oak
1305 ace
1303 bow
1298 jet
1269 kim
1264 owe
1257 cow
1255 lap
1204 rod
1188 log
1175 shy
1172 dot
1159 pie
1155 fry
1143 ted
1123 pig
1119 pin
1112 ash
1087 lid
1083 wee
1057 von
1032 hut
1010 lea
1000 dig
989 owl
976 jaw
974 dan
959 bat
950 dos
947 par
942 rat
941 dug

939 non
928 fog
914 ron
896 leo
895 ego
872 fur
869 meg
858 rex
856 hay
847 wit
835 pet
821 gut
809 kin
808 dim
795 del
781 beg
780 jam
771 arc
748 bin
738 nil
734 vic
729 fax
723 ski
721 ram
714 mug
706 hip
699 nod
695 pro
694 hid
687 spy
682 min
681 axe
679 amp
671 icy
666 rug
651 zoo
651 mac
650 thy
649 ink
646 nut
644 dee
640 rub
639 opt
630 vet
628 pad
628 apt
626 lou
614 rev
608 eva
605 dip
601 jar
600 toe
588 web
585 mob
585 ale

578 cox
576 ads
570 flu
569 wax
566 gig
564 toy
553 lab
552 wed
551 jug
551 huh
549 peg
546 dam
545 rim
543 gel
542 pal
528 tag
528 ivy
527 yen
526 sly
523 bee
521 mid
521 mat
521 amy
518 rio
510 spa
510 awe
506 tug
503 gin
502 fen
501 pam
493 rot
492 tee
486 den
482 ton
478 wow
472 mam
466 len
464 ore
459 nan
455 kay
455 con
453 val
451 cot
450 ion
449 rig
449 doo
446 hen
445 nun
443 wry
443 ant
435 loo
434 rag
423 sod
422 hop
419 jew

404 sid
404 cue
402 sic
394 tan
389 sub
386 rue
383 rum
383 jon
382 cod
377 bum
376 dye
375 sec
375 gum
375 cop
374 cam
371 rib
367 jed
367 bud
367 abu
365 zen
365 rip
364 gym
363 tow
359 duo
348 sip
347 rye
343 hal
339 mod
336 bra
332 jay
331 kev
327 hum
325 doe
325 doc
324 mao
323 ono
320 hug
317 zip
315 wig
310 mel
308 hon
306 woo
297 zoe
297 rap
297 fin
295 bug
295 ass
293 nay
293 bog
291 tub
289 cis
283 hem
278 lag
278 foe
278 dup

270 ebb
262 dun
261 mop
260 sow
258 nip
258 bun
257 chi
254 hub
248 rep
247 gem
245 wes
245 lyn
242 pip
242 gee
242 dew
239 tab
233 ned
230 fir
226 nah
224 wye
222 aah
221 din
220 sob
220 lib
219 boo
217 ark
214 vow
214 roe
209 sap
209 ida
205 tit
205 hue
205 elf
205 coy
205 ape
204 ugh
203 yew
202 hun
200 tar
200 sew
193 elm
191 jen
190 pod
190 git
187 cal
186 nos
185 vax
181 alf
179 ada
178 paw
178 cad
176 pre
173 lax
172 oft
172 dis

169 pup
167 ole
167 koi
166 nap
165 pic
164 pep
163 jog
163 emu
160 pea
154 sib
154 lob
153 eel
151 pol
149 tor
149 rem
146 gag
145 lac
144 jot
142 lin
140 eco
139 hob
138 sac
138 pew
135 wad
135 orc
135 aft
132 pee
132 ewe
130 mom
128 vie
127 dab
127 cob
124 yum
124 tot
123 pun
122 hog
121 viv
121 orb
121 dub
120 tho
120 app
118 jab
116 gal
116 cub
115 cum
114 ado
111 yea
109 jig
108 sag
108 imp
108 ere
106 ply
105 yon
104 urn
104 mag

104 hee
103 fro
102 pap
101 soy
100 dow
99 gus
99 bey
98 sis
98 fad
96 rut
95 nag
95 hag
94 pus
94 alt
93 woe
93 doh
90 pow
90 fay
89 dev
86 ops
85 wag
82 pry
82 pox
81 yuk
81 bib
80 tat
80 ode
79 ilk
78 abe
75 hah
71 zia
71 sax
71 hoe
70 lam
70 kip
70 keg
69 lei
66 goo
65 oar
65 mic
64 sci
64 fab
62 mow
61 sim
60 poo
59 arf
58 jib
58 bop
58 alp
57 tic
57 ire
56 nab
56 gob
56 cog
54 cud

51 maw
50 tad
50 jag
49 ohm
48 aba
46 ail
45 boa
45 biz
44 lex
44 fob
43 yow
43 rad
43 mah
42 wok
42 sop
42 roc
42 phi
42 lug
42 ent
41 awl
38 dud
38 caw
38 bro
37 pug
34 zap
33 bio
33 bic
32 fez
31 yaw
31 nib
30 yak
30 jut
29 fop
28 lux
27 bah
26 zed
26 oaf
26 moo
25 tau
24 tri
24 gab
24 fib
23 ska
23 hex
21 zig
21 elk
20 mew
20 het
20 geo
19 fap
18 oat
18 neo
17 nix
17 hew
17 fam

```
17 erg
15 uzi
15 foo
14 yap
13 tox
13 org
13 lye
12 yup
12 vex
12 naw
12 fie
12 fem
12 boi
11 vag
11 lol
11 emo
9 kia
9 bub
8 tux
8 roo
8 hod
8 hep
6 goy
6 gnu
6 ack
5 zit
5 neg
5 glo
```

2-letter words

Of the 577 two-letter words with 5 or more occurrences, I selected 46; a uniform selection encodes 5.52 bits, and so, in three bytes, 1.84 bits per byte. This is slightly lower entropy density than the three-letter list; concatenating them produces a tiny increase in entropy density, and breaks up monospace monotony a lot, but the more useful use for this list is for when you really want a six-byte or five-letter phrase instead of a six-letter or seven-byte phrase, and 14.9 bits of entropy is enough.

Some randomly generated phrases:

```
$ for i in $(seq 10); do

    echo $((shuf -n 1 2-letter-words; shuf -n 1 3-letter-words) | awk '{print $2}↩
↪')
done
me dry
ad rim
eh ate
am val
ha paw
us pew
go for
or war
```

```
me sob
on gay
$ for i in $(seq 10); do
    echo $((shuf -n 2-letter-words; shuf -n 2 3-letter-words) | awk '{print $2}')
done
my cat sum
of phi ilk
we zip pay
ax zoo bun
am dow yer
ax zen few
it met map
if led roy
be aah ray
do why lin
$ for i in $(seq 10); do
    echo $(cat 2-letter-words 3-letter-words | shuf -n 3 | awk '{print $2}')
done
new bow pig
yak jig dos
zed koi joe
oil or sir
imp am far
din wes san
cod ken cal
pi mao mow
ah me hal
fad lyn of
$ for i in $(seq 50); do echo $(cat 2-letter-words 3-letter-words | shuf -n 2
    | awk '{print $2}') | tr -d ' '; done | fmt
wadmad nawfem ayeeel saddog leehop butail deldoc devgit bussag titpry
bitgit skitau bebus putwin owpup hisbiz irakin hadoc tryan yukugh jibaah
leetho jugpea yerhis yakpeg actaid feewed zoomod incox rexfib dohyak
niltry toothe ranrot dandup uziram toymat lowow ahaam beelie dinent
awemet gutbop suedie poomoo dunspa oarlaw dubyea inktom lidrob
$ for i in $(seq 32); do echo $(cat 2-letter-words 3-letter-words | shuf -n 2
    | awk '{print $2}') | tr ' ' '-'; done | fmt
try-hat oat-egg lin-hag apt-zen hip-ale sag-pry if-egg way-ho out-neo
poo-bud ask-emo tip-poo neo-fry flu-cis bye-kev dog-pug cab-try sun-len
cod-foe ex-won wye-dad ash-art or-par lit-oaf min-met lad-tab fop-dub
ack-hit elf-run keg-ha few-pun nor-git
```

This is the list:

```
2941790 of
2544858 to
1849882 in
1089559 it
998867 is
697406 on
681379 he
664780 be
507370 by
478178 at
406705 as
```

```
370855 or
358792 we
344046 an
280701 do
237107 if
212158 so
209943 no
156837 up
152626 my
139028 me
90161 go
78198 us
68437 oh
26872 am
10091 ah
4721 un
4436 co
3618 ha
3492 eh
2860 ok
2005 ad
1700 ye
1490 di
1043 ho
1026 hi
834 ex
615 pi
476 vs
420 id
304 ow
192 ox
173 mu
65 ax
60 om
20 ew
```

This is probably a somewhat suboptimal tradeoff for passwords and hashes

A better tradeoff for the password or content-hash cases is probably to use the 2048 most common English words of five letters or less, which can encode 24 bits of randomness in two words, rather than only 18.7; with my wordlist, for example, 7362508 is "bent ash", 609933 is "why stud", and 11152019 is "ropes cia". A 48-bit password, which is reasonably strong for many purposes if combined with a good PBKDF, might be "karen sped ah cell", which I think is a little more memorable than "gas jug bad bye hee" or "yon sic ow boi mop", which are about the same strength.

Topics

- Programming (p. 1141) (49 notes)

- Security (p. 1224) (5 notes)
- Natural-language processing (p. 1284) (3 notes)
- Randomness (p. 1336) (2 notes)
- Passwords (p. 1344) (2 notes)
- Psychohazards
- Pareidolia

The spark-pen pointing device

Kragen Javier Sitaker, 02021-10-10 (updated 02021-10-12) (1 minute)

I was watching the CHM David Liddle interview, and I was surprised by his description of the "spark pen", a 01970s pointing device using a glass panel and microphones, using the audio transit time of the sound of a spark to measure the position you were pointing at.

You can get several MHz of acoustic bandwidth through a glass panel, and a spark gap has submicrosecond rise time, so you can get submicrosecond positioning precision — maybe a millimeter or so. Two things occurred to me about this:

- A very low-tech way to get the spark is by mechanically opening a switch with the pen point, pushing two pieces of metal apart, while running current through those pieces of metal in series with an inductor, ideally with regulated current. This is maybe easiest if the pen point doing the pushing is a glass rod, but metal would work too.

- If you instead have a spark gap that can be induced to spark frequently by opening a high-voltage MOSFET in parallel with it, you can send a pseudorandom sequence of sparks that sounds like white noise, allowing you to track the pen's position continuously instead of only on command.

Topics

- Pulsed machinery (p. 1167) (12 notes)
- Sparks (p. 1240) (4 notes)
- Input devices (p. 1252) (4 notes)
- Audio (p. 1304) (3 notes)

Beyond overstrike

Kragen Javier Sitaker, 02021-10-10 (updated 02021-12-30)
(13 minutes)

In my childhood, I used a mechanical typewriter with no exclamation mark “!”. It did have a vertical apostrophe “'” which was nearly right, lacking only the dot at the bottom. The solution was to type a “'”, press the backspace key, and then type “.”, thus creating an exclamation mark on the paper. (The vertical nature of the apostrophe allowed it to do double duty as a single-quote as well, much as the letters “l” and “O” did double duty as digits on many typewriters; I think that’s why typewriters with a separate “!” often put it on the “1” key.) If memory serves, it also had an underscore key (shift-6) used in the same way to produce underlined text.

Typewriters for other languages, for which diacritics were important, commonly had “dead keys” for accents. These would place an accent mark on the paper without advancing the carriage, so you can type â simply by typing ^a, without typing a backspace in the middle.

Overstrike and glass ttys

Traditional printers (including the ASR-33) are equipped with this ability to “overstrike”, typically using backspace or a carriage return without an accompanying line feed, which permitted them to do boldface, accented letters, and underlining, without any dedicated electronics or mechanics for those purposes. This is the reason ASCII includes the characters “~”, “^”, and “`”, characters which don’t exist in pre-ASCII English text. APL used this overstrike capability to create an unlimited variety of new operator symbols; for example, by overstriking Δ with |, you could get , the ascending-sort-order operator. Some printers also implemented superscript and subscript with a half-linefeed feature, so you could write x^2 or x_i in a more or less readable way — although these were the same font size due to mechanical limitations.

Glass terminals like the “DECScope” VT-50 did not implement these feature, because taking an arbitrarily long period of time to compose a glyph was not very compatible with scanning out pixels in real time to a CRT electron beam. Instead, “printing” a new character at the location of an existing character simply replaced it. While this removed the ability to overstrike, it made it possible to do real-time screen editing, updating any part of the screen to contain arbitrary new contents.

PLATO

One very notable exception was the terminals developed for the University of Illinois PLATO project; after spending the 01960s using Raytheon storage tubes, starting with PLATO IV, PLATO terminals used Owens-Illinois Digivue gas-plasma screens with (in the 01977 PLATO V incarnation) 8080 processors, and they *did* support overstrike, potentially doing a bitwise OR of the various characters in a character cell, an ability PLATO users used to compose

a variety of creative cartoons. The terminals' protocol was wildly nonstandard, involving 11-bit words from terminal to host and 20-bit words (21 including parity) from host to terminal, and they could handle overstrike because their quarter-megapixel screens were *bistable*, like Tektronix 4014 DVBST. For output they used a non-ASCII-related 6-bit character code with mode shifts, packed three to an instruction word, and for input they used an unrelated 7-bit modeless character code embedded in that 11-bit word.

Probably unsurprisingly, 5 of the built-in 128 characters in the PLATO terminal's ROM character set were accent characters apparently designed to be overstruck with letters: ˜, ¨, ˆ, ´, and ˋ, at positions 033 (octal) to 037 of the M1 font memory. (The character bitmaps can be seen in Figure 2.8.2 on p. 25 of the 01977 PLATO V terminal report.) There were also superscript and subscript control codes to move the baseline up and down by 5 pixels.

David Liddle explained the bistable nature of the panel:

> The plasma panel was invented, or discovered, if you want to think of it that way, that... if you, you probably remember this, though: it, of course, produced those orange dots. But it also had memory, automatically! As you wrote something on that screen, it stayed! You didn't have to refresh it, rescan it, or anything.

He explained that when they started the plasma-panel project in 01968, memory cost 1¢ a bit, so the 262144 bits in a flat-screen PLATO display of the resolution of 512×512 they eventually settled on would have cost US$2621.44, the price of a house; but by the time they finished the project, memory prices had fallen by more than an order of magnitude, greatly decreasing the advantage.

The original 01970 PLATO IV terminal paper explains (pp. 2–3, 7–8/30):

> The terminal should cost less than $5000 ...
>
> Direct viewing storage tubes may be used to overcome the flicker problem but these devices suffer from low brightness and the inability to perform selective erase operations on the displayed data.
>
> The use of a plasma panel, on the other hand, with its inherent memory, eliminates the refresh memory while preserving the selective erase function. Because each point is stored on the panel as it is displayed, the terminal electronics need operate only fast enough to stay ahead of the incoming data. A panel writing rate of 30 KHz is adequate for this [graphical computer-based instruction] application. The digital nature of the plasma panel also eliminates the need for any DA converters.

Brian Dear's "The Friendly Orange Glow" chapter 6 describes the history and principles of operation of these bistable plasma displays in more detail, starting with the January 01963 article "Large Displays: Military Market Now, Civilian Next" in *Electronics* Magazine, incidentally fingering Lear Siegler, later the maker of the ADM-3A, as the first manufacturer of non-bistable plasma displays, and mentions that Doug Engelbart had filed some patents in the 01950s. It credits Gene Slottow at UIUC with the key insight of moving the electrodes to the *outside* of the glass in 01964 or 01965, and said, "By 1967 Alpert and Bitzer had chosen the Owens-Illinois (OI) company, wizards of glassmaking, to be the manufacturer of the displays."

(In Chapter 10 Dear explains that PLATO IV finally established 'formal "prime-time" hours of service' in 01974, despite getting "hundreds of PLATO IV terminals" in 01972, so evidently it took a while to get those plasma terminals rolled out in volume.)

The rest of the terminal seems to have contained about 70 bits of registers and 128 8×16 glyphs' worth of softfont RAM, which are programmed as 1024 16-bit words in "Mode 2" (§2.5) after a LDA ("load address") operation, each 16-bit word being a column of 16 pixels. So the display panel contained 262144/16384 = 16 times as much memory as the entire rest of the terminak. The 64×32 character resolution of the terminal was slightly larger than the 80×25 VT-100.

In 01977, although the PLATO V terminal expanded the Memory Address Register set by a LDA request from 10 to 15 bits, but the only increased the terminal's RAM from from 2048 bytes of RAM to 8192 bytes of RAM, plus 8192 bytes of ROM. So, Liddle's interview aside, I think the memory plasma panel was still a killer advantage even in 01977, though perhaps the cost was set too high for it to go mainstream.

These 8192 bytes of RAM were used in part to expand from 128 softfont glyphs to 384 glyphs, and the base addresses of the softfonts could be changed, allowing you to use *all* of the RAM for softfonts, but you could also load 8080 code into it and run it on the terminal. There seems to have been no way to read pixels from the display, which would have quintupled the total storage available to the program.

Inserting

Anyway, back from the evolutionary dead end of PLATO to the DECScope-style glass ttys and replaceable characters that modern terminal emulators are emulating.

Replacing characters in this way was not ergonomically optimal for human editing of text, since although we do occasionally replace some text with other text consisting of the same number of characters, it's much more common to insert text, delete text, or replace text with text of some other length. So insertion became the standard response to typing a key in text editors, thanks in part to Macintosh, though in some sense Emacs and vi worked this way pre-Macintosh, and I think Smalltalk as well. (See Pipelined piece chain painting (p. 926) for some notes on how random-logic terminal hardware could have been designed to handle this better.)

A torrent of pixels; why text?

Framebuffer-driven displays like the Alto on which Smalltalk was built and like the Macintosh ("bitmap displays", early on) have now become so ubiquitous that other kinds of displays have mostly been forgotten; and GPUs are now fast enough to do many mathematical operations per pixel per frame, so you can recompute all the pixels every frame if you want. So now we can do whatever we want, and the critical question is no longer what the hardware to do but what would be most useful to do.

Given the ease with which we can sling around pixels nowadays, many people are surprised at the continuing primacy of textual programming languages, and even textual formatting languages like Markdown and HTML. My thought is that plain ASCII text has several big advantages; two of them are:

• A tiny gulf of execution: you may not know what your code needs to say, but once you do know what it needs to say, it isn't difficult to figure out how to type it in. If you want the code to say `if x == 3 {` then you press the "i" key, followed by the "f" key, followed by the space bar, etc. You don't have to try to figure out which menu the "if" or the "==" is hidden inside of.
• A tiny gulf of evaluation: similarly, you can tell what your code says, as long as none of the characters are confusingly homoglyphic.

However, there are some drawbacks: ASCII text is not very information-dense, and it's hard to get a lot of preattentively-processable information into it. Consequently, we rely rather heavily on mechanized refactoring tools and indentation.

One approach to solving this is keyboard bucky bits, printing more characters on more sides of your keyboard keys. In the early days of APL this was an easy thing to do, and with the move to mass-market keyboards and ASCII standardization in the 01970s it became impractical. Now, with the profusion of input method editors on cellphones and cheap custom keyboards, it would be feasible again.

APL-style overstrike is another potential way to improve the situation; by overstriking two or more characters into a bindrune like ⍞, , or , you can expand your symbolic vocabulary to some degree without expanding the gulfs of execution and evaluation.

Unicode combining characters are a potential way to get similar benefits, though you must always beware of he who waits behind the wall. They don't extend all that far, they aren't very orthogonal, and they can be hard to read and hard to figure out how to type.

What if we go beyond overstrike? There are more ways to combine and modify existing characters? Superscript and subscript, for example. How about changing colors or fonts? Stacking characters vertically? Squishing or stretching characters vertically or horizontally? You'd have to learn where to find the "stack vertically" or "bold font" command on your keyboard, or the toolbar, but once you did, you could apply it to any character at all.

Topics

• Pricing (p. 1147) (35 notes)
• History (p. 1153) (24 notes)
• Human-computer interaction (p. 1156) (22 notes)
• Graphics (p. 1177) (10 notes)
• Terminals (p. 1202) (6 notes)
• Post-teletype terminal design (p. 1207) (6 notes)
• Memory hardware (p. 1250) (4 notes)
• Illinois PLATO (p. 1280) (3 notes)
• Overstrike (p. 1347) (2 notes)

Pipelined piece chain painting

Kragen Javier Sitaker, 02021-10-10 (updated 02021-12-30)
(23 minutes)

Before people started embedding a computer into every terminal, as was done in the VT-100 (8085), Datapoint 2200 (discrete TTL), and PLATO V (8080) terminals, the traditional way to make a character-cell CRT terminal such as an ADM-3A was to build a hardware character generator. Even some terminals introduced after the Datapoint 2200, and of course many personal computer display driver boards, used this approach to lower costs.

Traditional character generators

We map each character cell to a fixed† position in a random-access screen-buffer memory, then control its memory address bus with a counter during raster scanout. A pixel counter overflows into the column counter, which overflows into a scanline counter, which overflows into a text-line counter. The pixel counter and scanline counters are used to drive some address lines on a font memory (often, tragically, a mask ROM) while the column and text-line counters drive the screen-buffer address lines, the data from which drives the other address lines on the font memory, perhaps delayed by a clock cycle or two of latches.

If the memory used for a screen buffer is fast enough, writes to it can be interleaved with the character generator's reads to avoid the need for dual-ported RAM or the occurrence of CGA-clone snow. One way to make the screen buffer faster is to make it wider, so that it can be read two or four glyphs at a time rather than one. This generally was not done at the time because video signal rates were so much higher than baud rates that snow wasn't a problem. Writes can be delayed until an HBI with a little buffering logic, and I think that's what was generally done.

This approach to driving the screen can produce a high pixel rate even with relatively slow RAM. Typical numbers from this epoch might be 60 Hz for vertical scan, 5 horizontal pixels per glyph, 80 glyphs per line, 8 scanlines per text line, 24 text lines per screen, and about 10% horizontal blanking interval (HBI) and 10% vertical blanking interval (VBI). A glass terminal might talk to the host over an asynchronous serial line at 1200–9600 baud, or rarely 300; at 9600 baud, each character took 1042 μs, 1.04 *million* ns. Multiplying this out, the visible part of the screen contains 192 scan lines, so including the VBI it's about 213, giving a horizontal scan frequency of 12.8 kHz (78 μs per line). In practice this is low enough to produce an annoying whine audible to many people, so often higher frequencies were used; suppose the horizontal scan is instead 20 kHz, including 400 pixels and a 44-pixel HBI. Then our pixel clock is 8.9 MHz; pixels come out 113 ns apart. This is fast enough to require significant attention to signal integrity and path-length issues.

One solution to this problem is to dump each character slice from an N-bit-wide font memory into a shift register like a 74165. The

shift register needs to run at the full dot clock speed, but the font memory can run N times slower; in this case, where N=5, that's 1.78 MHz, 560 ns per access, which is easily attainable with memory chips from the 01970s, and maybe even with faster kinds of core memory. The screen-buffer memory runs at the same speed.

To be concrete about sizes, the font buffer might contain 96 5×8 glyphs, 480 bytes in all, while the screen buffer is just under 2000 bytes. The VT50 or DECScope had only ROM for the font buffer and only half that amount of RAM, 12 lines of 80 characters, 960 bytes, 7-bit IIRC. When even that amount of RAM was too expensive, you were stuck with vector displays or printing terminals.

One drawback of this organization is that the ordinary text-editing operations of inserting or deleting a character require updating potentially a whole line of text in the screen-buffer memory. Doing this locally in the terminal was highly desirable, since retransmitting all those characters from the host to the terminal at 1200 or 2400 baud introduces a delay of a significant fraction of a second, and also likely imposes the cost of interrupt handling and perhaps even context switching on the host; but copying a variable number of glyph indices one item forward or back in the screen-buffer memory, potentially as many as a whole line's worth, is also potentially slow, especially if the screen-buffer memory is multiple glyph-indexes wide as suggested above.

† Scrolling the whole screen is usually necessary, and can be done in a cheaper way than inserting or deleting text, by changing the starting value the text-line counter has at the top of each frame. But inserting or deleting lines is also expensive.

Traversing piece chains in hardware

There's an alternative which I think *might* be simpler than embedding an entire computer into the terminal, which is to traverse a piece chain in hardware. Instead of mapping each memory location in the screen buffer to a fixed location on the screen (or one that is fixed except for scrolling), we build linked lists. Suppose we divide a 2048-byte screen buffer into 256 60-bit words, so addresses are only 8 bits, and store one variable-length *piece* of text in each word. The first byte of the word gives the address of the word holding the text to its right. The next three bits specify how many characters are in this piece, a number from 1 to 7, which is used to initialize a countdown timer in the character generator. Then there are 1 to seven 7-bit glyph indices, which are loaded into 7 shift registers whose low bits drive address lines of the font memory. (There's always at least one glyph per piece to ensure that the pipeline doesn't run dry.)

(There are 128 bytes out of the 2048 which are unaccounted for here, and note that we only support 1792 distinct characters on the screen at once, only 93% of the usual 1920. This is probably still enough for an 80×24 terminal screen almost all of the time, since most lines have some blank space. A blank-space trailer can be drawn with a single-glyph piece that points to itself, shared among all lines.)

The video generation pipeline with this system is slightly longer than that of a traditional character generator. The pixel counter, character-cell counter, scan-line counter, and text-line counter work

as usual, but the character-cell counter is only used to detect the end of the scan line, and the character-cell and scan-line counters are not directly connected to the screen-buffer memory. There's a 60-bit buffer for the next piece; when the countdown of glyphs in the current piece ends, it's used to reload the countdown timer and the glyph-index shift registers, and its next-pointer field is used to fetch the next piece from the screen buffer. Before the HBI ends, the buffer is instead loaded from a piece indexed by the text-line counter.

If every piece on a line is one glyph long, then the character generator will read a word from memory after every glyph, every 560 ns; if the memory is slow, this could reduce the bandwidth available for processing of screen updates. However, serial data transmission is so slow that I think this is unlikely to matter.

Traditional vertical scrolling just involves changing the piece at which each line starts.

An alternative to using linked lists would be to use a piece index vector for each line; you need between 11 and 80 pieces for each line on the screen, so you could use 11 to 80 bytes for piece indices, maybe reasonably in the neighborhood of 20. A disadvantage is that sometimes an insertion would require moving a lot of piece indices over by one to make room for the new piece, and if you were dynamically allocating the piece index vectors in a piece index vector memory, maybe you'd suffer from fragmentation. The potential advantage is that you could share common pieces between lines, providing data compression. Overall I think it's not worth it.

An intermediate approach, sort of a wheel-of-reincarnation thing, would be to have a "piece index stack" (of some limited depth like 4) and a "call bit" in each piece. If the call bit is 0, the behavior is almost the same as described above — the pointed-to piece just replaces the current piece. If the call bit is 1, then the current piece index is incremented and pushed onto the piece index stack; and when the call bit is 0 and the next-pointer is also 0, then the new piece index is popped off the piece index stack. This allows the reuse of common strings for data compression without making insertion difficult.

Serial protocols

It's clear that inserting or deleting a glyph can be done *efficiently* in this representation, but it probably is not *simple*. Deleting a character may involve removing a piece from the piece chain and returning it to a free list; inserting a character may involve allocating a piece from a free list, inserting it into the current piece chain, and moving some of the characters from the current piece into it. While it would be *possible* to do this kind of thing with microcode in the terminal, a much more reasonable thing to do would be to do the logic on a computer, then send to the terminal the commands to make the necessary changes: set the next of (undisplayed) piece 167 to 81 and its glyph count to 7, append the 3 glyphs "ges" to piece 167, set next(27) to 167 and its glyph count to 4.

That's maybe 11 bytes (set7 167 81, append3 167 g e s, set4 27 167) to insert a character, which would be a noticeable 92-ms delay at 1200 baud. You could maybe improve this by, when you might want to

insert into the middle of a piece, pre-copying the glyphs that follow the cursor into a new not-yet-displayed piece, which you point at the display list ahead of time: set7 167 81, append2 167 e s; then actually inserting the glyph is just split4 27 167 g, four bytes that sets both the size and next pointer as well as appending a glyph, or possibly set3 27 167, append1 27 g, six bytes. This is not as fast as the one byte needed for insert mode on a VT100, of course, but it also doesn't need an entire computer embedded in your terminak.

Probably a better option than a byte-oriented update state machine is to simply transmit 68-bit piece updates over the serial interface: an 8-bit piece index plus a 60-bit word to load into it. Under some circumstances this would impose a slight extra performance cost, but it would be less significant at higher baud rates, and it would simplify the terminal hardware significantly. When drawing bulk text this would require 68 bits per 7-glyph piece, compared to the 72 bits required by the byte-oriented approach above (or 90 bits if we're using N81 asynchronous serial with start and stop bits), while the 11-byte sequence above would be two piece updates (17 bytes), and both the 6-byte pre-copying sequence and the 4-byte or 6-byte splitting sequence would be one update (8½ bytes). Note that at 115200 baud, the fastest standard RS-232 baud rate, a 70-bit sequence including a start bit and a stop bit would be 608 μs, roughly 1100 times slower than the fastest frequency at which the character generator might need to read pieces.

This word-transmission approach might actually be faster overall if it permits the use of higher baud rates; the PLATO V terminal in 01977 ran its serial interface at only 1200 baud, perhaps in part so its 5 MHz 8080 could keep up.

The word-transmission approach has the additional merit that it's idempotent, so retransmission is safe, and it's weakly convergent in the sense that if you keep sending messages then probably eventually the state will be correct, instead of persistently diverging. Data transmission errors are likely to induce psychedelic effects with any of these data models, so this sort of convergence is important.

If there's no special support for scrolling, then scrolling a 24-line display would require setting 24 60-bit words, 180 bytes, which would take a janky 188 ms at 9600 baud and a totally unusable 1500 ms at 1200 baud. The easiest approach is to have an 8-bit screen-start register S with the semantic that the first line on the screen starts with piece S, the second line with piece S+1, and so on, and then you can scroll just by creating a blank line and setting that register. If you don't do a modulo of the screen size, you can scroll a window up and down over a larger area instantly, assuming you have enough space for the text, but you'll have to occasionally relocate live data the scrolling is about to steamroll.

Overstrike

If you fetch an average of 20 pieces for each scan line at 20 kHz, that's an average of one every 2500 ns. That's pretty slow, so even with 01970s hardware, you could imagine generating a couple of pixel streams in parallel and ORing them together. That would allow you to generate, for example, accented or struck-through characters. But

the second pixel stream would cost a similar amount of hardware to the driver for the first pixel stream, so this might not really be worthwhile.

Writable and proportional fonts

A softfont, even one with only a few writable positions, extends the graphical capabilities of such a device enormously. A transparent pixel-granularity mouse pointer or other sprite can be emulated with only four writable glyphs and unused glyph indices, and a few more writable glyph positions is sufficient to enable applications like schematic capture, dataflow diagrams, math, and limited foreign language support.

Of course, any kind of softfont capability requires some extra logic to distinguish glyph-update requests from screen-update requests (perhaps a 69th bit), and updating four 5×8 glyphs necessarily involves at least 20 bytes of data if uncompressed; that's only 21 ms at 9600 baud, but a janky 167 ms at 1200 baud.

This setup can even be extended to support proportional fonts in an analogous way: instead of initializing the pixel-within-glyph counter always to 5, you initialize it to a value taken from a font-metrics memory that's indexed by the same glyph index used to index the font-pixels memory. As before, you shift the glyph-index registers when this counter hits 0. This would enable proportional fonts without a framebuffer.

Variable line heights could be done in a similar way but much more easily, since they just involve conditionally resetting the scan-line counter and incrementing the text-line counter during the HBI. But once you have variable line heights you may start wanting the ability to have non-aligned baselines on different parts of the screen. And at that point you've just about moved to a scan-line rasterizer kind of model.

Graphical effects

The VT100 supported inverse video and, I think, some other graphical effects, maybe including double-width, double-height, bright, dim, and blinking characters. I think it had some line-widening logic that extended each bright pixel horizontally by a second pixel in order to reduce the size of the font ROM. Some successor terminals supported smooth scrolling, where scrolling happened a pixel at a time instead of a line at a time. The PLATO V terminal supported "character magnification" to get different font sizes.

In addition to this sort of thing, you could imagine using an LFSR to get "snowy" characters, either as foreground or background, or adjusting the letter spacing or line weight; this could all be done with a dedicated effects bitfield in each piece, avoiding any potential need for blank spaces on the screen. If the frame rate were high enough, or the display hardware possessed of sufficient persistence, you could also use PDM to get various brightnesses, but probably this is usually better done in other ways.

Alternatively, a display list

Compositing from a display list into a single-scan-line "framebuffer" is probably a better way to get things like overstrike, and it would also allow you to do pixel-perfect positioning of text, so variable line height is an easy thing to do. This "line buffer" would cost 50 bytes with the figures described above, but double-buffered, making 100 bytes.

The idea is that, while one part of the hardware is spitting out your pixels at 8.9 MHz from a 50-byte FIFO, another part of the hardware is writing to another 50-byte "back-buffer" register. It's driven by the kind of piece-chain structure described above, but instead of having a rigid grid of lines on the screen, you have a "display list", in which each item is a (y, x, height, piecenum) tuple, which is sorted by y. As we iterate over the scan lines in a frame, the display-list processor maintains two indices into this display list: the earliest item that's still visible, and the earliest item that's not yet visible; and it just draws all the piece chains in between those two indices into the back buffer, one after another, using OR or AND or whatever.

And then, when it's time to move on to the next scan line, the back buffer is loaded into the FIFO, and then the back buffer is cleared. Hopefully the display-list processor finished traversing the display list first, but at any rate it now starts rendering the new line.

This is basically just standard scanline rendering like you might use for a 3-D image rasterizer, but you need a barrel shifter or something in the middle, unless you want to have a second shift register running at 8.9 MHz in the middle of your otherwise relaxed sub-2-MHz system.

Alternatively, if you're using a fixed-width font, and you're willing to snap your horizontal pixel positions to character cells or half character cells or whatever, you can reduce the number of shifts you need between your font memory and your back buffer. If the back buffer is divided into 6-bit chunks, say, and the font is 6 pixels wide, you can AND aligned 6-bit chunks from the font buffer into the back buffer. If you allow 3-pixel shifts, then you need a 3-bit shifter that you can write 6-bit pixel slices into and copy 3-bit-shifted 6-bit pixel slices out of into the back buffer.

This level of control complexity may be high enough that it's justifiable to use a real CPU to draw into your line buffer, even if you have special-purpose hardware driving the actual video signal. At that point it no longer makes sense to think of it as a "terminal"; you want to run your entire program on that CPU as much as possible so that it can be instantly responsive and have as high bandwidth as possible to the display.

This might even be a reasonable way to do PAL video out of a working monochrome GUI on an ATMega328 Arduino: running at 16 MHz I think the TVout library can output pixels at 8 MHz, which leaves enough space for 320,000 pixels in a 25-Hz PAL frame. PAL (except for Brazil's PAL-M) is a 625-line standard with 576 visible lines (92.2% visible) with 51.95 μs of active video per 64 μs line. In theory this means 415.6 horizontal pixels, which is actually enough for 80 columns of 5×8 text (400 pixels). You absolutely can't do it with an in-RAM framebuffer, because the ATMega328 doesn't have enough RAM, but you could maybe do it with a per-line

framebuffer.

On the AVR the situation is a lot worse than in hardware, though: because you don't have the hardware parallelism, you can only render into your line buffer during the horizontal blanking interval. That gives you only about 200 clock cycles, which is not a lot of time, maybe 60–100 instructions.

I suspect it might be possible to steal part of the horizontal visible part of the screen for computation — leave it black (or white) and arrange for a timer interrupt at the right time to generate the back porch and sync pulses. In fact, I think TVout already does this, since it supports any output resolution (as long as you have enough memory) but can only output pixels on integer clocks: every 2, every 3, every 4, or every 5 clocks, but not every 2.5 clocks. Every line is still only 1024 clock cycles but you still ought to be able to do a few hundred instructions that way.

Topics

- Programming (p. 1141) (49 notes)
- History (p. 1153) (24 notes)
- Performance (p. 1155) (22 notes)
- Graphics (p. 1177) (10 notes)
- Terminals (p. 1202) (6 notes)
- Protocols (p. 1206) (6 notes)
- Post-teletype terminal design (p. 1207) (6 notes)
- Tiled graphics (p. 1269) (3 notes)
- Illinois PLATO (p. 1280) (3 notes)
- AVR8 microcontrollers (p. 1387) (2 notes)
- Fonts

An algebra of partial functions for interactively composing programs

Kragen Javier Sitaker, 02021-10-10 (updated 02021-12-30)
(3 minutes)

Consider the statement:

```
y := x * x;
```

One way to think of this is as a partial function from states of the world to states of the world. The prior state (the one in the domain) needs to have some variable `x` defined in it, and the posterior state has that variable `x` and also a variable `y`. We could maybe write that function as {x: x0, ...} -> {x: x0, y: x0², ...}, with the understanding that the two `...` tokens denote the same set of other variable assignments.

This is a partial function in the sense that it isn't defined on states of the world that don't have an `x` in them. We could additionally argue that maybe it requires an `x` for which multiplication to be defined in some way, and describe this as a *type*.

Similarly, we can consider the statement

```
z := y + 1;
```

to mean {y: y0, ...} -> {y: y0, z: y0 + 1, ...}. Instead of being undefined for environments that lack an `x`, this is undefined for environments that lack a `y`.

If we *compose* these two partial functions, the result corresponds to a sequence or progn of two statements:

```
y := x * x;
z := y + 1;
```

which denotes {x: x0, ...} -> {x: x0, y: x0², z: x0² + 1, ...}. The second statement's requirement for `y` in the environment has vanished because the first statement satisfied it directly.

(I've been thinking about a program-calculating environment where you manipulate scraps of program like these, constructing them bottom-up and combining them with elementary operations like sequencing, alternation, and iteration (and their inverses), seeing not only the procedures but the resulting extensional functions visualized as you manipulate them. It would be useful to also have additional non-elementary operations like specialization, loop unrolling, subroutine extraction, and conditional hoisting.)

This sort of inheritance of non-overridden variables is precisely the semantics of indexing I have been thinking about for my "principled APL" project.

Some interesting avenues for further investigation:

• Conditionals (alternation) are straightforward to add to this way of thinking about statements, but what about while-loops? Do they drive us to Dijkstra's weakest-precondition function? Or do they just denote the least fixpoint of a conditional function?
• What do conditionals and while-loops correspond to in APL-land?
• Is subroutine call just a slightly different form of composition in which most of the variable bindings from the called subroutine are discarded?
• We can think of expressions as being functions from these environments to non-environment values. The expression `45`, for example, denotes {…} -> 45, while a more interesting expression like `x * x` denotes the more interesting function {x: x0, …} -> $x0^2$.

Topics

• Programming (p. 1141) (49 notes)
• Composability (p. 1188) (9 notes)
• Programming languages (p. 1192) (8 notes)
• Program calculator (p. 1246) (4 notes)
• Apl (p. 1390) (2 notes)

Beyond op streams

Kragen Javier Sitaker, 02021-10-11 (updated 02021-12-30) (3 minutes)

I was reading about PLATO a lot yesterday (see Beyond overstrike (p. 922)); the PLATO IV terminal designed in 01970 had a relatively simple control system, executing a series of 21-bit words received over a 1200-baud serial line from the computer center. It included line-drawing and character-painting hardware, but because its gas-plasma panel display screen had inherent memory, it didn't need a framebuffer; the screen could be painted as slowly as necessary. And the terminal wasn't programmable at all; it was just a puppet of the supercomputer in the computer center. If it detected a parity error, it would stop processing and signal the error back to the supercomputer, which would then retransmit the operation stream from the point of the error. Keystroke handling was all done on the supercomputer.

Although such terminals were replaced by personal computers in the late 01970s and early 01980s, as I understand it, a similar sort of instruction-stream-executing setup is applied by modern cellphone LCD panels, is used by the "threads" in a GPU "warp" or "wavefront", and was used by the processors in early Connection Machines, which had predication but no separate control. The LCD panels do function as puppets of the CPU (or perhaps GPU), bringing about a predetermined result like the PLATO terminals, but in the other cases there is local data that computes a different result on each processor.

An interesting question to me is this: what's the smallest amount of additional local control you could add to such a system to make it more powerful?

Often such a system needs a current-instruction register, into which it periodically clocks another instruction from the incoming instruction stream. The simplest form of control would be to conditionally not do that, instead remaining with the already-received instruction, executing it a second time. But this needs some kind of termination bit computed somehow, or entering such a repetition mode would be permanent. A repetition-count field of 3–7 bits in the instruction word is one approach, perhaps coupled with some kind of chip-select line that allows you to load a repeated instruction into one such executor, and then while it is running, into another, and so on.

In other cases, the execution unit needs to buffer a sequence of recently received instructions, such as 8 or 16, in a small dual-ported FIFO, which it maintains an execution index into, perhaps lagging behind the incoming instruction stream. A backward-jump instruction can then set up a loop. In such a case, the looping and overwriting need not be conditional, as in the previous case; the looping can simply continue until the loop gets overwritten, though reliably synchronizing this with the loop execution could be tricky.

Topics

- Terminals (p. 1202) (6 notes)
- Post-teletype terminal design (p. 1207) (6 notes)

Inverse perspective

Kragen Javier Sitaker, 02021-10-11 (updated 02021-12-30) (1 minute)

I just noticed that FreeCAD sometimes seems to be using “backwards perspective”: the side of a part that is obscured by being on the opposite side from the camera is projected as *larger* rather than smaller. This is easy enough to achieve mathematically (you just reverse the test for distinguishing visible surfaces from invisible ones) and I understand that there are actually some physical lenses that achieve this in real life as well, sometimes used for machine-vision automted inspection systems.

It occurred to me that this is actually a potentially powerful UI technique for increasing the visibility of parts in CAD, since, with a wide enough viewing angle, you can see the part from nearly all sides at once. FreeCAD in particular is using a rather moderate viewing angle, so the effect is somewhat subtle, and it doesn’t seem to be happening all the time.

Topics

- Human-computer interaction (p. 1156) (22 notes)
- Computer-aided drafting (CAD)

Ranking MOSFETs for, say, rapid localized electrolysis to make optics

Kragen Javier Sitaker, 02021-10-11 (updated 02021-12-30) (8 minutes)

In Dercuano I listed a bunch of "jellybean" FETs in 02017, coming up with this table:

PN	Vds	A	ohms	Qg (nC)	¢	W	type
2N7000	60	.2	1.9	2	36	.4	
2N7002	60	.115	7	2	38		
IRF630	200	9	.4	45	86	75	
IRF9630	200	6.5	.7	29	151	74	P-chan
IRLI630G	200	6.2	.400	40	229	35	
IRLML6344	30	5	.029	6.8	36	1.3	
IRLML6402	20	3.7	.065	12	40	1.3	P-chan
EPC2036	100	1	.065	.910	97		GaN
SI3483CDV	30	8	.034	11.5	89	4.2	P-chan
FQP27P06	60	27	.070	43	134	120	P-chan
NTD4906N	30	54	.0055	24		2.6	obsolete
IRF7307	20	4.3	.140		83		dual (P&N)
BSS138	50	.200	3.5		24		
CPC3703CTR					70		depletion
2N5457	25	.01			230		JFET
2N5458	25	.01			230		JFET
SiS410DN	20	35	.0048	41	94	52	
PSMN4R0-40YS	40	100	.0056	38	88	106	holy shit
IRF540N	100	33	.044	71	145	130	fuck
IRF9540N	100	23	.117	110	189	110	P-chan
IRF9530	100	12	.300	38	138	88	P-chan SyC

One heinous Python expression later and we have them ranked by watts per cent:

```
>>> csv.writer(sys.stdout).writerows(sorted(
    ((float(v[1]) * float(v[2]) / float(v[5]), v[0], v[1], v[2], v[5])
    for v in [line.split('|')[1:] for line in t.strip().split('\n')]
    if v[1].strip() and v[5].strip()), reverse=True))
45.45454545454545, PSMN4R0-40YS ,  40 ,  100 ,  88
22.75862068965517, IRF540N      , 100 ,   33 , 145
20.930232558139537, IRF630       , 200 ,    9 ,  86
12.16931216931217, IRF9540N     , 100 ,   23 , 189
12.08955223880597, FQP27P06     ,  60 ,   27 , 134
8.695652173913043, IRF9530      , 100 ,   12 , 138
8.609271523178808, IRF9630      , 200 ,  6.5 , 151
7.446808510638298, SiS410DN     ,  20 ,   35 ,  94
5.414847161572053, IRLI630G     , 200 ,  6.2 , 229
4.166666666666667, IRLML6344    ,  30 ,    5 ,  36
2.696629213483146, SI3483CDV    ,  30 ,    8 ,  89
1.85, IRLML6402    ,  20 ,  3.7 ,  40
```

```
1.036144578313253, IRF7307       ,  20 ,  4.3 ,  83
1.0309278350515463, EPC2036       , 100 ,    1 ,  97
0.4166666666666667, BSS138        ,  50 , .200 ,  24
0.3333333333333333, 2N7000        ,  60 ,   .2 ,  36
0.18157894736842106, 2N7002        ,  60 , .115 ,  38
0.0010869565217391304, 2N5458        ,  25 ,  .01 , 230
0.0010869565217391304, 2N5457        ,  25 ,  .01 , 230
```

That is, in theory, the PSMN4R0-40YS (unavailable in Argentina) is capable of switching 4000 watts on and off for just under 90¢, so it can control 45 watts per cent, while the IRF540N and IRF630 (available, even listed on MercadoLibre for 80¢ and 65¢) are almost half as good, switching respectively up to 3300 watts or 145¢ (02017 price!) and 1800 watts for 86¢. I probably should have also listed the popular IRFZ44N (55V, 49A, thus 2700W, 76¢ locally) or IRLZ44N (55V, 47A, 87¢, thus 2600W, logic-level threshold).

Six of these monster transistors, plus the appropriate drive circuitry to control them, give you a three-way H-bridge to control a multi-horsepower "brushless" motor. One may be sufficient for a multi-kilowatt switchmode power supply, though maybe running off Argentine 240VAC you'd want two or three in series.

And, for electrolysis they can potentially drive material removal or deposition with jitter under 10 ns and pulse times of 100 ns or so. (This is inferred from the IRF630 datasheet, which has apparently renamed "HexFET" to "STripFET": "typ." 118.5 ns reverse recovery time, 5.6 ns turn-on delay time, 2.6 ns rise time, measured with 4.7 ohm gate resistance and 10 V; oddly they don't state td so it must be something terrible.) For pulses, all of these MOSFETs support even higher powers; the IRF630 is rated for only 9 A continuous, but 36 A pulsed.

Of course you'd have to run the electrolysis through a step-down transformer or SMPS if you wanted to deliver that kind of power in a useful way; 3600 A at 2 V would be a lot more useful than 36 A at 200 V, which would mostly just heat up the water. Such a transformer with 10MHz bandwidth might be hard to find.

The gate charge is "typ." 12 nC, so delivering it in 10 ns would require driving the MOSFET gate with 1200 mA, which is I guess why MOSFET gate driver ICs and pulse transformers are so popular. Getting a 10-ns-rise-time edge through the rest of your circuit is also doable, but nontrivial.

Suppose we *could* deliver 3600 A at 2 V for 100 ns. That's 0.72 millijoules of energy, a perfectly manageable amount for ordinary circuits, and correspondingly 0.36 millicoulombs, or 2.2e15 electrons, or 1.12e15 divalent cations, about 1.87 nanomoles; for copper that works out to be 119 nanograms, and for iron 104 nanograms, assuming perfect Faraday efficiency. That's about a 30-micron-diameter sphere of either of these metals: visible, but barely. (It would punch right through aluminum foil, though.)

(In practice such high current densities would be prevented by the formation of an insulating salt film on the surface. Also 0.72 millijoules in 100 nanograms is 7200 kJ/kg, which is still plenty to vaporize the metal.)

If an electrolytic cathode is flying over a flat metal substrate at 25 m/s, like in a laptop hard disk (but full of water), 100 ns is about 2.5 microns. The 10-ns jitter guessed at above amounts to 250 nm of imprecision. If you were using this to record information, you might encode 4 bits into the delay before each new pulse, with an average of 180 ns per pulse and rest, giving a data rate of 22 megabits per second.

If you wanted to archive a 10 GiB ZIM file of English Wikipedia on nickel foil this way, it might take an hour or so. You might want to reduce the current so you wouldn't be gouging huge 20-micron-deep craters in the surface of the metal that would be hard to tell apart; 130 mA for 100 ns would suffice to give you a hemispherical 1-micron-radius pit. Spacing tracks 2.5 microns apart would give you 400 tracks per millimeter, and 4.5-micron-long pulse-and-rest cycles would give you 889 bits per millimeter, so 2200 bits or 278 bytes per square millimeter, so, all in all, you'd need 39 square meters of nickel.

Consider instead the average material removal rate (or deposition rate), supposing we can step down an average of 9 A at 200 V to 900 A at 2 V; that's about 4.7 millimoles per second, about 300 mg/s of copper or 260 mg/s of iron, supposing divalent ions and 100% Faraday efficiency in each case. That's about 1 kg per hour.

However, 10-ns precision at 300 mg/s means 3-nanogram precision in how much material you remove. If that's spread over a square millimeter at 9 g/cc, that's an etching or electrodeposition precision of 0.3 nanometers, roughly one atom. If we step down to the kind of precision we need for optical systems of about 40 nm, that works out to about a 90 micron by 90 micron area.

So if you were using such a transistor to control the low-precision hogging-out phase of cutting a first-surface mirror, your kg/hour hogging-out process would hit its limit at 40-nm Z precision per 90-micron-square area. Of course, that assumes you're using laser interferometry or something for positional feedback of the electrode.

Then, by turning down the current for a finishing pass, you could overcome that resolution limitation and get the mirror surface more precise, still at the same bandwidth of a few tens of megabits per second.

One of the more interesting devices that can be usefully controlled at 10 MHz or more is a piezoelectric actuator. These don't require a lot of current but they do need relatively high voltages.

Topics

- Electronics (p. 1145) (39 notes)
- Pricing (p. 1147) (35 notes)
- Digital fabrication (p. 1149) (31 notes)
- Electrolysis (p. 1158) (18 notes)
- Pulsed machinery (p. 1167) (12 notes)
- Power supplies (p. 1176) (10 notes)
- Optics (p. 1209) (6 notes)
- Piezoelectrics (p. 1340) (2 notes)

The relation between solar-panel efficiency for air conditioning and insulation thickness

Kragen Javier Sitaker, 02021-10-11 (updated 02021-12-30) (3 minutes)

I was watching Adam Booth's video on the 3 kW generator he runs his travel trailer's air conditioner from, and I realized there was a connection between solar-panel efficiency and insulation thickness.

Straw bales conduct about 0.5 W/m/K of heat; modern insulation materials like polyisocyanurate foam are closer to 0.2, and IIRC firebrick is closer to 0.8. Mainstream solar panels are about 21% efficient, the solar constant is nominally 1000 W/m², and commonplace air-source air conditioners and similar heat pumps have a coefficient of performance (for cooling) of about 2, so they pump out about 2 W of heat for every W of electrical power they consume.

Suppose you have a dwelling pod that's perfectly insulated on every side, except that one side is covered in solar panels and has sunlight falling on it. The sun heats up the solar panels to 50°, even if the outdoor air is a little cooler, but the interior needs to stay below 24° to remain habitable. How thick does the insulation need to be for the air conditioner (hypothetically not blocking any sun) to keep up with the heat leakage through the insulation?

Well, you're getting 1000 W/m² of sunlight and 210 W/m² of electrical power, which provides you with 420 W/m² of cooling. The heat delta from the solar panel to the cool interior is 26° (26 K), so straw insulation has 13 W/m flowing through it, which is to say, 13 W m/m². Multiplying 420 W/m² by 31 mm of straw gives you those 13 W/m. As little as 12 mm of styrofoam might be enough.

If your pod has four times as much area exposed to the hot outdoors as to the hot solar panel, like a travel trailer might, the insulation needs to be four times as thick, like 150 mm of straw or 50 mm of styrofoam. This is still eminently feasible.

Aside from heat leakage by conduction through the walls, every square meter of sunlight you let in through a window adds 1000 W of heating during the day; removing that requires just over 2 m² of solar panels to power the air conditioner.

Normally when you're calculating the energy output of solar panels you need to take the capacity factor into account, which may vary from as little as 10% in polar countries like Germany or the UK to more than 30% in very sunny places (California's about 29%), but, in this case, when the sun isn't shining to power the air conditioner, it also isn't heating up your roof and walls.

Possibly desiccant-based air conditioning systems are a better fit for solar energy, since they can directly use the heat from the sunlight, but nowadays PV panels are amazingly cheap even compared to solar thermal collectors.

Topics

- Energy (p. 1170) (12 notes)
- Solar (p. 1203) (6 notes)
- Life support (p. 1251) (4 notes)
- Air conditioning

An even simpler offline power supply than a capacitive dropper, with a 7¢ BOM

Kragen Javier Sitaker, 02021-10-14 (updated 02021-12-30)
(7 minutes)

Watching BigClive's air "freshener" teardown I was surprised to see an even simpler DC power supply for a microcontroller from the powerline than I'd seen previously, but it turns out that it's only cheaper for microcontrollers dissipating less than about 25 mW.

In the air freshener it consists of two 15K resistors in series, fed from the powerline by a fuse and a diode used as a half-wave rectifier (two diodes in series actually), across a 5V zener and a 100 microfarad 16-volt electrolytic storage cap, bypassed with a ceramic capacitor for high frequencies. There is no galvanic isolation.

I think this is even cheaper and simpler than a capacitor dropper: fuse (in the air-freshener case, shared with the line-voltage heating elements), resistor, diode, zener, and the two caps, six components in all, and you have your 5 volts out. It's extremely inefficient (maybe 2% efficient), but that's nearly irrelevant at the power levels it's cheaper for; it isn't very important if it's wasting 0.1–1 W.

You could cut it to five components if you used a fusible resistor, of which Digi-Key has 140 in stock, like the US$0.05 Vishay NFR25H0001001JR500, a half-watt "flameproof" 1-kilohm jobbie. But how much resistance do we need, and how much power does the resistor need to dissipate?

Suppose the microcontroller draws no more than 10 mA (50 mW), and you're designing for a 240VAC environment. Your rectum-fried DC will peak at 340V, but what's more important here is probably the RMS voltage, which is actually only 120V with half-wave rectification, and the mean voltage, since we want the mean charging current to be 10 mA. The mean of an ideally half-wave rectified signal is about 0.318 of its peak, about 108 V in this case, so we'd need no more than 10.8 kilohms of series charging resistance; lower resistance would be fine but would produce more waste heat, so 4.7k might be better. With 4.7k you get 23 mA average current, 26 mA RMS current, and thus 3.2 watts of power burned in the resistor. That's a curtain-burner!

So you need maybe a 5-watt resistor. These are off-the-shelf parts like the Vishay AC05000004701JAC00, but they're quite a bit more costly; that one costs 71¢, while the 45¢ TE RR03J5K6TB would almost work at 5.6kilohm and a 3-watt power rating. And, perhaps unsurprisingly, none of the resistors Digi-Key lists in that power range claim to be "fusible".

Evidently this simplification is only economical for microcontrollers using significantly less power than that, because using a capacitor to drop that voltage instead would be cheaper. The air-freshener circuit being dissected used 30 kilohms instead of 4.7 or

10, so evidently its microcontroller needs 3 mA or less. Accordingly the resistor's power is only half a watt, and that's distributed over two resistors, which I think are sized for half a watt each, and are located a substantial distance apart on the board, perhaps with the objective of avoiding a concentration of heat.

The voltage across the dropper resistor is effectively the whole half-wave rectified power supply voltage, so the power it dissipates is linearly proportional to the current: 3.2 W at 23 mA (4.7 kilohms), but 0.32 W at 2.3 mA (47 kilohms), and 0.03 W at 0.23 mA (470 kilohms). There's actually a lot you can do even at 0.23 mA.

At 10 mA the 100 microfarad capacitor would also be too small. Because of the half-wave rectification, there's a dead time of just over 10 ms when there's no current charging up the cap, and in that time, it would drop from 4.9 V to 3.9 V. So 100 microfarads is adequate for maybe a 4 mA sustained current draw. But at a lower power level, like 0.23 mA, 10 microfarads is likely enough.

So here's a parts list for a 5V 0.23 mA power supply:

- Dropper resistor: 470 kilohms, >0.03 W: Yageo CFR-12JR-52-470K (0.97¢, 170 mW) or TE CRGCQ0402F470K (0.28¢, 63 mW, SMD 0402, would need potting for creepage because max working voltage is nominally only 50V because of the 0402 package).
- Diode: a 1N4004 or 1N4005 from, say, Micro Commercial Co. (2.2¢, 400V, 1A).
- Zener: something like the Nexperia BZX84-C5V1-235 (1.96¢, 5.1V, 250mW).
- Electrolytic: something like the Würth 860020372001 (5.4¢, 16V, 10 microfarads ±20%, 35 mA max ripple, 5mm diameter, 12.5mm long)
- Ceramic cap: maybe a Samsung CL05A104KA5NNNC (0.22¢, 25V, 100 nF, X5R SMD 0402)

That gives us a total bill of materials of 10.75¢, almost exactly half being the electrolytic. At such low currents, it might be feasible to replace both the electrolytic and the ceramic with a ceramic like the 0805 Samsung CL21A106MQFNNNE, which is 10 microfarads, rated for 6.3 volts, and costs 1.9¢, which would drop the BOM cost to 7¢.

The fuse is essential for safety in case the rest of the apparatus fails short, but it's potentially just a piece of wire that's free to melt without setting anything on fire. The cheapest off-the-shelf fuse is something like the Eaton C310T-SC-4-R-TR1, which costs 14¢, twice the cost of the whole power supply. PowerStream's fuse wire chart suggest that 40-gauge copper wire (79 microns diameter) should fuse at 1.8 amps, and anything 28-gauge or smaller (320 microns diameter) should fuse below 15 amps, which is low enough to keep a house circuit breaker from blowing.

Using 58 megasiemens per meter as copper's conductivity, 79-micron diameter wire (0.0049 square mm) gives you 3.5 ohms per meter. At 1.8 amps, that's 11.4 W/m, or 11.4 mW/mm, and so at 0.0049 mm^3/mm we have 2.3 W/mm^3. Copper is about 9 mg/mm^3 so that's about 2300 W/mg, which does seem like the kind of power

that tends to melt metal.

Because we have about a factor of 8000 between the fusing current and the normal working current, it should be easy to provide enough insulation to permit fusing without causing the fuse wire to go into thermal runaway under normal loads.

A few millimeters of such thin wire (far enough for safe creepage allowances at 240VAC RMS) in an environment that won't catch on fire if the wire melts would be a perfectly adequate fuse.

Topics

- Electronics (p. 1145) (39 notes)
- Pricing (p. 1147) (35 notes)
- Power supplies (p. 1176) (10 notes)

Trying to quantify relative speeds of different digital fabrication processes with “matter bandwidth”

Kragen Javier Sitaker, 02021-10-15 (updated 02021-12-30)
(5 minutes)

2-D cutting fabrication techniques like laser cutting, plasma torch table cutting, waterjet cutting, and CNC milling are often immensely faster than 3-D printing. But they typically require more assembly steps and more difficult design.

Cutting sheet metal in particular is promising because you can bend it after cutting, both hardening it and getting a 3-D shape. Given some sort of press, you can bend it by pressing it between dies cut from the same sheet metal, either many of them stacked up or a smaller number in a crisscross pattern (ideally not quite at 90°). With something like a beading roller, you can use a fairly small die to make a fairly large part. Metal has a lot of other advantages.

Another potentially interesting process for getting a solid 3-D surface from a 2-D contour is to roll up or accordion-fold a strip, using some minimal number of alignment slots to get the successive layers properly aligned.

Laminated sheet metal is actually better than solid metal for electromagnetic purposes, although random mild steel will perform an order of magnitude worse than genuine electrical steel. Permanent magnets are not needed for variable reluctance motors.

Matter bandwidth

Ultimately I think the figure of merit that matters most for digital fabrication processes is the "matter bandwidth", which (being rusty on my Shannon) I roughly define as the number of bits exceeding the noise floor you can impress into a physical object per second. If you can produce a surface with 100-micron precision, then the height of a point on that surface anywhere in a 3.2-mm range counts as 5 bits. But if you have 10-micron precision, that's a little over 8 bits. The reason is that if you can make things more precise, you can make them smaller, while if you suffer less precision, they have to be bigger.

And smaller is actually faster. After having a system that works at all, speed is my most important goal: being able to iterate quickly will enable me to overcome almost any obstacle at all, from fragility (because I can make a stock of spare parts in time) to debugging (because I can try many things) to political opposition (because it arises too slowly to be relevant). This is an enormous reversal from coal-age industrial processes in which mass production and mass processing was of paramount importance.

Very roughly, I think that for a production time of one month (2.6

megaseconds) and a production complexity of a million "voxels", one voxel per 2.6 seconds is adequate. But an order of magnitude better than that would get us out of marginal territory. Given that existing machinery (e.g., laser and inkjet printers) is seven orders of magnitude faster than this, and even RepRap FDM printers are about 10 voxels per second, it seems likely to be achievable, but of course that's drawing on billions of dollars of industrial infrastructure in the form of semiconductor fabs.

(Actually, if we figure 90 mm/s is a normal print speed, 500 microns is a normal trace width, and the resolution is 100 microns, that's 4500 voxels per second.)

Looking at the Maker's Muse product shill infomercial for the US$2900 Phrozen Sonic Mega 8K LCD stereolithography printer, I see it prints layers of 7680×4320 43-micron pixels. The salesman in the video says it he set the first (50-micron) layer to cure for 50 seconds to get it to work reliably in his cold winter, but that this is a really long cure time. I think 20 seconds is a more normal layer time for these UV-cured resin printers, but if we suppose it's 30 seconds, that's still 1.1 megavoxels per second. Later on, he showed that he got 10 hours 11 minutes for a build plate full of 35-mm-tall miniatures, which I think works out to 52.4 seconds per 50-micron layer, mostly due to peel time, since the cure time per layer was only 4 seconds. (The company claims 70 mm/hour print speed, which would be either 2.6 seconds per layer or, more likely, much thicker layers.) He compared to the much smaller Photon Mono X with cure times of 1.5–2 seconds and 60 mm/hour maximum print speed with, I assume, a similar layer height.

If we compare to laser printers, they are typically 600 dpi (23.6kp/m or 558Mp/m²), and the A4-sized ones (0.06237 m², thus 34.8 million pixels) typically print between 5 and 22 pages per minute, which works out to 2.9 million to 12.8 million pixels per second. They're not in very precise places, but the imprecision with respect to the other pixels is predictable and consistent. Unfortunately, it's difficult to convert laser-printed pages into almost anything else; if you print on transparency film such as cellulose acetate, you may be able to take a mold of the printout, but it's hard to stack that up.

Topics

- Digital fabrication (p. 1149) (31 notes)
- Frrickin' lasers! (p. 1168) (12 notes)
- 2-D cutting (p. 1201) (7 notes)
- Self replication (p. 1204) (6 notes)
- Independence (p. 1215) (6 notes)

Balanced ropes

Kragen Javier Sitaker, 02021-10-16 (updated 02021-12-30) (7 minutes)

Is there a really simple way to balance ropes?

Basic ropes

I wrote a simple implementation of ropes last night in 45 minutes:

```
#!/usr/bin/python3
from collections import namedtuple
from functools import cached_property  # 3.8 or later

def as_rope(obj):
    return obj if isinstance(obj, Rope) else Leaf(obj)

class Rope:
    def __add__(self, other):
        return Concat(self, as_rope(other))

    def __radd__(self, other):
        return as_rope(other) + self

    def __str__(self):
        return ''.join(self.walk())

class Leaf(Rope, namedtuple("Leaf", ('s',))):
    def __getitem__(self, slice):
        return Leaf(self.s[slice])

    def __len__(self):
        return len(self.s)

    def walk(self):
        yield self.s

class Concat(Rope, namedtuple("Concat", ('a', 'b'))):
    def __len__(self):
        return self.len

    @cached_property
    def len(self):
        return len(self.a) + len(self.b)

    def walk(self):
        yield from self.a.walk()
        yield from self.b.walk()

    def __getitem__(self, sl):
        if sl.start is None:
            sl = slice(0, sl.stop)
```

```
    if sl.stop is None:
        sl = slice(sl.start, len(self))

    if sl.start < 0:
        sl = slice(len(self) + sl.start, sl.stop)
    if sl.stop < 0:
        sl = slice(sl.start, len(self) + sl.stop)

    # Important special case to stop recursion:
    if sl.start == 0 and sl.stop == len(self):
        return self

    a_len = len(self.a)
    if sl.start >= a_len:
        return self.b[sl.start - a_len : sl.stop - a_len]

    if sl.stop <= a_len:
        return self.a[sl]

    # At this point we know we need part of a and part of b.
    # Since slicing a leaf creates a Rope, we can blithely just do this:
    result = self.a[sl.start:] + self.b[:sl.stop - a_len]
    # Avoid making Concat nodes for lots of tiny leaves:
    return Leaf(str(result)) if len(result) < 32 else result
```

Improvements

The main critical flaw here is the problem of unbalance: if you append a byte to something in a loop, you end up with a radically unbalanced tree, so you will run out of recursion space trying to traverse it, and also some operations that ought to be logarithmic-time become linear-time.

It occurred to me that the code would be better if we put most of the hairy logic down in the concatenation code instead of the slicing code. Empty string plus X? X. X plus empty string? Also X. X + Y when the result is short? A new leaf. This would also avoid the construction of suboptimal trees by non-slicing means.

Weight-balanced trees

Can we solve the unbalance problem the same way? In a concatenation node X + Y, if we want to concatenate Z, we can improve balance by choosing whether to parenthesize (X + Y) + Z or X + (Y + Z). It seems that we should perhaps choose whichever split will be more even: the former if len(X) + len(Y) ≤ len(Z), the latter otherwise. It might be better to have a bias toward the former, which doesn't involve visiting the child nodes of X + Y, so maybe the criterion should be something like len(X) + len(Y) ≤ 2×len(Z). And of course we have the symmetrical procedure for concatenating some X onto concatenation node Y + Z. This amounts to balancing a binary tree with single rotations.

(In an OO language like Python, implementing this by overriding concatenation for Concat nodes is appealing, since the children are readily accessible and no explicit check for leaves is needed; but then

you need some kind of double dispatch to handle the case of prepending a byte onto a giant tree.)

Constructing all our concatenation nodes in this way would seem to ensure that the depth of the tree is logarithmic, because descending one level in a tree thus constructed always rules out at least ⅓ of the bytes that were left over, so at most you have to descend log(N)/-log(⅔) levels: at most 12 levels for 128 bytes (assuming at least one leafnode per byte), at most 27 levels for 65536 bytes, at most 55 levels for 4 gibibytes, etc.

But the very simplicity of the solution makes me suspicious of it: if maintaining a balanced binary tree were that simple, surely someone noticed this before 02021? Because I think it's applicable to binary search trees, too, not just ropes. This approach must have some killer disadvantage for people to have invented AVL trees, B-trees, 2-3-4 trees, red-black trees, splay trees, treaps, and so on. What's the catch?

This structure, or a slight variation on it, seems to be known as a weight-balanced tree, and normally it also requires double rotations. (See below about this.)

It seems straightforward that we can assure the weight-balance property whenever we do a concatenation in the way I described above, and that this will ensure that slicing is logarithmic time. (Possibly slicing would be simpler if decomposed into a prefix-removal operation and a suffix-removal operation, or perhaps a single split-at-point operation, but almost certainly slower.) That eliminates the possibility of a fatal flaw in traversal and slicing, leaving only the possibility that concatenation itself according to this algorithm is fatally flawed by taking more than logarithmic time.

But that won't be the case either. When we're appending two trees A + B, we only recurse down the right edge of A and the left edge of B, and we only do an O(1) amount of work at each node, so we also have a logarithmic bound on concatenation.

Splitting middle concatenees

Aha! I think I found the fatal flaw with this simple approach: in (X + Y) + Z, Y might be 1048576 bytes while X and Z are one byte. So moving Y to one side or the other doesn't help; you might have to split it. Of course that presupposes that (X + Y) already violates the balance condition, but if #X = 32, #Y = 64, and #Z = 32, you have a transition from the balanced state into the unbalanced state. So sometimes you need to split Y, which takes logarithmic time rather than constant time. (And this reinforces the suggestion of building slicing out of splitting rather than vice versa.)

Can you use *only* such splitting? That is, if a concatenation would violate the balance condition, can you just split the larger concatenee in half, instead of rotating? I think the answer is yes, but the cost might be O($\lg^2$ N) concatenation, because you could potentially end up splitting a whole bunch of right-edge nodes in half for a single concatenation.

The standard approach here is to use double rotations, where instead of considering three concatenees X + Y + Z you consider four W + X + Y + Z, which can be concatenated as ((W + X) + Y) + Z, (W + X) + (Y + Z), (W + (X + Y)) + Z, W + ((X + Y) + Z),

or W + (X + (Y + Z)). WX+Y+Z+, WX+YZ++, WXY++Z+, WXY+Z++, WXYZ+++.

Topics

- Programming (p. 1141) (49 notes)
- Algorithms (p. 1163) (14 notes)
- Python (p. 1166) (12 notes)
- Ropes (p. 1333) (2 notes)
- Tree rotation

Flexural mounts for self-aligning bushings

Kragen Javier Sitaker, 02021-10-18 (updated 02021-12-30)
(3 minutes)

I was watching a video about the Open Source Ecology large 3-D printer and the problems they're having with bushing misalignment. Basically the problem is that they have this big 3-D-printed block with spaces in it for bronze bushings, so that it will ride smoothly and with low friction on a pair of steel rods, but it doesn't.

The problem is diagnosed to be misalignment: the bushings are 12-micron tolerance, so a 12-micron deviation in the shape of the cavity they fit into is enough to get them out of tolerance, and possibly kick the two bushings for a single rod out of parallel enough that the rod can't slide through the easily.

It occurred to me that this is another case of needing to worry about not only geometry but the derivative of geometry with respect to force, which is to say, compliance. If there's enough space around the ends of the bearings for them to rotate a little bit in their seats to comply with the rod, this wouldn't be a problem, as long as the solid ring around the middle of the bearing can support the necessary load. (The idea of leaving some play to avoid binding is mentioned around minute 39 of the OSE video, but I don't really understand if they're talking about leaving play in the same place I'm talking about leaving play here.)

More generally you can leave space in the "solid" plastic *around* the cavity which allows the cavity as a whole to rotate but not translate, using established flexure designs to provide selective compliance (whether using FACT or another design approach). Even just using a softer plastic would diminish the binding problem, but that might create undesired compliance in other degrees of freedom; spaces for compliance, like a 3-D version of the Snijlab living hinge, can permit large compliances in selected degrees of freedom with minimal compromise on strength and the stiffnesses in other degrees of freedom. Leaving such spaces can be done even with conventional molding and subtractive manufacturing processes, but it's much easier with 3-D printing or digital 2-D cutting processes.

Another aspect of the problem is that they're building a gantry with lots of prismatic joints (built, in turn, out of cylindrical joints), which pose a lot of problems like binding under side loads, and revolute joints would have been a better choice.

Topics

- Mechanical (p. 1159) (17 notes)
- 3-D printing (p. 1160) (17 notes)
- Flexures (p. 1232) (5 notes)

Triggering a spark gap with an exploding wire

Kragen Javier Sitaker, 02021-10-19 (updated 02021-12-30) (1 minute)

Sometimes a spark gap is used as a trigger for an exploding wire, but it occurred to me that maybe you can do it the other way around: use an exploding wire to trigger a spark gap.

If you mechanically put a thin wire *partway* across a near-breakdown spark gap, you will get corona discharge around the end of the wire, which will supply ions to seed the arc and tend to start forming a streamer. If that glow discharge turns into an arc, the wire will start to heat up, and if it's thin enough, it will explode into plasma, completing the arc.

If the wire manages to bridge the spark gap completely instead of partly, then it might develop a hot spot where it touches one or the other electrode, which could melt it down to a round knob that's too round to create corona discharge and too close to the electrode for avalanche to kick in (below the Paschen minimum). Possibly a small inductor in series with the wire (but only the wire, not the spark gap itself) would avoid this danger.

Topics

- Electronics (p. 1145) (39 notes)
- Pulsed machinery (p. 1167) (12 notes)
- Bootstrapping (p. 1171) (12 notes)
- Sparks (p. 1240) (4 notes)

Triggering a spark gap with low jitter using ultraviolet LEDs?

Kragen Javier Sitaker, 02021-10-20 (updated 02021-10-23)
(8 minutes)

The Akhilleus-heel of the spark-gap closing switch is its high jitter, due to the stochastic nature of streamer formation and cosmic-ray bombardment in the working gas. By stimulating the emission of photoelectrons from a surface with a low work function, you can lower the breakdown voltage of a spark gap significantly, enough to get it to trigger reliably with low jitter; if you can drop the breakdown voltage by more than about 25%, you should be able to reliably trigger a discharge with little risk of prefires (and, in addition to dropping the breakdown voltage, this photoemission also significantly reduces the variability). But, historically, the only reliable way to get the short-wavelength light necessary to stimulate such photoelectron emission was either an incandescent object or another electrical discharge, leading to a chicken-or-egg problem.

LEDs in the 365–400-nm range are now widely available, including US$10 high-power 3-watt jobbies, and for US$4.56 Digi-Key will sell you a 278nm 1-milliwatt Everlight ELUC3535NUB-P7085Q05075020-S21Q UVC LED designed for UV sterilization. LEDs normally have no jitter and can have rise times measured in nanoseconds, though the larger ones have enough junction capacitance to slow this down, and the time lag of photoemission is typically also around a nanosecond.

To reduce the threshold frequency of light necessary to provoke photoemission from the spark-gap cathode, we can coat it with a low-work-function surface; I think barium oxide is the traditional choice for this in vacuum tubes. In a spark gap you would need to ensure that the air within was dry enough not to allow the barium oxide to become wet. Zinc might be a friendlier metal, but it still must be protected from oxidation — or nitridation! This suggests you might need a controlled atmosphere in the spark gap.

I think $h\nu$ is the energy of a photon, where h is Planck's constant 6.63×10^{-34} J/Hz, and ν is the frequency; and the threshold frequency is that at which this energy is the work function of the surface. This gives 3.1 eV at 400 nm, 3.4 eV at 365 nm, and 4.5 eV at 278 nm.

Barium metal's work function varies from 2.52–2.70 electron volts on different crystal faces, putting it within the grasp of even the 400-nm "debatably ultraviolet" LEDs, though its enthusiastic reactivity is problematic; also thus reachable are sodium at 2.36 and lithium at 2.9, cerium at 2.9, and maybe yttrium at 3.1. The more sedate cerium and yttrium are perhaps more promising, though they are pyrophoric and quickly oxidize in air, the oxides are passivating.

The 365-nm LEDs might additionally be able to spall photoelectrons off manganese at 4.1 eV and neodymium at 3.2, and the 278-nm ones could bring within reach zinc at 3.63–4.9,

lanthanum at 3.5, molybdenum at 4.36–4.95, and even tin at 4.42 and lead at 4.25. Unfortunately all the metals whose oxides are less stable in air than the metals themselves (gold and some of the platinum group) have work functions that are still out of reach.

Barium oxide formed in a certain way on a silver substrate has a work function around 3.2 eV, and the barium peroxide (which BaO tends to turn into at room temperature, given the chance) is up around 3.6. On tungsten, barium oxide mixed with oxides of strontium and calcium lowers the work function below 2 eV, and baria alone is calculated to be 2.7 eV. Magnesia, much more chemically stable than baria, has apparently also been used to good effect; although by itself its work function is 4.22–5.07 eV, a thin film of it on a metal surface apparently reduces the work function? I don't know.

So, this suggests a setup with a hermetically sealed gas-filled spark gap where the anode has one or more holes in it through which an ultraviolet LED can shine onto the cathode; the cathode has a partial coating of one of the above systems, such as a thin film of barium oxide on top of a coating of titanium, or a coating of cerium, a coating of lead, a coating of zinc, a coating of lead-tin solder, or a coating of tin. When the spark gap is held a little below its breakdown voltage, a pulse of current through the LED can initiate abundant photoemission into the interelectrode gap, lowering the breakdown voltage enough that the spark gap triggers without any voltage change, and with a jitter measured in nanoseconds.

The advantage of having many holes is that a larger area of anode is exposed to the photoelectron-enriched region of the gas, potentially permitting higher current. The advantage of not having many holes is that lower arc inductance can be achieved by having many parallel arcs around the edges of a single round hole, and it's easier to fabricate.

All of these same techniques can also be applied to a pseudospark switch, and of course any other low-work-function material can also be used. Pseudospark switches normally have jitter down in the tens of ns.

Although UV irradiation drops the breakdown voltage, I'm not sure it drops it *below* the lowest safe non-UV-irradiated breakdown voltage. If that is the case, this approach will always have a significant chance of prefires. (I don't know why free electrons in the gap don't drop the lowest breakdown voltage, but apparently free ions in the gap from a previous firing do.)

A hybrid approach, however, should work extremely well: use an UV LED to illuminate a conventional electrically-triggered spark gap, using a third triggering electrode (whether insulated with quartz or not). This should give jitter that's as low as could be hoped for with the LED, while eliminating the risk of prefire. You still need a low-work-function electrode surface to keep the work function low enough to overcome with a mere LED. (Hofstra sells a UV-illuminated spark gap using this principle, but I think it uses conventional a mercury-discharge-lamp UV source rather than LED illumination; it appears to be hand-sized.)

There are reports that in the twilight zone below the Paschen

minimum, where pseudospark switches operate, electron injection *is* adequate to reliably trigger a discharge, and “UV flash” is an existing triggering approach. Perhaps electron injection via UV-LED-induced photoelectrons would be sufficient. Normally pseudospark switches operate at neon-sign-style vacuums (10–50 Pa) in order to get past the Paschen minimum, but you could instead simply make them very small (micron-sized gaps), reducing the distance factor rather than the pressure factor.

To avoid arcing *to* the low-work-function “seed” surface — for example, if it’s delicate, has annoyingly high resistivity, or would contaminate the dielectric gas undesirably — it can be placed closer to the anode than the main cathode is, connected to the rest of the cathode with a heavy high-value resistor. As long as only the photocurrent is flowing in the gap, the voltage across the high-value resistor is effectively zero, but once the spark initiates to the seed, the seed quickly reaches the potential of the anode, so the spark will rapidly propagate to the rest of the cathode through the gas, since now the entire gap’s breakdown voltage is across the much shorter distance between the two parts of the cathode, and across the resistor.

Topics

- Contrivances (p. 1143) (45 notes)
- Electronics (p. 1145) (39 notes)
- Pricing (p. 1147) (35 notes)
- Pulsed machinery (p. 1167) (12 notes)
- Sparks (p. 1240) (4 notes)
- LEDs (p. 1286) (3 notes)
- Photoemission (p. 1341) (2 notes)

Binomial coefficients and the dimensionality of spaces of polynomials

Kragen Javier Sitaker, 02021-10-20 (updated 02021-12-30)
(4 minutes)

The number of coefficients of a general polynomial of a given degree in a given number of independent variables is a binomial coefficient, nCm. This is surely well known, but it was surprising to me.

The general quadratic polynomial in two variables is $ax^2 + bxy + cy^2 + dx + ey + f$; it has six coefficients. Each of its terms has degree 2 *or less*.

(By introducing a third variable z=1 we can make them each have degree 2 *exactly*: $ax^2 + bxy + cy^2 + dxz + eyz + fz^2$. So 6, the number of coefficients, is also the number of ways to get 2 by adding up three natural numbers (including 0): 2+0+0, 1+1+0, 0+2+0, 1+0+1, 0+1+1, 0+0+2.)

In one variable, the number is 3: $ax^2 + bx + c$. In general, in one variable, the number of coefficients for degree d is d+1.

In no variables or in degree 0 for any number of variables, the number is 1: k.

One way to factor $ax^2 + bxy + cy^2 + dx + ey + f$ is as a quadratic in x, $ax^2 + dx + f$, plus y times a linear in x, $y(bx + e)$, plus y^2 times a constant in x, cy^2. In matrix form:

```
[ f   e  c ] [ y⁰ ]
[ dx bx  0 ] [ y¹ ]
[ ax² 0  0 ] [ y² ]
```

So it seems like the number of coefficients in a two-variable polynomial will be a triangular number; to extend this to cubics, for example, we'd add a new leftmost column that's a general cubic in x, some zeroes to fill out the bottom row, and y^3 to the bottom of the vector.

This is a general property: the general degree-d polynomial in n variables is a sum of the general polynomials of all degrees up to and including d in n-1 variables, each multiplied by the appropriate power of the newly introduced variable to bring it up to the correct degree.

If we instead try the same trick to go to a quadratic in three independent variables, introducing, say, w, we can take this general two-variable quadratic and multiply it by w^0, take a general two-variable linear polynomial (3 coefficients) and multiply it by w^1, take a general two-variable constant polynomial (1 coefficient) and multiply it by w^2. So we have 1 + 3 + 6 = 10, but not because it's the fourth triangular number; rather, because it's the third tetrahedral number.

Without formally proving it, it seems like this is a matter for

Pascal’s triangle of binomial coefficients, $(a+b)!/(a!\ b!)$:

1	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0
1	2	1	0	0	0	0	0	0	0	0
1	3	3	1	0	0	0	0	0	0	0
1	4	6	4	1	0	0	0	0	0	0
1	5	10	10	5	1	0	0	0	0	0
1	6	15	20	15	6	1	0	0	0	0
1	7	21	35	35	21	7	1	0	0	0
1	8	28	56	70	56	28	8	1	0	0
1	9	36	84	126	126	84	36	9	1	0
1	10	45	120	210	252	210	120	45	10	1

The number of coefficients for polynomials in zero variables are in the first column; for polynomials in one variable in the second column; for polynomials in two variables in the third column; and so on.

The fourth column says that in three variables, a degree-0 polynomial has one coefficient; a degree-1 polynomial, 4; a degree-two polynomial, as explained above, 10; and a degree-three polynomial, 20.

This also means that the number of coefficients in a polynomial in three variables grows as $O(d^3)$; specifically, it's $(d+1)(d+2)(d+3)/6$.

Topics

• Math (p. 1173) (11 notes)

Finite element analysis with sparse approximations

Kragen Javier Sitaker, 02021-10-20 (updated 02021-12-30)
(2 minutes)

Normally finite elements are constant, linear, quadratic, or cubic, so you have a relatively limited number of trial functions per element, and a relatively large number of elements. In a three-dimensional element, an arbitrary scalar cubic polynomial has 20 degrees of freedom (coefficients). In three dimensions, the number of coefficients ($(d+1)(d+2)(d+3)/6$, see Binomial coefficients and the dimensionality of spaces of polynomials (p. 957)) grows as the cube of the degree, which gets annoying quickly, so normally instead of adding more coefficients you just use smaller elements. (Also, as I understand it, normally you impose continuity conditions which reduce this dimensionality greatly, as with cubic splines in one dimension.)

(There are lots of sets of 20 polynomials you can use as a basis for this 20-dimensional space, even a convenient orthogonal basis.)

But suppose that, instead of using low-degree piecewise polynomials, we use a smaller number of larger elements, each with a much-higher-dimensionality function space, and then try to keep things computationally tractable by seeking a *sparse* approximation in that larger space, or at least, a sparse state evolution rule over time?

In a sense this is the farthest thing from a new idea; it's how Fourier solved the heat equation and Wiener solved everything, by lumping the entire system into a single element and picking basis functions that were eigenfunctions of temporal evolution, so the system evolves independently in each of these "vibrational modes", making computation enormously easier.

Fourier space is of course a useful space to seek a sparse approximation of lots of things, but there are numerous other alternatives, including chirplets and all kinds of wavelets.

Topics

- Math (p. 1173) (11 notes)
- Numerical modeling (p. 1229) (5 notes)
- Sparsity
- Finite-element methods (FEM)

Implementation and applications of low-voltage Marx generators with solid-state avalanche breakdown?

Kragen Javier Sitaker, 02021-10-20 (updated 02021-12-31) (39 minutes)

It occurred to me that a back-biased red or green LED or base-emitter junction might make an interesting substitute for a spark gap in a tiny Marx generator. These often break down in avalanche mode with minimal or no damage to the semiconductor, often at voltages under 10 volts. So you could easily produce a pulse train at 100 volts or more, perhaps from a 5-volt source using only tiny surface-mount components. This led me to dig a bit into the existing literature on similar devices.

Possible characteristics of such semiconductor-switched Marx generators

The average achievable power of the device probably will not be large, since especially base-emitter junctions are not optimized for power dissipation, and, according to folklore, high-power LEDs generally do not tolerate avalanche breakdown well. Moreover, the properties the device relies on here (avalanche energy tolerance, reverse breakdown voltage) are not among the properties the manufacturer normally specifies, except I guess on JFETs. But JFETs are tiny high-precision devices, so JFET gate junctions are probably even worse for this application.

I suspect that, at least with silicon, you won't get a good avalanche effect until 7 volts or more; below that, the less abrupt Zener effect will dominate. Purpose-built "zener" avalanche diodes might also work, but since they're sold as voltage references, they might be designed to spike as little as possible — the spikes we're after here are troublesome noise in voltage-reference applications. At least you'd have a well-characterized and optimized power rating. And of course diacs would definitely work and have been used in production Marx generators in the past. (Can you get MOS latchup out of power MOSFETs? Probably not.)

Looking at Can you get JLCPCB to fabricate a CPU for you affordably from "basic" parts? (p. 347), I see that 5% 1k resistors occupying half an 0402-equivalent cost 0.385¢ soldered (and a few other values are available); discrete 1% 0402s are 0.305¢ in a wide variety of sizes; 0402 MLCCs are 0.4¢, while 1206 MLCCs are 1.3¢; NPN S9013 transistors are 1.58¢ and MMBT3904 is 1.32¢; red 0805 LEDs are 1.54¢; zeners only come in 5.6V and 3.3V and cost 1.35¢; 5.8V TVSs cost 2.91¢. Each Marx stage requires an avalanche element, a cap, and two resistors, so maybe an MMBT3904 (3 mm × 3 mm?), an 0402 cap (1 mm × ½ mm), and two 0402 resistors, for a total of 2.33¢ and 10½ mm², probably 25 mm² in practice.

A particularly interesting question is how fast the device completes its avalanche discharge and recovers (through recombination of the freed-up charge carriers). It wouldn't surprise me to find a transistor was capable of producing higher frequencies in this avalanche-diode mode than when being used as a real transistor. But are we talking about potential pulse repetition rates of 10 kHz, 100 kHz, 1 MHz, 10 MHz, 100 MHz, 1 GHz, 10 GHz? How high do the harmonics go? I'm guessing that they're much faster than a corona-stabilized spark gap (≈20 kHz repetition rate) but I don't know how much. Even high-voltage SCRs typically manage 10 kHz.

Possible applications

• Pulsed power for operating LEDs at higher power without time for thermal runaway through current hogging. Probably very inefficient.

• Neuron-style spike-train computation, in which a stimulus pulse can produce a much larger (synchronized) response pulse if the "neuron" is currently charged up enough — so you get implicit addition of spikes that arrive nearby in time;
• Phase-locked spike-train computation (a technique published in a blog comment by Pete Castagna in 02016 and probably long before), in which a "clock" spike train defines 2–5 phases at which "slave" oscillators can run; depending on their charging rate, their phase can be advanced by feeding them one or more additional stimulus pulses at the right time. Although the hardware is different, higher-level logic design out of such devices might be similar to the bistable elements described in Snap logic, revisited, and four-phase logic (p. 115) and in my Derctuo note about "majority DRAM logic".
• There's no need for all the slaves to have the same period; this would result in nonstop phase displacement among them. Three slaves of periods 3, 4, and 5 produce a period-60 counter, spiking at the same time once every 60 clock spikes.
• Low-jitter pulse amplification for trigger pulses for larger devices.
• RF signal generation, either for UWB communication and ranging, or to produce a crude high-voltage VCO (CCO?) that can then be filtered to a band of interest. Because the time-domain comb signal from an idealized free-running Marx includes equal amounts of all harmonics (dc, f, 2f, 3f, etc.) the actual frequency of interest may be far above the spike frequency.
• If you drive two divide-by-2 slave generators from the same master clock, then the *difference* between them will only contain the odd harmonics of the slave frequency: f, 3f, 5f, etc., notably excluding dc.

• Pulse density modulation, in which the device is either triggered or not triggered every microsecond or so. This is appealing not only because of power gain (which it might or might not provide) but also because of the potential for driving high-impedance analog loads like piezoelectrics.
• RF demodulation: by keeping the Marx generator charged to the instantaneous voltage of a poorly filtered radio signal, then reliably triggering it with a "sampling-comb" pulse train at some harmonic of the desired frequency (say, 2×, 3×, or 4×), you will selectively amplify the subharmonics of the pulse-train frequency, effectively

multiplying it by a sampling comb in the time domain, and therefore convolving it with a comb in the frequency domain, converting every subharmonic of the sampling frequency to baseband. For this application, the Marx generator itself is a somewhat suboptimal many-pole RC low-pass filter, which should mitigate the aliasing problem. This requires avalanche times an order of magnitude or more above the frequency of interest. Honestly a regular S&H is probably a lot better for this unless you can't get a fast enough transistor.

- Polyphase analog filtering: by running multiple slave oscillators at the same period and different phase shifts, you can generate spike trains to multiple different analog samplers.
- Similarly for CDMA-type time-domain communication applications, where instead of "sampling" at regular intervals, you sample at times pooped out by an LFSR or something. If you have two such sampling devices, each powering an integrator (or one charging an integrating cap and one discharging it), you ought to be able to decode your signal of interest from the difference. This of course requires extreme temporal synchronization, but that might be doable in a simple PLLish way, letting the sequence generator run free at a slightly wrong speed until it detects a signal, and then running two detectors off it at a slight delay to get the phase error to drive the PLL.
- Powering a Cockcroft–Walton generator, for example, to fire a xenon strobe lamp on a camera.
- RF switching: if the capacitors in the Marx are reverse-biased diodes, their capacitance goes down and their impedance goes up as they charge up. Every time you erect the Marx, they will discharge and briefly be low impedance to RF signals, before charging up again. This potentially allows you to sample more than just a point on a signal. It probably requires inductors on the avalanche elements in order to slow the edge and remove interference it would otherwise create in the band of interest, plus of course bias tees to couple the RF signal in and out.
- Generating magnetic pulses for magnetoforming, for example, aluminum foil.

One particularly amusing hackish thought: the avalanche elements can be back-biased diodes, the capacitors can be beefier back-biased diodes, and the charging speed limiters could also maybe be tiny diodes instead of resistors, though that's a trickier proposition; this would enable you to do digital logic entirely out of diodes! If it's possible, and fast, this seems like it would have been a killer advantage in the 01950s and early 01960s, when transistors were expensive and hard to get, and vacuum tubes more so; some Russian electronics were "ferrite/diode" systems in which the diodes took care of the combinational logic and (square-loop?) ferrite transformers handled memory and inversion.

When I was 9, I proposed solid-state switching elements for computation that worked on the neon-lamp-like principle of avalanche breakdown in ionic solids. Those are not practical, because the device would require very high voltages, and its crystal structure would rapidly lose integrity (almost certainly, anyway). But avalanche discharge in solid-state semiconductors is commonplace;

it's the way diacs, triacs, and other thyristors and SCRs work.

Notes on other people's work

Kerry Wong's minimal 2N3904 pulse generator

Kerry Wong built a 2N3904-based pulse generator, running it off 120V (!!) and back-biasing the *collector*—base junction, which he says consistently avalanches around 100V, saying:

> Avalanche transistors can be used to generate fast rise time pulses. Their usage in the hobby world was made popular following an application note ([Linear] AN72 2 [now at Analog[9]]) by Jim Williams and was further publicized via this EEVBlog video. …
>
> R2, C1 along with the NPN transistor form a relaxation oscillator. The capacitor gets charged via R2 and then rapidly discharges when the collector-emitter voltage reaches the avalanche voltage. The discharge current flows through R1 during the avalanche and forms a fast-rise pulse between ground and the emitter. The choice of R2 and C1 is pretty liberal. In general, C1 can range from a few pF's to tens of pF's and R2 can range from 100K to 1M. The larger the value of C1, the wider the avalanche pulses due to increased discharging RC (R1C1) constant. But C1 cannot be too large as the energy released during the short avalanche period could cause the PN junction to fail. The RC constant (R2C1) determines the operation frequency. For the values given [220kΩ and 22pF], the pulsing frequency is at roughly 30 kHz. R1 is chosen to match the characteristic impedance of the load. …
>
> During my build process, I sampled a large batch of 2N3904's, and found that most can avalanche pretty consistently at around 100V. …
>
> …The following picture shows the same pulse observed on a Tektronix 2445 (150MHz bandwidth) with matching input impedance. The measured rise time is around 1.5 ns which corresponds to a bandwidth of approximately 230 Mhz [$0.35/T_r$].

His pulses look like only 50 volts, though, suggesting that they might actually be much faster than 1.5 ns, and being limited by the oscilloscope's 150MHz input bandwidth. Oddly, log(120V/100V)/(220kΩ 22pF) works out to about 16 kHz, not 30 kHz.

Using the huge collector-base junction instead of the teensy emitter-base junction probably means you can handle a lot more power, and it is at least a somewhat controlled process parameter, since people actually do often require that their transistors resist a back bias on the base-collector junction; ST's 2N3904 datasheet specifies a minimum of 60 V for the base-collector reverse breakdown and, surprisingly, provides a minimum value for the base-emitter reverse breakdown voltage as well: 6 V. Interestingly, its delay time and rise time (for normal transistor operation) are specced as 35 ns, with 200 ns for storage time and 50 ns for fall time, and a 270 MHz transition frequency.

This means that the pulse's rise time is more than 20 times faster than the pulse you'd get using the transistor *as* a transistor *switch*, but maybe no faster or even a bit slower than if you were using it in its linear region. But probably the transistor is not the limiting feature here.

He also cites a 01997 paper by Kilpelä and Kostamovaara and a pulse generator project by Andrew Holme.

Holme's 2N3904 pulse generator

Holme used a 2N3904 and an open coax transmission line rather than a 22-pF cap to get a rectangular pulse with about a 400-ps rise time, which he says is limited by his oscilloscope. Astonishingly, he did this with through-hole components.

The coax transmission line suggests how to get arbitrarily high gain from such a circuit, considered as an amplifier: an arbitrarily short input pulse can produce an arbitrarily long output pulse, as long as the current is high enough to maintain the avalanche but not high enough to overheat the transistor. (I think you can do this with a capacitor too — it's just messier.)

Holme mentions that you can trigger the circuit by applying short pulses to the base, which is a thing I hadn't thought of; both Wong and Holme are taking their main signal from the emitter and just tying the base to ground through a big resistor. I suppose that you'd pull the base *negative* to trigger it in that case, thus increasing $|V_{CB}|$ enough to cause an avalanche — just treating the transistor as a pair of back-to-back diodes? (This is wrong; see below.)

Holme also cites Jim Williams's Linear AN72 and AN94. I guess when Analog bought Linear they broke the link, but I found it anyway.

AN94: The Taming of the Slew

Jim Williams's AN94 is about measuring an amplifier slew rate at 2.8 GV/s, for which he had to build a 360ps-rise-time 15–20V pulse generator for this purpose, because his fancy 1-ns rise-time pulse generator was too slow for the amplifier he was measuring, but subnanosecond-rise-time pulse generators cost US$10k–30k. So he used a 2N2501 (or maybe a 2N2369) as an avalanche transistor, biasing its collector to 70 volts above ground. Interestingly, my understanding of triggering with a base pulse is incorrect, at least for this circuit: he uses a *positive-going* trigger pulse into the base of the avalanche transistor to trigger the avalanche, which I'd've thought would be counterproductive. He AC-couples the trigger pulse with a 5pF cap and protects the avalanche transistor's base with a Schottky up from ground.

The 2N2501 looks like a perfectly ordinary (but old) small-signal NPN transistor: 350 MHz, $\beta \approx 50$ (or >3.5 for small signals), 40V minimum $V_{(BR)CBO}$, 1.2 W, 100 mA; the 2N2369 is pretty similar, but maybe 500 MHz and 200 mA. The datasheets show them in a 01960s-style TO-18 metal can rather than a modern TO-220 or similar epoxy package; an advertisement for the 2N2501 appeared in the May 4, 01964 issue of *Electronics*, though with only 20V "BV_{CBO} ", and both transistors appear in the 1965 Motorola Semiconductor Data Manual, with ratings more like the 40V I mentioned earlier. Neither is billed as an avalanche transistor or has a datasheet with avalanche characteristics, and there's nothing to suggest that they can in any way be used to generate 400-picosecond edges.

I wonder if Williams used them even in 02003 instead of more modern parts because the modern parts were "much improved" — in the sense of having an inconveniently higher base-collector breakdown voltage. (Does that also imply a larger junction capacitance?) Williams comments that not every transistor of this model was suitable:

Q5 requires selection for optimal avalanche behavior. Such behavior, while characteristic of the device specified, is not guaranteed by the manufacturer. A sample of 30 2N2501s, spread over a 17-year date code span, yielded 90%. All "good" devices switched in less than 475ps with some below 300ps.[6] In practice, Q5 should be selected for "in-circuit" rise time under 400 picoseconds.

Note 6: 2N2501s are available from Semelab plc. Sales@semelab.co.uk; Tel. 44-0-1455-556565 A more common transistor, the 2N2369, may also be used but switching times are rarely less than 450ps. See also Footnotes 10 and 11.

AN72: A Seven-Nanosecond Comparator

Previously Williams wrote AN72, which also covers the technique, but at less length; most of AN72 is a primer on the basics of working with VHF and faster circuits, or explanations of what you might want to use Linear's new high-speed LT1394 comparator for. But in pp. 32–34 and in appendix B, he gives a couple of simplified versions of the AN94 design using a 2N2369, with a 2pF capacitor instead of a transmission line. Here he also explains the measures necessary to prevent the 250-ps-rise-time avalanche pulse from overwhelming the output of the comparator providing the trigger pulse: 100Ω and six ferrite beads! There must be a reason a diode wasn't enough but I don't know what it is.

This suggests that after writing AN72 he found that the 2N2501 gave better results than the 2N2369, which implies that probably most transistors will be significantly worse.

He describes the transistor a couple of times as "a 40V breakdown device".

The avalanche pulse measures 8V [4.8V where text is duplicated in appendix for slightly different circuit] high with a 1.2ns base. Rise time is 250ps [216ps in appendix], with fall time indicating 200ps [232ps in appendix]. The times are probably slightly faster, as the oscilloscope's 90ps rise time influences the measurement.

Q5 may require selection to get avalanche behavior. Such behavior, while characteristic of the device specified, is not guaranteed by the manufacturer. A sample of 50 Motorola 2N2369s, spread over a 12-year date code span, yielded 82%. All "good" devices switched in less than 600ps [650ps in appendix]. C1 is selected for a 10V amplitude output. Value spread is typically 2pF to 4pF. Ground plane type construction with high speed layout, connection and termination techniques is essential for good results from this circuit.

Note that 8V is a lot less than the 70V or 90V to which the 2pF capacitor is charged via a 1MΩ resistor; he says that 90V gives about a 200kHz free-running frequency.

In AN94 he reported that he was having a hard time getting down to 300 ps, so maybe that 250-ps transistor was just a super good one, or maybe he decided it wasn't really 250 ps.

You might think that it would be hard to get less than 2pF parasitic capacitance between PCB traces and stuff, but in the photo it seems he just constructed that part of the circuit soldered to the backside of a BNC connector.

This circuit also reconfirms that, contrary to my previous expectations, he was triggering the avalanche by forward-biasing the base-emitter junction, just like you normally would to operate a transistor.

For more information about avalanche transistors for pulse generation Williams refers us to these references:

17: Williams, J. "High Speed Amplifier Techniques," Linear Technology Corporation, Application Note 47 (August 1991)

20: Tektronix, Inc., Type 111 Pretrigger Pulse Generator Operating and Service Manual, Tektronix, Inc. (1960) (Williams says his method "borrows heavily from" this device.)

22: Williams, J., "Practical Circuitry for Measurement and Control Problems," Linear Technology Corporation, Application Note 61 (August 1994)

27: Haas, Isy, "Millimicrosecond Avalanche Switching Circuits Utilizing Double-Diffused Silicon Transistors," Fairchild Semiconductor, Application Note 8/2 (December 1961)

28: Beeson, R. H., Haas, I., Grinich, V. H., "Thermal Response of Transistors in the Avalanche Mode," Fairchild Semiconductor, Technical Paper 6 (October 1959)

Tektronix Type 111 Pretrigger Pulse Generator

This could be used with the Tektronix Type N Sampling Plug-In Unit, back when Tektronix was an oscilloscope company; in 01960 the Type N claimed an 0.6-ns rise time on its front panel; it was used to trigger an oscilloscope to repeatedly sample an otherwise-too-fast signal in the analog domain:

> The sampling system thus formed permits the display of repetitive signals with fractional nanosecond (10^{-9} second or nsec) risetimes. By taking successive samples at a slightly later time at each recurrence of the pulse under observation, the Type N reconstructs the pulse on a relatively long time base. … The sampling system formed by the combination of the N Unit and a conventional oscilloscope is quite different in operation from normal oscilloscope systems. A conventional oscilloscope system traces out a virtually continuous picture of waveforms applied to the oscilloscope input; a complete display is formed for each input waveform. The sampling system, however, samples the input waveform at successively later points in relative time on a large number of input pulses. From this sampling process, a series of signal samples is obtained. The amplitude of each signal sample is proportional to the amplitude of the input signal during the short time the sample is made. Input waveforms are then reconstructed on the screen of the oscilloscope, as a series of dots, from these signal samples. The oscilloscope bandpass required to pass the "time stretched" signal samples is much less than the bandpass which would be required to pass the original input signal.

The Type 111 pulse generator was used to transmit the timing information to the type-N sampler:

> As described previously, to trigger the Type N Unit you must first connect a triggering signal to either the TRIGGER INPUT or REGENERATED TRIGGER INPUT connector. When triggering signals are applied to the TRIGGER INPUT connector of the N Unit you must adjust the TRIGGER SENSITIVITY control for stable triggered operation.
>
> …
>
> When the Type 111 Pretrigger Pulse Generator is used, no triggering adjustments are necessary except to turn the TRIGGER SENSITIVITY control of the N Unit fully counterclockwise. The N Unit is started automatically each time a pulse from the 111 is applied to the REGENERATED TRIGGER INPUT connector of the N Unit.

I haven't been able to find the "Type 111 Operating and Service Manual", just the "Instruction Manual" from 01965 (59 pp.) This explains that it runs at up to 100 kHz ("kc") and has a risetime of 500 ps at at least 10 volts, which is *astounding* for 01960: "Determined from observed system risetime of 615 psec using a Tektronix sampling oscilloscope with a risetime of 350 psec. See Calibration section."

It evidently used an external coax “charge line” to produce a rectangular pulse, so you could hook up different lengths of cable there to produce pulses of different lengths, but only up to 142 ns for serial numbers below 800: “Exceeding these limits may damage the avalanche transistor, Q84.” Higher serial numbers could use pulse widths up to 1500 ns, so I guess they beefed up Q84.

You could also couple the pulse generator’s output pulse into the device under test, I guess so that what you were viewing on the oscilloscope was its pulse response.

The circuit is explained (p. 3–2, 22/59):

> **Output Pulse Generator (S/N 800-Up)**
>
> The positive output pulse from the Comparator blocking oscillator is applied to the Output Pulse Generator (avalanche stage) through C75 and D80. Since the collector voltage of Q84 is set just short of the point where the transistor will avalanche, when the voltage pulse from T60 turns on D80, a fast current pulse is applied to the base of Q84, causing the transistor to avalanche. This allows the internal charge line (and the external charge line, if any) to begin to discharge. The resulting positive voltage step at the emitter of Q84 produces the start of the output pulse.
>
> ...
>
> **Output Pulse Generator (S/N 101-799 only)**
>
> The positive output pulse from the Comparator blocking oscillator is applied to the Output Pulse Generator (avalanche stage) through two paths.
>
> One path is through C75 and R75 to the collector of Q84. The pulse which takes this path is a current pulse and is most effective when short time duration charge lines are used. The collector voltage of Q84 is set just short of the point where the transistor will avalanche. Consequently, when the positive pulse from Q60 is applied to the collector of Q84, the signal is sufficient to cause Q84 to avalanche.
>
> The second path, from T60 through C76 to the outer conductor of the internal charge line and to R77 and R78, couples a positive voltage pulse to the collector of Q84. This pulse is more effective than the current pulse at getting Q84 to avalanche when long charge lines are used. The internal charge line is passed through a ferrite toroid core (T78) to prevent the voltage pulse from being shorted to ground. The toroid core effectively isolates one end of the internal charge line.

So, fascinatingly, they redesigned the circuit to trigger through the base instead of by adding more voltage to the collector, starting with serial number 800! I guess they didn’t realize they could do that in 01960 and only figured it out around 01965.

There’s a parts list in the manual starting on p. 49 and absolutely beautiful schematics on pp.54–55/59 (initialed TR 964 and TR 366), annotated with expected oscilloscope traces in callouts and dc voltage levels as well. There are only three transistors in the whole instrument!

The all-important Q84 avalanche transistor was originally “Selected from 2N636”, but switched at serial number 800 to “Silicon Avalanche, checked”, with Tektronix part numbers. In the pre-800 schematic I think its V_{BC} is given as 37 volts, and its base is pulled down to a (germanium) diode drop below ground.

C75 is a 47 pF ceramic up to S/N 799, 10 pF in 800 and up, 500 V. The resistors are [carbon?] composition; R75 is 1kΩ, ±10% ½W up to 799, ±5% 1W in 800 and up. R77 and R78 are ½-W 10-Ω jobbies, deleted in 800 and up. T60, cleverly arranged so that the trigger-pulse-generating transistor Q60 that triggers Q84 turns itself

off, is a TD20 toroidal transformer up to 799, a 4T bifilar transformer in 800 and up (actually trifilar on the schematic).

D80 is exotic: for serial numbers X241–799 it's a Tektronix germanium diode, and for 800 and up it's a Tektronix gallium arsenide diode. (To be fair, most of the 16 diodes were germanium; only 2–4 were silicon, plus five more in a typewritten erratum stuck in the back of the manual.)

This probably explains why Williams didn't use a diode to block the current pulse surging back through the base of his avalanche transistor: his diodes were too slow! He probably didn't have a superfast GaAs diode handy, so he opted for ferrites.

The General Electric 2N636 was a 15MHz germanium NPN transistor specified for 20 volts of "BV_{cb}", 200 mA, and β=35, according to one 01962 compendium, or 300 mA and β=70, according to one from 01973. It appears in GE's 01958 Transistor Manual, categorized as "computer" rather than "audio", "amplifier & computer", "unijunction", "tetrode", or "IF", and rated for 300 mA, β=35, and only a 15-volt "punch through voltage" (p. 145, 143/167).

Fullwood 01960

Fullwood says:

> The pnp transistor types 2N501, 2N502, 2N504, 2N588, as well as the npn types 2N635, 2N636, 2N697, 2N706, and 2N1168 have all been found to avalanche with the same order of rise time [which he states in the abstract and later to be about 1 ns]. However, the decay time that is observed varies greatly with type, being related to the transistor's normal performance as a switch. ... One hundred and twenty 2N504 were tested as to whether or not they would avalanche at all in the circuit of Fig. 1. About 80% were found to operate satisfactorily with the zero bias arrangement as shown and without oscillating at this steady current.

An interesting thing about this is that he *was* triggering the avalanches with a pulse on the base, unlike the 01960 version of the Tektronix device. Because the 2N504 was pnp, it was a negative-going pulse, and the circuit was driven from a -300 V power supply. Trigger pulses were supplied from a "mercury pulser".

DOI 10.1063/1.1716847, "On the Use of 2N504 Transistors in the Avalanche Mode for Nuclear Instrumentation", by Ralph Fullwood (under Walter Selove) at U Penn (later at RPI), *Review of Scientific Instruments*, Volume 31, Number 11, November, 01960, interestingly the same journal that published Kilpelä and Kostamovaara 37 years later (see below).

He cites:

- D. J. Hamilton, J. F. Gibbons, and W. Shockley, Proc. IRE 47, 1102 (1959).
- I. A. D. Lewis and F. H. Wells, Millimicrosecond Pulse Technique (Pergamon Press, New York, 1959), 2nd ed.

This paper is interesting because it has a number of very simple circuits that do interesting things, like amplify the tiny pulses from a photomultiplier tube.

AN122: Never has so much trouble been had by so many with so few terminals

After Holme's project, Williams and David Beebe revisited pulse generators in Linear AN122 in 02009. In Appendix B, "Subnanosecond Rise Time Pulse Generators for the Rich and Poor", on p. 11/20, they explain:

> The Tektronix type 111 has edge times of 500ps, with fully variable repetition rate and external trigger capabilities. Pulse width is set by external charge line length. Price is usually about [US]$25. ... Residents of Silicon Valley tend towards inbred techno-provincialism. Citizens of other locales cannot simply go to a flea market, junk store or garage sale and buy a sub-nanosecond pulse generator.

Then they again present the circuit from AN94, unmodified as far as I can tell, but this time its performance has been derated again, to a 400ps rise time. And in Appendix F, they explain, "The Tektronix type 109 mercury wetted reed relay based pulse generator will put a 50V pulse into 50Ω (1A) in 250ps." Perhaps this is the "mercury pulser" Fullwood was talking about.

Wong's Reverse Avalanche

Kerry Wong revisited the theme in 02014 using the lower-voltage emitter-base junction as I suggested above, producing the following table of emitter reverse breakdown voltages with a 1000μF (!!!) cap:

2N4401 ~12.5V
SS9014 ~12.5V
2N4124 ~12V
2N3904 ~12V
BD137 ~11V
BD139 ~11V
BC337 ~9V
SS9018 ~8.2V

He found some important limitations:

> Also, while I could get most NPN transistors to oscillate in their reverse breakdown regions I could only get a couple of BD138 PNP transistors to oscillate using the same circuit above (power polarity is reversed). And the oscillation only occurred at a very tight voltage interval (e.g. ±0.05V).
>
> One of the useful features of a standard avalanche pulser (like this one [linking to his other project]) is its extremely fast rise time (subnanosecond), so can we use negistors to build similar pulsers?
>
> Well, the short answer is no. After some experiments it appeared that the rise time of a negistor pulser is magnitudes higher (e.g. ~100ns) than a typical avalanche pulser.
>
> ...the capacitance cannot be arbitrarily small. In my case, 100nF seems to be near the lower limit.

Importantly, he says in the comments:

> Just the e-b junction won't work, it would just act like a Zener diode.

analogspiceman posted the following SPICE model in the comments:

```
* UpsideDown.asc - a single transistor relaxation oscillator model for LTspice
V1 1 0 10
R1 1 2 1k5
C1 2 0 1µ Rser=8m
XQ1 0 NC_01 2 2N2222r
*
.subckt 2N2222r e b c ; this subckt just turns the NPN upside down
Q1 c b e 2N2222r
.model 2N2222r npn Is=10f Xtb=1.5 Rb=10 ; nondirectional parameters
```

```
+ Br=200 Ikr=0.3 Var=100 tr=400p ; reverse (forward) parameters
+ Bf=7 Ikf=0.5 Vaf=10 tf=100n Itf=1 Vtf=2 Xtf=3 Ptf=180 ; fwd (rev) params
+ Re=.3 Cje=8p Ise=5p ; emitter (collector) parameters
+ Rc=.2 Cjc=25p Isc=1p BVcbo=7 ; collector (emitter) parameters
.ends 2N2222r
*
.opt plotwinsize=0
.tran 0 10m 0 1u uic
```

Wong mentions the term "negistor" Richard Phares used in Popular Electronics in 01975 for this configuration (an avalanche discharge has negative differential resistance, so "negative resistance transistor"). Phares notes that germanium transistors and pnp transistors will not work, recommending the MPS-5172, the 2N2218 (7.7V), the 2N2222, or the 2N697. Unfortunately, the term "negistor" seems to have been largely co-opted by Keelynet crackpots lacking even the most basic knowledge of physics and electronics. However, Alan Yates, for example, built some oscillators using the term. Prolific electronics hacker sv3ora reports, "The 2N4124 gave the lowest oscillation voltage, around 6.8V," and confirms that grounding the base kills the oscillation. Jean-Louis Naudin reports oscillation at 16.4 volts on a 2N2222A and also characterizes its available stable avalanche currents ranging from 5.47 V at 10 mA up to 6.54 V at 2 mA.

Kilpelä and Kostamovaara's 01997 laser

These folks wanted to make 5–10-ns semiconductor laser pulses for LiDAR, but at tens of amps. They said a transistor in avalanche mode is faster than a thyristor or MOSFET, though GaAs thyristors were reported to have reached the 500-ps-level most of the above discussion has centered on. They tried an MJE200, a 2N5190, two 2N5192s, and a Zetex ZTX415 SOT-23 avalanche transistor; the MJE200 started breaking down below 100V but was consistently only about 15A no matter how high the voltage, while the others all required 250V to break down, reaching 70 A at 400 V. These all got rise times in the 2.5–4 ns range; this extreme slowness (ha!) is probably because parasitic inductances matter more at 70 amps than at 1 amp.

Their circuit is very different from all the others I've seen, full of inductors, and I don't understand it yet. The paper has lots of good explanation about how avalanche transistors work, though.

70 A at 400 V for 10 ns is 28 kW, but only about 0.28 mJ.

Alex McCown's

Alex McCown (onebiozz) built a pulse generator to test his oscilloscope around a 2N3904, getting a 1.56-ns rise time (which he thinks is the scope's limit, not the circuit's) but wished he'd used a BFR505:

> I have to say this was a fun $0 project, but if i were to spend some cash what would i have done differently knowing what i do now? Well for one i would not use an 2N3904, the BFR505 appears to be a better solution at a simple 30v avalanche of ~200-300pS.

M. Gallant's speed-of-light measurement

Michel I. Gallant put 20ns pulses through an infrared LED using a 2N2369a avalanche transistor to measure the speed of light to within about 1% in his living room, but the 25 MHz Vishay TSFF5210 LED he chose slowed their rise time to 10ns. Very simple perfboard circuit. As a detector he used a 200 MHz Vishay BPV10 PIN photodiode amplified by an AD8001 configured for 35× gain and 50MHz, but they also built the circuit on a solderless breadboard, so it might have suffered some signal integrity problems from that and from the long leads on their components too.

Also interesting for fast-circuit purposes, he measured the response of different common LEDs up to 10MHz: the TSFF5210 had drooped less than 1dB at 10MHz, a red 08LCHR5 AlInGaP drooped 3dB, and a white 08LCHW3 InGaN drooped 3dB at 2MHz and 6dB at 3MHz. Presumably that's a composite of fast blue and slow yellow, but the pulse response he shows doesn't show much fast blue.

Michael Covington's notes

Covington notes that the avalanche effect of the emitter-base junction makes a good white noise source, and also a good low-leakage low-capacitance "zener diode", citing EEVBlog #1157.

Topics

- Electronics (p. 1145) (39 notes)
- Pricing (p. 1147) (35 notes)
- Pulsed machinery (p. 1167) (12 notes)
- Numerical modeling (p. 1229) (5 notes)
- Sparks (p. 1240) (4 notes)
- LEDs (p. 1286) (3 notes)
- LiDAR (p. 1355) (2 notes)
- Avalanche breakdown

The astounding UI responsivity of PDP-10 DDT on ITS

Kragen Javier Sitaker, 02021-10-22 (updated 02021-10-23) (28 minutes)

I just watched Lars Brinkhoff's demo of PDP-10 programming in the DDT debugger under ITS, (cheat sheet for mostly using DDT as a shell, newbie guide to using DDT for debugging, AIM-147a describing an earlier version of DDT from 01971) which is truly astounding. Why couldn't GDB be half this good?

To be clear, it's not that DDT can do anything GDB can't. GDB is vastly more powerful, and I think that was true even in its earliest versions. It's that the things DDT *can* do are done with great grace and fluency, and they are closely analogous to things I do all the time in GDB, where they are very clumsy.

Summary of the video

Brinkhoff, an enthusiastic PDP-10 novice, demonstrates "programming in the debugger", a technique Minsky was famous for, interactively writing a hello-world program in PDP-10 assembly (I think 7 instructions), incrementally, then saving the resulting memory image as an executable. He works by repeatedly executing the partly written program; when it tries to execute uninitialized (zeroed) memory, it halts and disassembles the offending instruction, and then Brinkhoff adds assembly instructions to it, then continues execution. In that, it's fairly similar to its contemporary interactive environments for BASIC, FORTH, or LISP in the 01970s, and MS-DOS's DEBUG.COM and CP/M's DDT.COM were capable of similar feats, although I don't think they had an easy way to initialize memory to all illegal instructions or debug breaks.

The UI seems to be designed for a teletype printing terminal, though a full-duplex one (ESC is echoed as $, as in TECO and CP/M ED.COM), which is pretty limiting; it's impossible to have a live display of *anything*, even the current program counter or registers. And I don't want to do all my programming in assembly language, and interactively patching the machine code of a broken or incomplete program is not something I spend a lot of time on, and it's what Brinkhoff spends most of the video on. So, what's so great about it?

The ways DDT is head and shoulders better than GDB

What's amazing to me is the stuff Brinkhoff can do *instantly*, which don't take up much of the video, but which make up *most* of what I do in GDB.

Examining memory

To see what's at location (octal) 100, he types `100/`; DDT immediately responds with the disassembled instruction at that

location, or, failing that, its numeric value, leaving the cursor ∎ at the end of the line to permit more operations on either the location (like putting an instruction there) or the value (like following it to where it points in memory with another /:

```
$g
ILOPR; 100>>0   0/   0   0/   0
100/   0   ∎
```

As it happens, / actually uses the *last 18 bits* of the expression as the pointer, because the PDP-10 used 18-bit addressing. The significance of this will be explored later.

Numeric display and label definition

In this case, he types `.=` to ask for the value of `.`, the current location, interpreted numerically, and then `.  go:` to define a new symbol GO with that value, all without ever hitting Enter:

```
100/   0   .=100   . go:   ∎
```

In a sense, the three spaces are like FORTH's `ok` prompt, but don't send you to a new line. (But at this point Brinkhoff hits Enter to go to a new line anyway, for reasons I do not know.)

(As it happens, according to the September 01971 DDT reference manual, §XII, p. 38 (40/84), not even this is needed; Brinkhoff could have just typed `go:`, leaving `.` implicit, but the ITS mentorship lineage has been broken, and Brinkhoff is reviving it from artifacts. It's possible he's using a version of DDT whose semantics had changed, too.)

Disassembling memory in GDB

A similar command to `100/` in GDB, but using a longer address since 0x40 is in the zero page Linux never maps:

```
(gdb) x/i0x80495c5
   0x80495c5 <addr>:    add    (%eax),%al
```

Instead of /, one keystroke, I had to type `x/i↵`, 4 keystrokes, with the address in the middle.

Numeric value display

GDB stores the address in the convenience variable `$_`, so instead of typing `.=` to see it (perhaps superfluous in this case, since GDB automatically displayed it as part of the `x` output) I can type:

```
(gdb) p $_
$6 = (int8_t *) 0x80495c5
```

That's `p $_↵`, 5 keystrokes instead of 2.

Convenience variable creation in GDB

Now, if I want to store that in a new variable called `go` (I haven't found a way to get GDB to create new labels at runtime) instead of `.`

go: (4 keystrokes) I can type `p $go=$↵` (8 keystrokes), where `$` is GDB's name for the last value output by `p`. (`set $go=$↵` is silent and doesn't clobber `$`, but is more awkward to type.)

```
(gdb) p $go=$
$8 = (int8_t *) 0x80495c5
```

Intermission: GDB is 10 strokes over par and in the sand trap

So at this point the golf score is 7 (key)strokes for ITS DDT, 17 for GDB, not counting typing the address. *Programming* golf is not a good metric on which to compare *programming* languages, but in this case we're counting *user interface actions* that must be taken to reach a goal.

But we haven't really reached the same goal, because DDT will use the label GO to make future disassembly more readable, and GDB won't.

Easy or hard variable and memory access

Brinkhoff's next move is to see the value of GO `go=` (superfluous in both GDB and DDT) and then examine memory there, `↵go/` (which I think could have just been /):

```
go=100
go/   0   ▮
```

To do the same in GDB is 10 keystrokes instead of 7:

```
(gdb) p $go
$10 = (int8_t *) 0x80495c5
(gdb) x $
   0x80495c5 <addr>:     add    (%eax),%al
```

(As it happens, the data I have stored there is actually a `sockaddr_in` struct, but GDB doesn't know that; it's disassembling because the last time I told it how to examine memory it was with `x/i`.)

Focus on the PC

A funny thing about GDB is, not only doesn't it disassemble the instruction the program counter is at by default, the default thing to examine is the thing after the thing you last examined, and the default format is the format you last examined something in. For example:

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: ...

Breakpoint 1, 0x08048151 in main ()
(gdb) x
0x80495cb <addr+6>:     0x0000
(gdb) x $pc
0x8048151 <main>:       0x73b9
```

```
(gdb) x/i$pc
=> 0x8048151 <main>:    mov    $0x8049573,%ecx
```

That x/i$pc↵ at the end is what DDT does by default, with no keystrokes, because when it hits an exception (or, I think, a breakpoint), it sets the current location . to PC.

In this case the instruction contains an immediate which is in fact a pointer. It might be useful to look at memory at this pointer, but in GDB the most convenient way to do this by copying and pasting the address from the screen so you can type x 0x8049573↵. On the PDP-10, such immediates are packed into the right half of the instruction word (as I understand it, this instruction might have said MOVEI 2,563 to load octal address 563 into accumulator 2) so you could just type /.

Effectively, DDT turns not just the living data of your program into hypertext, but even the machine code, so you can just follow the link with a single keypress. (It even had a $e command for searching all of memory for such pointers to a given address.)

Switching between alternate display representations: SIXBIT and sockaddr_in

Later after Brinkhoff has added an instruction, he disassembles the same place again with go/ and then continues on to disassemble the next word (the PDP-10 was a 36-bit word-addressed machine) with, I think, ^J, which displays the address relative to the most recent defined label:

```
go/   OPEN TYOC,
GO+1/   0   ▮
```

Typing ↵ in GDB (or ^J) will also continue to disassemble the next instruction, because it repeats the last command (with slight tweaks), but using my convenience variable $go to contextualize it:

```
(gdb)
   0x80495c7 <addr+2>:  pop    %ds
```

That also isn't really an instruction; it's actually the first byte of a TCP port number. Brinkhoff encountered a similar problem with a variable he called TTY. It packed two variables into a 36-bit word; one is an 18-bit I/O unit number (?), and the other is the three-character SIXBIT string "TTY". But DDT tries to disassemble it as an instruction when he types tty/, treating them as an opcode and an operand:

```
tty/   TYOC,,646471   ▮
```

So, to change his view of the memory, he types an apostrophe ' to see it as SIXBIT:

```
tty/   TYOC,,646471   '$1'  !TTY'   ▮
```

The $1' syntax DDT outputs is the same input syntax it supports for SIXBIT strings, but Brinkhoff had terminated his with altmode

(ESC, $) which I think is mandatory.

As far as I can tell, the shortest way to do the corresponding thing in GDB is x/h$_↵, 6 characters:

```
(gdb) x $go
   0x80495c5 <addr>:     add    (%eax),%al
(gdb) x/h$_
0x80495c5 <addr>:        0x0002
(gdb)
0x80495c7 <addr+2>:      0x961f
```

Here h means "halfword", 16 bits, which is appropriate because the first two fields in a sockaddr_in are 16-bit fields, though unfortunately the port number is in network byte order, and the i386's byte order is not network byte order.

Switching between alternate display representations: floats and ASCII strings

The ; key in DDT prints a value in "semi-colon" mode, initially floating-point, instead of SIXBIT or octal or machine instruction or whatever. (There are two-character commands to change semicolon mode to be any of the available "type-out modes".)

Sort of amusingly, DDT doesn't remember the types of the values Brinkhoff stores; for example, the words starting at his label HELLO are actually strings (in ASCII (§X.B.5), which supports lowercase but only gets five bytes per 36-bit word, using $1" instead of $1' in the UI). So he has to tell it again. Here's the part of the session where he first enters the strings, then starts looking at them, and it starts disassembling them as instructions:

```
hello/   0   $1"hello$
HELLO+1/   0   $1" worl$
HELLO+2/   0   $1"d

$
hello/    TLCE B,@466337(15)   ▮
```

At this point he presses " and sees the right view, then presses (I think) ↵ to go to the next word, repeating the process:

```
hello/   TLCE B,@466337(15)   "$1"hello$
HELLO+1/   MOVES 13,@771331(15)   "$1" worl$
HELLO+2/   TRZ 6,@200001(A)   "$1"d^M^J$   ▮
```

So to see those three words of ASCII, given the start address, he had to type /"↵"↵", 6 bytes. In some sense GDB is a little better about this sort of thing, part of which is because character data is much less of a pain on a byte-oriented computer; I only need x/s$_↵, also 6 bytes, to change my view to NUL-terminated text:

```
(gdb) x &text_plain
0x80496e3 <text_plain>: 25972
(gdb) x/s$_
```

```
0x80496e3 <text_plain>:  "text/plain; charset=utf-8text/css; charset=utf-8applica⚬
⚬tion/pdf\r\n\r\n"
```

(Let's forget about the `&` here for the time being; it's a thing that I constantly mess up, because even though GAS demands `$` prefixing, when I'm programming in assembly I think of `text_plain` as being *the address* where I set that label (or the value I EQU it to), but GDB thinks `text_plain` means "the contents of memory at address `text_plain`"; this is particularly confusing because `x/i main`↵ will in fact treat the symbol for function `main` as the memory address to examine, rather than a place to look for a pointer to it. But it's not clear which interpretation actually imposes more keystrokes on the debugger user.)

GDB UI mode errors

GDB's cleverness here can lead to subtle mode problems in the UI, because `/s` isn't quite a unit size specifier as strongly as it is a format specifier:

```
(gdb) x/h$_

0x80496e3 <text_plain>:  u"整瑸瀯慬湩※档牡敳㵴瑵\x2d66琫硥⽴獣㭳挠慨獲瑥甽晴㡾灡汰捩瑡潩⽮摰屦     \n"
```

Getting back to seeing memory as two-byte integers requires an explicit output format, thus `x/dh$_`↵ (7 keystrokes) instead of DDT's `=`:

```
(gdb) x/dh$_
0x80496e3 <text_plain>: 25972
```

Stepping backwards and changing memory

Above I mentioned that in both GDB and DDT you can move forward in memory you're examining by just hitting ↵. But later on Brinkhoff steps *backwards* in memory to get back to an instruction he wants to change, using apparently the two-keystroke sequence ↑↵, which his terminal renders as `^`:

```
LOOP+4/   0   . die:
die/   0   .value
^

LOOP+3/   JRST LOOP   ^
LOOP+2/   .IOT A,B   ^
LOOP+1/   JUMPE B,   ▮
```

Stepping backwards over i386 instructions is probably too much to ask for, since in numerous cases there are i386 instructions that are proper suffixes of other i386 instructions. But what about halfwords? The best approach I've found is `x $_-1`↵ (7 keystrokes instead of 2).

```
(gdb) x/xh&addr
0x80495c5 <addr>:       0x0002
```

```
(gdb)
0x80495c7 <addr+2>:     0x961f
(gdb) x $_-1
0x80495c5 <addr>:       0x0002
```

Surely, in many cases, the whole necessity to step backwards is avoided by GDB's facility at dumping vaast tracts of … memory:

```
(gdb) x/20xh &bind_args

0x80495b9 <bind_args>:  0x0000  0x0000  0x95c5  0x0804  0x0010  0x0000  0x0002  0⚬
⚬x961f

0x80495c9 <addr+4>:     0x0000  0x0000  0x6962  0x646e  0x2928  0x0000  0x0000  0⚬
⚬x0500
0x80495d9 <listen_args+5>:      0x0000  0x6c00  0x7369  0x6574
```

(The egregious display misalignment of the DDT output was bothering me, but here we've caught GDB at it too. Also, those lines are 86 characters wide.)

In recent GDB 7.12, negative repeat counts have been added to `x` to allow you to examine memory backwards. So you can start stepping backwards through memory with something like `x/-1↵↵↵`, returning to stepping forward with `x/1↵`.

But, in the video, Brinkhoff was navigating backwards in order to find where he wanted to *set* something, a pointer to a label in fact, although the place he was poking it into was an instruction. I feel like this is a thing you might reasonably want to do with GDB too. For example, I might want to change the port number the server binds to to 1536, which byte-swaps to 6; DDT is *really* designed for this kind of thing, allowing you to type numbers or assembly code whenever you want and just poking it into memory wherever you are, so you could set a variable to 6 just by typing "6↵". Here's what the interaction looks like in GDB:

```
(gdb) x/xh&addr
0x80495c5 <addr>:       0x0002
(gdb)
0x80495c7 <addr+2>:     0x961f
(gdb)
0x80495c9 <addr+4>:     0x0000
(gdb) x $_-1
0x80495c7 <addr+2>:     0x961f
(gdb) p*$_=6
$19 = 6
(gdb) x/3 &addr
0x80495c5 <addr>:       0x0002  0x0006  0x0000
```

To set the current memory location to 6, I typed `p*$_=6↵`, 7 keystrokes instead of 2. Or 6 keystrokes instead of 1, if we leave out the actual value I'm setting it to.

Of course I don't want to switch to DDT,

I'm not insane

Well, I mean, I *am* insane, just not in that particular way. GDB supports the computer I actually have, high-level programming languages, and operating systems I actually want to use like Linux. It's scriptable, including in Python (as well as in its own UI command language, which is not only unreadable but also by far the slowest interpreter I've ever used) and I can sort of remedy the annoying fact that it defaults to not disassembling the instructions it's stopped at, even for programs with no debug info where it obviously can't show me the high-level source, by launching its TUI with `layout asm` or just `display/i $pc`. It has time-travel debugging, though it's far too slow to use for anything but very short runs, and watchpoints, which are fast. It remembers not just 8 printed-out expressions but all of them. (According to the 01971 manual, DDT has a ring of the last 8 *locations* and another of the last 8 previous *expression values*.)

Moreover, modern debugger UIs like WinDbg, radare2, and the IntelliJ IDEA debugger offer lots of improvements, like expression watch windows, memory view windows, and graphical control-flow graphs.

Here's an edited excerpt of one of my GDB sessions a few months ago:

```
(gdb) p *seq

$30 = {capacity = 0, used = 93824992367072, arena = 0x0, elements = 0x5555555c1728}
(gdb) s
262             uint8_t b = 0;
(gdb)
263             HCountedArray *seq = H_CAST_SEQ(p->ast);
(gdb) n
264             size_t digits_processed = 0;
(gdb) p *seq
$31 = {capacity = 4, used = 2, arena = 0x5555555bfbd0, elements = 0x5555555c17a8}
(gdb) p $.elements
$32 = (struct HParsedToken_ **) 0x5555555c17a8
(gdb) p *$@2
$33 = {0x5555555c1da8, 0x5555555c2558}
(gdb) p *$[0]

$34 = {token_type = TT_UINT, {bytes = {token = 0x7 <error: Cannot access memory at address 0x7>, len = 0}, sint = 7, uint = 7,

    dbl = 3.4584595208887258e-323, flt = 9.80908925e-45, seq = 0x7, user = 0x7}, index = 0, bit_length = 0, bit_offset = 0 '\000'}
(gdb) p *$$[1]

$35 = {token_type = TT_UINT, {bytes = {token = 0xd <error: Cannot access memory at address 0xd>, len = 0}, sint = 13, uint = 13,

    dbl = 6.4228533959362051e-323, flt = 1.821688e-44, seq = 0xd, user = 0xd}, index = 0, bit_length = 0, bit_offset = 0 '\000'}
```

By contrast, following a cdr-linked list in ITS DDT was a matter

of typing a / or [(which forces numeric output) at each node, if each word contained two pointers with the (18-bit) cdr packed into the right half of the word. If you instead wanted to follow a pointer in the left half of the word, maybe a car, it was $/ or $[. `p*$[0]↵` is not only more than three times as long, it's also a lot harder to type, with two shifted keys not adjacent to the home row (four for me, since I've swapped () and [], but I mean for normal people).

(Remember that in DDT $ is "altmode", the ESC key, which was not shifted and usually in a much more convenient place than on modern keyboards.)

It's really thought-provoking that the things I do *most of the time* in GDB, which require awkward commands full of hard-to-type line noise characters, like following chains of pointers or getting an alternate display of the thing I just looked at, require *dramatically* less typing in *the debugger Stallman maintained and used on a daily basis before he wrote the first version of GDB*, because they're bound to single-keystroke commands. And there's apparently no way at all to add a label to an address once you figure out what it means so that GDB will use it in its output. I have no idea how this could have happened! My best guess is that he thought a DDT-like user interface would be too alien to Unix programmers accustomed to dbx, so they wouldn't use it. But Unix had adb, which was very similar (see below).

Lessons for debuggers and similar programs

I think there are a few principles to extract here.

One is the use of single-keystroke commands for the most common things; the difference between 7 keystrokes and 10 is maybe marginal, but the difference between 1 keystroke and 4, or even 1 keystroke and 2, is enormous, if it's something you're doing frequently, *especially* repeatedly. Unfortunately, this clashes pretty strongly with modern modeless UI conventions; the / key should always insert a /, not do something like follow a pointer. Some possible compromises here:

- Insert the /, and then react to the textual change, which is sort of what DDT is doing.
- Use control-/ or alt-/.
- Use an onscreen button, perhaps contextually available.

(Uses of DDT as a command shell, file manager, task manager, etc., also followed this approach; instead of `ls↵` you would just type ^F, and instead of `bg↵` or `%&↵` you would type ^P.)

Another is the importance of immediate feedback. Even if 7 characters like `p*$_=6↵` is a reasonable length for a command to set a memory location (given how much less important the debugger is nowadays as a way of loading data into memory), it would be better to display the current contents of the memory once you get to `p*$_`, and maybe to use a postfix operator (like DDT's / or Pascal's ^) instead of the prefix * operator used in C and Rust. Like recent versions of Android's calculator app, you should compute and display the value of any expression being entered whenever this has no side effects.

A thing that barely reared its head here is the importance of

reversibility for user interfaces; this is really the main reason I don't use debuggers much. Following a pointer or stepping forward through memory is normally reversible (Brinkhoff maybe didn't know this, because he would start over when he followed a pointer chain too far, but `$↵` goes to the previous location in the `.` ring buffer) but single-stepping a program rarely is, so when I'm using a debugger I often go very slowly to avoid having to restart my debugger session from the beginning.

By contrast, when the program runs fast enough, I can debug it by progressively adding tests, assertions, and logging, and running it a very large number of times, without ever having to slow down to avoid stepping just one step too far and losing minutes or hours of work. As computers have gotten faster and faster, this monotonic approach has become more and more appealing. A debugger in which almost all actions could be undone (and in which the irreversible actions were easily distinguishable) would allow me to use it much more quickly.

A *lot* of DDT's UI's advantage over GDB's is its implicit focus on "the current location" and "the current value", analogous to "the selection" in many GUI systems or "the top of the stack" in systems like HP RPN calculators, Forth, and PostScript; this avoids GDB's requirement to explicitly name `$` or `$$` or `$_` all the time. It's not yet clear to me to what extent this transfers to touchscreen UI design, but it seems pretty central to keyboard UIs. In csh I would frequently use `$!` to avoid having to name the same file repeatedly in subsequent commands, and in bash I use M-. all the time for the same reason. The big difference is that, in a debugger, the values of interest are not numbers or filenames, but regions of memory with associated interpretation information — you might say "with types", but this information might also include things like how many digits of floating-point precision you want to display or whether child nodes should be collapsed or expanded.

It's surprising that GDB didn't copy this from DDT or one of its predecessors, particularly since MDB *did*:

> MDB retains the notion of dot (.) as the current address or value, retained from the last successful command. A command with no supplied expression uses the value of dot for its argument.
>
> ```
> > /X
> lotsfree:
> lotsfree:       f5e
> > . /X
> lotsfree:
> lotsfree:       f5e
> ```

MDB copied this from Stephen Bourne's `adb`; quoting the ADB tutorial from 01977:

> ADB maintains a current address, called dot, similar in function to the current pointer in the UNIX editor. When an address is entered, the current address is set to that location, so that:
>
> **0126?i**
>
> sets dot to octal 126 and prints the instruction at that address. The request:
>
> **.,10/d**

prints 10 decimal numbers starting at dot. Dot ends up referring to the address of the last item printed. When used with the ? or / requests, the current address can be advanced by typing newline; it can be decremented by typing ^.

It's interesting to note that /, ˆ (or ↑ in ASCII-1963), and . have the same functions as in DDT, but in adb they required you to type a newline, as I think they did in some versions of DDT. `adb` also, like DDT, uses : as a prefix for some extended commands. I'm not sure whether these feature are inherited from Dennis Ritchie's earlier Unix debugger DB.

Another aspect of this is the ease with which DDT switches between different presentations of the current value, with keys like `'`, `"`, `=`, and `;`. In DDT's case, these are a fixed, closed set, and entirely insensitive to context, but even more useful would be an extensible set of pretty-printers like GDB has — obviously posing the difficulty of how to assign keys to them, for a keyboard interface — and perhaps the possibility of backtracking as parsers do.

A lightweight version of this facility is present in DDT in the sense that by defining symbols you can enhance its future display of addresses and instructions: it will use those symbols to clarify its output; instead of `JRST MAIN+21` perhaps it will say `JRST MAINLOOP`.

WinDbg can also switch between presentations of different memory regions without having to name the region explicitly, but it's a pull-down menu, so it requires three mouse operations, roughly equivalent to keystrokes. At least it by default displays the memory around PC, but AFAICT none of the memory-dump options is "disassemble", which is usually what I want to do around PC.

One benefit of command-line interfaces like GDB's is that they provide an easy and somewhat readable extension mechanism: by putting a sequence of commands in some sort of container, you have a macro-command; and in GDB, hitting ↵ repeats the most recent command line, which thus allows you to repeat a complex command. The GDB manual gives this example:

> One of the ways to use a convenience variable is as a counter to be incremented or a pointer to be advanced. For example, to print a field from successive elements of an array of structures:
>
> ```
> set $i = 0
> print bar[$i++]->contents
> ```
>
> Repeat that command by typing \<RET>.

A sequence of DDT commands can of course also be canned; it's a keyboard macro. And sometimes that's the best you can do. A simple improvement over most such macro facilities would be to interactively roll up the last N commands as a "macro" *after* the fact, once you realize you want to repeat them.

But I suspect that a more complete programming-by-demonstration facility could provide a great deal of programming power in a much more usable way.

Topics

- Programming (p. 1141) (49 notes)
- History (p. 1153) (24 notes)
- Human-computer interaction (p. 1156) (22 notes)
- Assembly-language programming (p. 1175) (11 notes)
- End user programming (p. 1217) (6 notes)
- Debugging (p. 1375) (2 notes)

Example based regexp

Kragen Javier Sitaker, 02021-10-24 (updated 02021-12-30)
(5 minutes)

Programming by example ought to be easily applicable to regular expressions, and by extending that with computations and assertions, it should be possible to make a usable nondeterministic programming system.

By typing the string "bye" I am writing a program that generates the string "bye", and if I'm doing it in Emacs's incremental-search, all the instances of "bye" will be highlighted. If I then back up back to the "b", maybe by twisting a time-turner or hitting shift-backspace, and type "ad", I have programmed the regexp "b(ye|ad)", which is a nondeterministic program that can generate the strings "bye" and "bad". I might then want to hit some other key to cross the two streams back together and then type "law", so I have the regexp "b(ye|ad)law". A third key could progressively cast a spell of Kleene closure over a gradually-increasing number of the last few nodes in the graph, giving the following progression:

```
b(ye|ad)law
b(ye|ad)law*
b(ye|ad)l(aw)*
b(ye|ad)(law)*
b((ye|ad)law)*
```

As I'm doing this, it's reasonable to concurrently display random strings generated from the program, the shortest strings generated from it, a FSM diagram of it, a derivation tree of the regexp itself, the matches it finds in searching some corpus of text, and/or derivation trees for those matches as well.

It is, I think, straightforward to add zero-width negative lookahead assertions and positive lookahead assertions to the UI in the same way. Providing a UI for tagging some subexpression of the regexp for reuse, then reusing it elsewhere, is straightforward from the UI perspective, but of course increases the expressiveness of the system to the point where it can no longer be parsed precisely with a finite state machine.

If, instead of typing a string of characters, I type a sequence of assignment statements, then my program, instead of producing a sequence of characters from nothing, produces an output set of variable assignments from a (usually smaller) input set. In the same way, I can move around my program and add alternatives or iteration to parts of it, making it nondeterministic; if I then add assertions to parts of it, which fail if some condition is not met, I may be able to make it deterministic again.

Such a program can have arguments, which are variables that it may read from before assigning to them. If all my operators are monomorphic, it can infer some sort of types for all of these variables, and inject some example values. And extracting an expression or sequence of statements into another subroutine is, if not trivial, at

least a straightforward thing to do. All of this can be displayed as I'm typing.

The traditional and terser alternative to this sort of assertion-pruned nondeterministic control flow is Boehm-Jacopini control flow, consisting of sequencing, conditional repetition, and if-else conditionals. You could imagine a key to extend an if-block backwards from your cursor, whose condition was initially just `true` and whose else-block was initially empty (or inserted assignments to any variables needed to keep the following parts of the program happy), and perhaps a while-block could initially use the condition `false`, with the selection left on the condition in either case.

A different approach is Dijkstra's guarded-command language, in which I suppose you could do the analogous thing.

The approach I'm most interested in at the moment, though, is Baker's COMFY approach. While the Boehm-Jacopini constructs, like Kleene's constructs, have a single entry and a single exit, Baker's forms of combination have one entry and *two* exits: a success exit ("win") and a failure exit ("lose"). His conditionals and loops are ruled not by a *value* computed by their conditionals, but, as in Icon and Unicon, by whether those "conditionals" succeed or fail. IIRC you can reduce Baker's combinations to three, like those of Kleene and those of Boehm and Jacopini, consisting of the following connections:

```
do x y:
    entry = x.entry
    x.win = y.entry
    y.win = win
    x.lose = y.lose = lose

if x y z:
    entry = x.entry
    x.win = y.entry
    x.lose = z.entry
    y.win = z.win = win
    y.lose = z.lose = lose

while x y:
    entry = x.entry = y.win
    x.win = y.entry
    x.lose = win
    y.lose = lose
```

This straightforwardly permits the construction of early exits from loops or from subroutines, error handling, and short-circuit Booleans, and it is easy to compile with a recursive strategy starting from the end of a subroutine.

Topics

- Programming (p. 1141) (49 notes)
- Human-computer interaction (p. 1156) (22 notes)

- Program calculator (p. 1246) (4 notes)
- Domain-specific languages (DSLs) (p. 1260) (4 notes)
- Kleene algebras (p. 1287) (3 notes)
- Control flow (p. 1299) (3 notes)
- COMFY-* (p. 1300) (3 notes)
- Incremental search (p. 1362) (2 notes)

Adversarial control

Kragen Javier Sitaker, 02021-10-25 (updated 02021-12-30)
(13 minutes)

Model predictive control is now the standard approach to control systems: given a model of the system and an objective function, you use an optimization algorithm to seek a behavior that maximizes your objective by modeling the effects of different candidate behaviors on the system.

The reinforcement learning problem is in some sense more general, in the sense that in model predictive control, the control system is given a model of the system being controlled (the "plant") which is presumed to be correct, so it can predict the effects of its actions, but a reinforcement learner doesn't initially know what behaviors will have what effects; it must build that world model by observing the results of different behaviors. This means that it benefits greatly from doing some undirected exploration, focusing its actions on the parts of the possibility space where its predictions are the least confident.

This is somewhat disquieting for someone contemplating control systems for a chemical plant or a satellite, where incorrect control commands could end a million-dollar mission or cause a toxic-waste release into the neighborhood. But of course it is not only undirected exploration of the system's parameter space that can cause such results; running a model predictive control system with an incorrect model can also cause them.

The amount of undirected exploration that is needed diminishes over time as the control system solidifies its system model, a process sometimes called "system identification" in the literature on dynamical systems modeling and control theory, so one way this can be handled is by initially training the control system in some sort of "sandbox" where its ability to cause damage is limited; the humans call this "childhood" when they provide it for their larvae.

Another safety strategy is to focus experimentation on the possibilities that your existing system model tells you have low probability of causing significant harm, but which have high uncertainty in some other dimension that isn't so costly. Call this "safe experiment design": a human might use high-powered lasers to can observe effects only observable with high-powered lasers, but wear the correct laser goggles to reduce the chance of being blinded in the process.

But a third, and I think most important, strategy for this kind of system identification is *dreaming*, and generative adversarial networks (GANs) are a strategy used with great success for dreaming in the world of artificial neural networks. Essentially the idea of a GAN is to optimize a generative model of some kind of probability distribution, such as the probability distribution of photographs of dogs, or the probability distribution of satellite telemetry traces, by playing off two networks against one another: a generator and a discriminator. The generator generates random deviates from its approximation of the distribution, and the discriminator distinguishes

these adversarial inputs from real training-set inputs.

You alternately optimize these two networks, typically with some variant of gradient descent driven by automatic differentiation. By carrying out automatic differentiation all the way through the generator and discriminator, we can find the gradient of the generator's parameters that would worsen the discriminator's ability to discriminate, using that to drive an optimization algorithm like Adam; and by differentiating just through the discriminator, we can find the gradient of the discriminator's parameters that would improve its ability to discriminate. It's important to optimize these somewhat in lockstep; if either network gets too far ahead of the other, the loser will stop improving. If done correctly, the generator produces inputs that are very, very difficult to distinguish from real inputs from the training set.

(The typical "networks" here are standard ANNs, where matrix multiplies and weight-vector additions alternate with ReLU and maybe convolutional and pooling stages, but almost any computational model could potentially be used. Being differentiable enables the use of derivative-based optimization algorithms, and it's important to have enough expressivity to reasonably represent the distribution in question but not so much that you overfit the training set, but there is an enormous field of unexplored models here.)

As far as I know, nobody is using GANs for system identification for control systems, much less model predictive control in particular. There are several particular ways I think it would be useful to apply such "adversarial control".

• You can use a GAN to produce a black-box system model from the observed behavior, which you use in the usual model-predictive way: use any standard optimization algorithm to compute a control strategy (in the sense of a sequence of planned control outputs) that maximizes your utility function (or, equivalently, minimizes your cost or loss function), by using the generator from the GAN to predict what would happen if each candidate strategy were followed. The generator normally needs to be stochastic, since there are always unobserved variables driving the behavior seen in the training set, and so it's possible to get estimates of uncertainty — at least by drawing several deviates from the generator and looking at their spread.

It's fairly straightforward to use this to assess the "risk" of the candidate strategy — you just look at the spread of objective-function evaluations, or possibly even the derivative of the objective function with respect to the hidden variables that you feed to the generator to make it act stochastic — but I feel like there's also some way to derive from this the "learning value" of the proposed strategy as an experiment. I think the story is that you first differentiate the objective function (over the whole distribution the generator can generate), or possibly its derivative with respect to those hidden variables, with respect to the parameters of your *generator*, to get a gradient that tells you which parameters of your generative model are most important to simulate accurately; and then you assess the learning value of the proposed control strategy by calculating how much you're likely to update those parameters with that strategy.

That is, you're looking for parameters of your generator which the outcome of this experiment would nudge in a way that makes a significant difference in your objective function in some scenarios, but ideally not the scenario you're facing at the moment (the safe experiment design problem). By dreaming of being chased by monsters (a scenario drawn from your generator in which a bad strategy results in you dying horribly) you learn which aspects of reality your generator needs to model better, and so what you should try to find out by taking actions in real life; for example, you may want to look under your bed, because it's a safe thing to do, but if there are monsters there, you will see them and can update your system model in a way that will greatly increase your utility.

This is not a metaphor; I'm proposing that the humans' neurology actually works in the way similar to what I'm describing above, though it doesn't use automatic differentiation and backprpagation, and that is why they literally look under their beds for monsters; or at least that this formalism is a good way to get similar kinds of intelligent behavior.

- You can use a GAN to optimize a control network built out of whatever components are inexpensive in your deployment context, such as transistors, resistors, capacitors, and diodes, or Mark Tilden's BEAM-robotics Nv neurons, or opamps, or FPGA LUTs and flip-flops, or links and kinematic pairs, by simulating different such candidate control networks in a wide variety of scenarios and scoring their performance on an objective function. The appeal of this approach is that it allows you to build a control system that can handle low-level control tasks with very low latency and low cost, while being itself controlled by a higher-level control system which provides them with some kind of time-varying set point. A feedback loop consisting of a few RF transistors and some passives has a potential latency in the nanoseconds rather than the milliseconds typical of hard-real-time software control loops, but it will necessarily be wildly nonlinear and have many unpredictable characteristics due to manufacturing variation, aging, and temperature.
- Applying the adversarial-control approach recursively, in order to accurately model your control network, you can build a generative adversarial model of your circuit components and of circuits built from them: the discriminator tries to distinguish data measured from real circuits processing real signals from simulations generated by the generator, while the generator tries to simulate the real circuit so faithfully that the discriminator can't tell the difference.
- In a further level of self-reference, if the discriminator is a recursive neural network or other dynamical system, we can give it another tool to beat the generator with: let it generate test signals to feed into the circuit and observe the response, thus exploring corners of the circuit's behavior that the generator hasn't yet succeeded in simulating. (In some cases you will have to optimize the discriminator not to generate signals that your simulations suggest will damage the circuit.)
- If both your control network and your generator are themselves differentiable, you can differentiate not just your objective system but the system's state vector with respect to *either* the initial system state vector *or* the hidden-variable inputs that make the generator stochastic, which has an established name that I forget. (Sometimes

people talk about things like "the latent space of faces", and though I think that's more a variational autoencoder kind of thing than a GAN kind of thing, that's the kind of hidden variables I'm talking about.) One potential benefit of this is that it allows you to make statements about the stability of your control-stabilized system in a way that you can't with standard MPC. Another is that it makes the kind of experiment design that I described in point #1 above amenable to gradient-driven optimization, because you can tell how to tweak your control network so that it will spontaneously engage in safe experiments.

In the case of time series, both the generator and the controller will generally need some history to start from rather than just a single observed state vector, because they need to infer the state of some variables in the system that are not directly observable. For example, if something has been warming up significantly even though you aren't applying heat, maybe there's an unobserved heat source in contact with it; the generator needs to simulate this situation for the purpose of evaluating strategies, and the controller needs to take it into account when formulating them, applying less heat than it would otherwise. And if you're planning a candidate toolpath or simulating its effects, you'll need to know at what height the tool touched off on the touch-off sensor, even if that was several minutes ago.

Controlling digital fabrication is a particularly important application because it enables the materialization of controllers, and it is particularly interesting in several ways. It includes parameters that vary over a wide range of timescales, which is especially challenging to simulate: a machine may wear out year by year; the air around a machine may be hotter in summer than in winter, causing parts to cool down slower; a tool may wear as it cuts, getting progressively shorter and rougher minute by minute; the temperature of a tool or hotend may increase second by second; and a tool that chips will suddenly cut differently. It normally takes into account many physical phenomena: vibrational modes of parts, rigidity of machines, temperatures, electric currents, side forces, measurement error, acceleration force, etc. And the objective function to optimize may shoot through the entire design process: for example, how can we position such-and-such a thing under such-and-such loads for the lowest manufacturing cost?

Topics

- Programming (p. 1141) (49 notes)
- Digital fabrication (p. 1149) (31 notes)
- Manufacturing (p. 1151) (29 notes)
- Frrickin' lasers! (p. 1168) (12 notes)
- Control (cybernetics) (p. 1262) (4 notes)
- Artificial neural networks (p. 1307) (3 notes)
- Mathematical optimization (p. 1348) (2 notes)
- Gradient descent (p. 1364) (2 notes)
- Generative adversarial networks (GANs) (p. 1366) (2 notes)
- Dreaming (p. 1373) (2 notes)

Constant weight dithering

Kragen Javier Sitaker, 02021-10-28 (updated 02021-12-30)
(5 minutes)

Suppose you want to encode some digital data in a one-bit-deep (black-and-white, no grey) image, but you want the image to also depict something; to make this simpler, let's say that what it depicts is independent of the data encoded. One way to do this is with M-of-N or constant-weight codes.

Consider a 2×2-pixel area of this image; there are 16 possible patterns: one all black, four three black and one white, six two black and two white, four three white and one black, and one all white. This gives you five possible levels of brightness for this 2×2-pixel area, but three of these levels have multiple possible ways to achieve them. With conventional dithering, you use the choice among them to improve the high-frequency reproduction of the image — the precise locations of edges and things like that.

Suppose, instead, that you dither the image down to a 5-level grayscale image, then replace each pixel of the 5-level grayscale with one of these 2×2 blocks, with the appropriate brightness. You can use a 1-of-4 code, a 2-of-4 code, or a 3-of-4 code (the complement of the 1-of-4 code) within these blocks to encode arbitrary data. The 2-of-4 code gives you lg 6 = 2.58 bits per block, while the other two give you 2 bits per block. If the image's contrast is destretched enough to put essentially all of the dithered 5-level pixels within that range of grays, you might get about 2.2 bits per 4-pixel block, which is 0.55 bits per pixel. That is, a slight majority of the data in the final image is devoted to encoding your chosen data. (By histogram equalization you can arrange to distribute the image brightness across the available levels in almost any conceivable nontrivial way, though possibly at the cost of beauty or comprehensibility.)

If instead of 2×2 blocks we use 3×3 blocks, then instead of 1 4 6 4 1 possibilities at the different gray levels, we have 1 9 36 84 126 136 84 36 9 1, allowing us to encode respectively 0, 3.17, 5.17, 6.39, 6.98, 6.98, 6.39, 5.17, 3.17, and 0 bits, averaging 4.34 bits per 9-pixel block; if we exclude the ends, it's 5.43 bits, or 0.60 bits per pixel, 10% better than the 2×2 case with less compromise of the contrast. With a little more compromise on contrast, you can probably push that past 6 bits per 9×9, 0.67 bits per pixel. The tradeoff, of course, is that you've lost more than half of the spatial resolution previously devoted to encoding the carrier image, in the sense that you're encoding less than half as many 9-level grayscale pixels as you were 5-level grayscale pixels.

This can be straightforwardly extended to the case of non-monochrome images. Instead of 2 possibilities per pixel, you might have 4 (RGB), 5 (CMYK), 8 (superposable RGB), 16 (superposable CMYK), or more, so each of the constant-weight codes you're using to encode the data is no longer a *binary* constant-weight code, and the reduced-palette image you're encoding from is no longer grayscale.

In most practical uses of this method, you would need error correction coding.

It's possible that this method is already covertly in use for printer tracking dots, with the justification being the prevention of counterfeiting paper money. Other possible uses include:

• Provenance information, such as EXIF data, in a photograph, which ought to only be done by the voluntary choice of the photographer;
• A machine-readable circuit model in a circuit schematic;
• A machine-readable version of a program in a printed program listing;
• Including a machine-readable version of a program in the program's *output*;
• For example, including the equations and parameters used to generate a fractal image in the rendered fractal;
• Steganographic communication between people seeking privacy;
• Including a machine-readable data table in a data plot;
• Privacy-invading watermarking uses similar to the tracking-dots approach mentioned above, allowing the producer of many versions of an image to track down the first step of path by which a particular image made it to a particular recipient;
• Watermarks claiming credit or copyright, like the easter egg in Commodore PET Basic Version 2 which would display "MICROSOFT!" when you typed "WAIT6502,1" or the Stolen From Apple logo in the Macintosh firmware following the Franklin Ace lawsuit.

Topics

• Programming (p. 1141) (49 notes)
• Graphics (p. 1177) (10 notes)
• Security (p. 1224) (5 notes)
• Encoding (p. 1256) (4 notes)
• Communication (p. 1264) (4 notes)
• Barcodes (p. 1385) (2 notes)
• Dithering

Hashing text with recursive shingling to find duplication efficiently

Kragen Javier Sitaker, 02021-10-30 (updated 02021-12-30) (6 minutes)

Suppose you divide a text into consecutive four-byte windows. If the text is not a multiple of four bytes long, one of the windows will not be full; traditionally we pad at the end, but we can also pad at the beginning. There are four ways to do such sequence alignment. If we hash the text in each of these windows into some alphabet, perhaps one large enough that hash collisions are improbable, each of these window alignments converts the text into a text from a larger alphabet with one fourth the length.

At this point we have converted the original N-byte text into four N/4-letter texts; call them "first-level summaries". Suppose that we choose only three of these, for a total of 3N/4 letters. Repeating the process on each of the 3 first-level summaries gives us 9 second-level summaries, each 1/16 the size of the original text (though in a larger alphabet) by repeating this until we are reduced to a single letter, we end up with (almost) 3N hashes for different parts of the text, each computed over four letters, so this process takes linear time.

Suppose the text consists of two copies of some motif concatenated. Then the hashes in the first level will be mostly the same. If the original motif is a multiple of 4 bytes, *all* the hashes in the first-level summaries will be the same, except those overlapping the boundary; but if not, then the second copy of the motif will be byte-misaligned. Suppose that the misalignment is 1; then, the hashes in first-level summary #1 of the first copy of the motif will be found a second time in first-level summary #2 of the second copy, those of first-level summary #2 of the first copy will be found in first-level summary #3 of the second copy, while the hashes in first-level summary #1 of the second copy and first-level summary #3 of the first copy will not be found again. The other possible misalignments, 2 and 3, have similar properties: two thirds of the hashes in the first-level summaries will occur twice.

In higher-level summaries we have the same sort of property, that a repeated motif results in two or three repeated sequences of hashes in the summaries of all levels small enough for the plaintext of the motif to be entirely contained inside a single hash at the next level up.

By starting at the topmost level summary and working down, we can efficiently detect duplicate text of any length anywhere in a corpus — in linear time, if we treat hash-table probing as constant time, or linearithmic time in a more realistic scenario. This provides an efficient solution to the basic version of the sequence alignment problem, the rsync problem (without using sliding hashes), and, I think, the diff-with-rearrangement problem.

A given 4-byte substring is not guaranteed to be covered by a hash

in the first-level summary, but of the two 4-byte substrings of a given 5-byte substring, one or both will be. Similarly, in a 17-byte substring, one or both of its two 16-byte substrings will be covered by a 4-letter substring in the first-level summaries, which may or may not have a hash in the second-level summary, but a 5-letter substring in the first-level summaries is guaranteed to have one, and every 21-letter substring of the original string is thus guaranteed to contain at least one second-level hash. So the maximal size of an unrepresented substring in a given summary level proceeds by this logic of $f(i) = 4f(i-1) + 1$: 5, 21, 85, 341, 1365, 5461, 21845, 87381, 349525, 1398101, etc.

(There might be some way to stagger the skipping across summaries to get this series to increase a little slower.)

I think that, by this scheme, you would add 15 levels of summary to a gibibyte of text, as follows:

- 1,073,741,824 bytes of text;
- 805,306,368 hashes, 6,442,450,944 bytes in 8-byte hashes, each covering 4 bytes;
- 603,979,776 hashes, 4,831,838,208 bytes in 8-byte hashes, each covering 16 bytes;
- 452,984,832 hashes, 3,623,878,656 bytes in 8-byte hashes, each covering 64 bytes;
- 339,738,624 hashes, 2,717,908,992 bytes in 8-byte hashes, each covering 256 bytes;
- 254,803,968 hashes, 2,038,431,744 bytes in 8-byte hashes, each covering 1024 bytes;
- 191,102,976 hashes, 1,528,823,808 bytes in 8-byte hashes, each covering 4096 bytes;
- 143,327,232 hashes, 1,146,617,856 bytes in 8-byte hashes, each covering 16384 bytes;
- 107,495,424 hashes, 859,963,392 bytes in 8-byte hashes, each covering 65536 bytes;
- 80,621,568 hashes, 644,972,544 bytes in 8-byte hashes, each covering 262,144 bytes;
- 60,466,176 hashes, 483,729,408 bytes in 8-byte hashes, each covering 1,048,576 bytes;
- 45,349,632 hashes, 362,797,056 bytes in 8-byte hashes, each covering 4,194,304 bytes;
- 34,012,224 hashes, 272,097,792 bytes in 8-byte hashes, each covering 16,777,216 bytes;
- 25,509,168 hashes, 204,073,344 bytes in 8-byte hashes, each covering 67,108,864 bytes;
- 19,131,876 hashes, 153,055,008 bytes in 8-byte hashes, each covering 268,435,456 bytes.

(Actually, I think I'm slightly overestimating the higher levels because I'm omitting the hashes that would be hashing entirely missing data off the end of the file.)

This is 26,809,069,296 bytes, about 25 gibibytes in all, the original gibibyte plus almost 24 gibibytes of summaries. If you are only interested in finding large coincidences, more than 5, 21, 85, 341, or 1365 bytes, you can discard the first few levels of summaries, saving you most of those gibibytes.

The hash function you use needs to be reasonably good to avoid false positives. If you're willing to accept a small false positive rate, you can use a smaller hash, such as 4 bytes. Collisions only matter within a summary level, so it might be reasonable to use smaller hashes at higher levels.

Topics

• Performance (p. 1155) (22 notes)
• Algorithms (p. 1163) (14 notes)
• Hashing (p. 1293) (3 notes)

My Heathkit H8

Kragen Javier Sitaker, 02021-11-03 (updated 02021-12-30)
(2 minutes)

(Comment on https://hackaday.com/2021/10/27/vcf-east-2021-preserving-heathkits-8-bit-computers/.)

My H8 was hooked up to an H19 terminal and H17 floppy drives, so the experience was similar to using a TRS-80 or an IBM PC. On HDOS there were Microsoft BASIC, the less capable Benton Harbor BASIC, lots of video games using the semigraphics character set (and the H8's built-in speaker), text editors called PIE and SCRIBE, text formatters similar to nroff called TEXT and RUNOFF, and text files traded on disk at user-group meetings, including porn. For electronic engineering, you could do a lot of calculation and simulation in BASIC that would have been really hairy on a programmable calculator, although its ability to plot graphs on the screen was pretty limited.

I was a kid, so my favorite use was the games (and the porn). My favorite games included Munchkin (a Pac-Man clone), Invaders, SEABATTL, A Remarkable Experience (a puzzle-solving text adventure similar to ADVENT or Zork), CASTLE, and Star Trek, where you would fly around shooting Klingons with your phasers and photon torpedos and try not to fly the Enterprise into a star. Other games I played included Lunar Lander, Hammurabi, Towers of Hanoi, Reversi, chess, and a significantly enhanced non-turn-based version of the "robots" game in the bsdgames package.

Under CP/M there was WordStar, a mostly WYSIWYG word processor with only a few nroff-style dot commands left, and SuperCalc, a spreadsheet. There was a huffman-coding utility called SQ[ueeze]. Later I installed MODEM7 under CP/M and dialed up BBSes with a modem, and I could upload and download files with XMODEM. Sometimes, though, I couldn't download a file unless the BBS sysop was kind enough to break it up into pieces that were smaller than the (100KiB) floppy disks.

Despite the availability of PILOT, effectively everything that wasn't written in BASIC was written in assembly language; both HDOS and CP/M came with an assembler, a linker, and a library facility to pull only the library routines you needed out of a library. Later on BDS C and Turbo Pascal brought high-level languages to the platform, but they were too late and not competitive in performance with assembly language or beginner-friendliness with BASIC. Unfortunately, I never learned how to program in assembly.

Topics

• History (p. 1153) (24 notes)
• BASIC (p. 1303) (3 notes)

Orthogonal rational vectors

Kragen Javier Sitaker, 02021-11-04 (updated 02021-12-30)
(4 minutes)

Consider measuring four values x0, x1, x2, and x3 (for any value of "four"). One way to do this is to take a separate measurement of each value, but often this is difficult to do, for example because your measuring tools have unknown offsets. So sometimes you might prefer to measure four linear combinations of x0, x1, x2, and x3; then, any fifth measurement will allow you to determine such an unknown offset.

Popular bases for this include the Fourier basis, in which you measure the average value and the dot products with three sine waves. But that has the awkward problem that in many cases the coefficients are transcendental; it also has the property that some of the coefficients may be 0, as in the case of four values, where I think the Fourier basis is x0 + x1 + x2 + x3, x0 - x2, x1 - x3, and x0 - x1 + x2 - x3. The Hadamard-Walsh basis is another alternative, in which all the coefficients are either 0 or 1, but these are also sparse, in the sense that they don't depend on all the values; or, alternatively, you can use -1 and +1.

In some sense the simplest kind of number that could give you an orthonormal basis for this kind of thing is rational numbers. It occurred to me that with one or more Pythagorean triples, you can construct an arbitrary number of orthonormal bases with rational coefficients. Consider the triple 3-4-5. This gives us the orthonormal basis [[3/5, 4/5], [4/5, -3/5]]. We can use this to rotate an arbitrary rational orthonormal basis into another rational orthonormal basis; for example, starting with the identity-matrix basis:

```
[ 1  0  0  0 ]   [ 3/5  0  4/5  0 ]   [ 3/5  0  4/5  0 ]
[ 0  1  0  0 ]   [  0   1   0   0 ]   [  0   1   0   0 ]
[ 0  0  1  0 ]   [ 4/5  0 -3/5  0 ] = [ 4/5  0 -3/5  0 ]
[ 0  0  0  1 ]   [  0   0   0   1 ]   [  0   0   0   1 ]
```

By iterating this sort of rotation we can get an infinite number of orthonormal bases, and after a few such rotations our matrix is dense. For example,

```
[[ -900, -1200,  2500, -1125],
 [ 1293,  -776,  1200,  2460],
 [-2376, -1293,  -900,  1280],
 [ 1280, -2460, -1125,  -900]]/3125
```

or equivalently

```
⎡ -36      -48                     ⎤
| ─────    ─────       4/5    -9/25|
| 125      125                     |
|                                  |
| 1293    -776       48      492   |
```

```
|  ─────    ──────     ────     ────  |
| 3125    3125    125     625 |
|                             |
|-2376   -1293   -36      256 |
|───────  ───────  ─────    ────  |
| 3125    3125   125      625 |
|                             |
| 256    -492            -36  |
| ───     ──────     -9/25 ─────  |
⌊ 625     625             125  ⌋
```

is a dense rational orthonormal matrix representing five such rotations. So, returning to our original measurement problem, we could measure -900 x0 - 1200 x1 + 2500 x2 - 1125 x3, 1293 x0 - 776 x1 + 1200 x2 + 2460 x3, and so on, and by multiplying by the transpose of this matrix and dividing by 9765625 (3125 squared), we should get the original values of x0, x1, etc. For example, for [11, 5, 8, -2] we get [254/125, 15023/3125, -42361/3125, -1084/625] when we divide by 3125 once, and multiplying back through the transposed original matrix does indeed give us back [11, 5, 8, -2].

Even though I don't know how to define norms in a prime field, you can convert a rational matrix like this into its equivalent in some prime field, and do the same thing in that field, and you'll still get back the original numbers.

A thing that's bothering me is that it seems like you ought to be able to do something like this that gives you back *more* values than you put in, and somehow use that for erasure coding, or N-of-M secret sharing, or for noise reduction in noisy measurements. But this requires giving up orthogonality; you can't have five orthogonal vectors in a four-dimensional space, so you can't just transpose the matrix, you have to use some other approach to solving it, like calculating the pseudoinverse.

Topics

• Math (p. 1173) (11 notes)

Thread rolling roller screw

Kragen Javier Sitaker, 02021-11-04 (updated 02021-12-30) (1 minute)

A roller screw can have its rollers either move with the shaft through the middle, in which case only a short length of the shaft needs to be threaded, but the entire outer tube needs to be threaded on the inside; or it can have the rollers move with the nut, in which case the rollers will attempt to engage a thread along the entire length of the shaft.

It's possible to arrange for the advance of the roller screw to be arbitrarily small in a differential fashion, potentially a tiny fraction of the thread pitch per rotation. This suggests that, by using tapered, hardened rollers, it should be possible to use a roller screw to roll a thread onto a previously unthreaded shaft, with an extremely manageable low torque, much like a hand thread-cutting die, but rolling a thread on the shaft rather than cutting one. The threads on the nut would be tapered to compensate.

By a similar principle, it should be possible to roll threads onto the inside of a hole by using a tapered shaft with tapered thread rollers rolling around it.

Topics

- Contrivances (p. 1143) (45 notes)
- Mechanical (p. 1159) (17 notes)
- Hand tools (p. 1197) (7 notes)
- Roller screws (p. 1275) (3 notes)

Viscoelastic probing

Kragen Javier Sitaker, 02021-11-04 (updated 02021-12-30)
(2 minutes)

If you have a probe pressing against a material, its force and position are functions of time. If you're using a spring attached to an actuator, you can try to jam it in harder or less hard, which will tend to increase or decrease the force, respectively; but the material can then yield. Viscoelastic materials will yield to different degrees at different time scales, and every material is somewhat viscoelastic; this gives us a complex spectrum in which the real part at a given frequency is the elasticity (or resonance) and the complex part is a frictional loss.

If you apply a Heaviside step function to the probe, in theory this contains all frequencies and so you can determine the material's entire viscoelasticity spectrum from its response to the step function. In fact, though, your step function is going to be bandlimited, and the material's response at low frequencies may be lost in the noise. By applying a *sequence* of such step functions, sometimes in opposite directions, at random times, you can get more data points, which will allow you to estimate not only the viscoelastic spectrum of the object but also the frequency and Q of its vibrational modes.

A piezoelectric actuator can straightforwardly produce frequency components up to a megahertz or so, and a strain gauge or piezoelectric force sensor can measure its special mix of force and displacement at similar speeds.

You can use the same approach for dielectric spectroscopy of time-dependent permittivity and magnetic spectroscopy of time-dependent permeability.

Topics

- Electronics (p. 1145) (39 notes)
- Sensors (p. 1191) (8 notes)
- Piezoelectrics (p. 1340) (2 notes)

An aluminum pencil for marking iron?

Kragen Javier Sitaker, 02021-11-06 (updated 02021-12-30)
(2 minutes)

Aluminum forms a brittle intermetallic with iron (as well as some other elements like nickel and cobalt), which I think is the cause of steel screws seizing in aluminum when unlubricated. Perhaps you can exploit this property to mark visible lines on iron and steel using an aluminum rod. I think there are three fundamental obstacles: surface preparation, activation energy, and aluminum cohesion.

If the aluminum is in contact with grease or iron oxide rather than raw iron, it won't be able to react. I think that if the surface is *reasonably* clean to begin with, it should be possible to surround the aluminum "pencil lead" with a friable abrasive "wood" to adsorb the grease and grind through the oxide, maybe a mixture of things like talc, kaolin, aluminum oxide, and binders.

The activation energy problem is that the aluminum needs to be at a relatively high temperature to get this reaction. Since it happens by accident with screws, I think this is a feasible thing to achieve by hand, but it might help if the aluminum is in smallish particles, a foam or sponge, or thin aluminum-foil-like layers, so it won't conduct the heat away from the reaction surface. Maybe thin fibers of something very hard inside the aluminum would also help by scratching hot particles off the iron.

The aluminum cohesion problem is that we want the newly formed intermetallic to stick to the steel rather than the aluminum. For this we want to keep the "pencil lead" from having too much structural strength of its own; if the aluminum is in the form of, again, small particles, foils, or foam, that should help with this. This is somewhat in conflict with the desire to apply a great deal of pressure to the surface in order to overcome the activation-energy barrier.

Topics

- Materials (p. 1138) (59 notes)
- Contrivances (p. 1143) (45 notes)
- Aluminum (p. 1180) (10 notes)
- Hand tools (p. 1197) (7 notes)
- Steel (p. 1222) (5 notes)

Embedding runnable code in text paragraphs for numerical modeling

Kragen Javier Sitaker, 02021-11-06 (updated 02021-12-30)
(6 minutes)

Watching a demo of Keras in TensorFlow in IPython, I thought a Python comment was a Markdown header. And it occurred to me: why *not* have formatted text comments in your code? Or code in your text document?

Literate Haskell works like this: you write text like normal, but mark your code with a leading >:

```
> 3 + 4
```

You could imagine a sort of word processor that works this way, automatically displaying the results of your code within the document, like a spreadsheet or like Darius Bacon's Halp. You could even put bits of code in the middle of a paragraph, displaying the code, its result, or both at your whim. This works best with non-procedural programming paradigms like logic programming and pure functional programming.

Programs normally contain a mix of commentary with executable code. Traditional programming languages make the code primary, relegating the commentary to a secondary role, while literate Haskell reverses this. Python doctest docstrings do something similar: the docstring can contain code, but it is specially marked with >>> or .... IPython and ObservableHQ notebooks follow a middle way, treating either code or commentary as primary depending on which kind of cell you're in. But I think that normally the commentary should be primary and the code secondary.

Ideally when your cursor was near something easily identifiable as a quantity, like `3`, `3.17 bits`, `23 AWG`, `45 psi`, or `2.4 GHz`, it would become available for naming, referencing, and calculations; maybe you'd have a sidebar that showed you the names, definitions, and current values of nearby bits of code. Maybe as part of this sidebar you'd have a stack of active data, so that by typing alt-+, alt--, alt-/ or alt-* you can apply arithmetic operations on the top item on the stack, or by typing some other magic key like alt-X you can invoke some named operation on it, such as getting some property.

So you might type "23 AWG wire has a cross-sectional area of ", and then, looking at the object in the sidebar which represents a cylinder with a diameter of 0.57 mm and an unknown length, type alt-X and begin typing the name of the "base area" property, releasing Alt when you've typed enough to disambiguate; but on seeing that 2.58e-7 m² appears in your document but is not a very useful way to display it, type alt-A or something to get a menu of likely units, then select mm² (rather than, say, `argentina_area` or 分地), thus changing the display of the value to 0.258 mm².

Then you can type " and, if copper, a resistance of ". Maybe the

system recognizes that “copper” is the name of an entity which has properties, so it shows up in the sidebar, or maybe you have to open a search box for it; either way, you get its resistivity of 16.78 nΩ·m (at 20 °C) or, according to units(1), copperconductivity of 58 siemens m / mm^2 (the Wikipedia value above works out to 59.59). At this point you can divide the resistivity by the surface area, or multiply the conductivity and then take the reciprocal, with a couple of keystrokes, to get the resistance: .0668 kg m / A^2 s^3, which is to say, .0668 Ω/m. (Ideally that would be the default way it was presented.)

At this point you can go back and drag your mouse over the “23 AWG” to change the wire gauge, or click on a dropdown arrow next to “copper” to select other metals.

Of course, many wire resistance calculators already exist, and you can do this problem in units(1) as well:

```
$ units
Currency exchange rates from FloatRates (USD base) on 2019-05-31
3460 units, 109 prefixes, 109 nonlinear units

You have: 1/circlearea(awg(23)/2) copperconductivity
You want: ohms/m
    * 0.066785595
    / 14.973289
You have:
```

But it typically requires a fair bit of blundering around to get there, and in fact I solved the problem wrong the first time.

(Perhaps the “focused item” in the sidebar should have an expanded view that tells you what the result of each property and possible operation would be.)

A different approach to solving this would be to instantiate a cylinder by binding the “material” property of the 23 AWG cylinder to “copper” and its length to 1 meter; this defines its mass, volume, surface area, electrical resistance, thermal resistance, etc. Then you could just pop in the electrical-resistance property.

Once you’ve reached this point, you can name the resistivity result, the wire metal, and the wire size as properties of the current document, and incorporate some more text: “If iron, ” and then construct `this{metal=iron}.resistivity`, the resistivity property of an instance of the current document with the metal changed to be iron. This is just bog-standard Bicicleta.

You might also want to do things like plot the resistivity against AWG or against, say, mass per meter, perhaps for different metals. You might want to include a table of the current calculations for, say, a variety of different materials. You might want to apply gradient descent to maximize some property.

Topics

• Human-computer interaction (p. 1156) (22 notes)

- End user programming (p. 1217) (6 notes)
- Numerical modeling (p. 1229) (5 notes)
- Editors (p. 1257) (4 notes)
- Bicicleta (p. 1384) (2 notes)

Paeth prediction and vector quantization

Kragen Javier Sitaker, 02021-11-06 (updated 02021-12-30) (1 minute)

Suppose you want constant frame rate video over a bandwidth-limited channel with very simple encoding and decoding. Paeth's predictor predicts that each new pixel will be equal to one of the three pixels N, NW, and W of it, based on an estimate of the gradient from those three pixels, and then you encode, normally, a precise residual from that prediction. This approach can be straightforwardly extended to use a three-dimensional prediction incorporating the previous frame as well.

However, instead of transmitting a precise residual, you could transmit a quantized approximate residual. For example, you could transmit a choice of 0, ±1, ±4, ±16, or ±64, which is 9 choices, 3.17 bits per pixel per color channel. If an image holds still for a little while, it'll settle down to the correct image fairly quickly.

However, this still sucks, because the compression ratio is less than a factor of 3. You can do a little better by transforming to YUV (16-bit maybe) and subsampling the chroma, but to do a lot better by some such approach, you need to not transmit any bits for most pixels.

Topics

- Graphics (p. 1177) (10 notes)
- Compression (p. 1263) (4 notes)
- The Paeth predictor (p. 1345) (2 notes)

Wire brush microscope

Kragen Javier Sitaker, 02021-11-06 (updated 02021-12-30) (1 minute)

If you want to measure the shape of a metal surface, you can gently touch it with a bunch of sharp springs, insulated from one another and constantly separately tested for electrical continuity with the metal. By moving this springy brush around, you can measure when each wire enters and leaves contact with the surface; by only barely touching the surface, you keep the stresses on the spring tips low enough to avoid plastic deformation. Using springs with a slight curve or coil to them, rather than straight bristles, reduces the stress for a given strain, protecting the tips from being blunted.

It's important that the bristles not bend plastically or creep, so it is advisable to make them out of a brittle substance with a high melting point, such as tungsten, or such as aluminum oxide with thin metal plating. Sharp points can be achieved by growing crystals with a naturally acicular habit, like mullite or some phosphates of calcium, which will normally have atomically-sharp points which will be blunted slightly by the metallic plating, or by electrolytic sharpening of metallic points.

Topics

- Contrivances (p. 1143) (45 notes)
- Precision (p. 1183) (9 notes)
- Sensors (p. 1191) (8 notes)
- Scanning probe microscopy (p. 1242) (4 notes)

New nuclear power in the People's Republic of China

Kragen Javier Sitaker, 02021-11-09 (updated 02021-12-30)
(2 minutes)

Comment on https://news.ycombinator.com/item?id=29151741.

"The cost of China's new nuclear ambition has been estimated at US$440 billion." But it doesn't say how much power that is, just that it's 150 new reactors, so it's about US$3B per reactor. https://archive.md/A5B6S shows that the article didn't say how much power it was 9 hours ago either. So, is this 15 GW, 150 GW, or 1500 GW?

PV modules are about US$0.2/Wp, so if it were PV panels, US$440B would buy them 2.2 terawatts peak. At an average capacity factor of 20% (though, as pfdietz points out, most new utility-scale solar has single-axis tracking, which pushes it to 30%) that would be 440 GW, but China's historical PV capacity factor has been terrible, more like 12% IIRC (maybe due to a Chinese version of the irrational misregulation robomartin documents in California in https://news.ycombinator.com/item?id=29155094). 12% would make it more like 260 GW. But a PV power plant includes things that aren't panels; balance of plant (inverter, wiring, grid connection, monitoring, mounting, security) is typically roughly equal to the module cost. So it would be more like 130 GW. (Total costs of utility-scale solar in the US are about twice that in China https://www.irena.org/-/media/Files/IRENA/Agency/Articles/2017/Jul/Bonn-Uni-Lecture--True-costs-of-renewables.pdf?la=en&hash=B7DD1720455A1ED042094C007D8B8C74F274AAFC at about US$0.89/Wp according to pfdietz https://news.ycombinator.com/item?id=29155644.

In the US, nuclear plants cost about US$8/We. If China's program was at the same cost, it would provide 55 GW. If it was closer to the cost of US nuclear plants in the 01970s about US$1/We (and if the US$440B number is correct), it would provide 440 GW.

The threads at https://birdsite.xanny.family/pretentiouswhat/status/1293961095892279296 and https://birdsite.xanny.family/pretentiouswhat/status/1318838054891573249#m provide some more context, suggesting that the "HPR1000, aka Hualong One 华龙一号", is the reactor being used at these 150 sites. https://www.sciencedirect.com/science/article/pii/S2095809916301515 says the HPR1000 is 3.050 GW thermal, 1.070 GW electric, net. (See also https://www.ukhpr1000.co.uk/the-uk-hpr1000-technology/hpr1000-design/ and https://en.wikipedia.org/wiki/Hualong_One.) So 150 of them would produce 161 GW electric, which (if that's US$440B) would put the cost around US$3/We, about twice the cost of the same generation capacity via PV with single-axis tracking, not including any cost of storage. But maybe that's 150 *power plants*, each

with nuclear reactors, not 150 *nuclear reactors*?

Topics

- Energy (p. 1170) (12 notes)
- The future (p. 1220) (5 notes)
- China (p. 1379) (2 notes)
- Nuclear

Ivan Miranda's snap-pin fasteners and similar snaps

Kragen Javier Sitaker, 02021-11-11 (updated 02021-12-30) (3 minutes)

For his "mini tank" video, Ivan Miranda developed a very neat snap fastener. Adjacent segments of his tank treads are held together by a pin that runs through a series of holes alternating between tabs on the two segments, as is usual for a pin hinge. The surprising feature is how the pin is prevented from sliding out the end of the hole: it's flexible, and the end of the hole is curved, so you create a matching curve in the pin as you begin to insert it into the hole, then move the curve along the pin to finish inserting it.

So far this is just a spring-loaded friction connection, like a screw fastener or a nail. The nifty bit is that there's a notch cut into the outside of the curve for the end of the pin to snap into, once you insert it far enough. At this point the pin is pressed up against the notch at one end and the end of the hole at the other. So you have a snap fastener with a significant energy barrier to overcome in order to disconnect it.

In order to make the pin removable, Miranda includes another hole at right angles to the pin at the end of the notch, through which you can insert a rod to bend the pin back into the curve, and a smaller hole at the other end of the pin's hole through which you can press the pin back out.

This seems like an astoundingly neat idea, and I don't understand why I haven't seen it before. It entirely eliminates the need for screw fasteners in such loose-fit cases, while being immensely more vibration-resistant than screws. If the end of the notch (or the pin) is slightly slanted, the bending force of the pin will produce a preloaded compression force on the pin.

It isn't necessary for the pin to be entirely enclosed; for example, the tusk of a tusk tenon could be used as such a "pin" by merely sliding in a curved groove, or by pressing against two dowels protruding from the surface it slides against.

It's not obvious how to do a precisely analogous thing in a 2-D cutting environment, since there's no way to curve the holes through which a fastener slides. But if the fastener instead expands after sliding through two holes, or rather slots, we have a standard spring-hook fastener, which can also be built onto the edge of a panel to slide through a single other slot rather than through two separate slots.

Topics

- Contrivances (p. 1143) (45 notes)
- Flexures (p. 1232) (5 notes)
- Fasteners (p. 1297) (3 notes)
- Snaps (p. 1325) (2 notes)

Rendering 3-D graphics with PINNs and GANs?

Kragen Javier Sitaker, 02021-11-11 (updated 02021-12-30) (10 minutes)

(This is pretty much talking out of my ass since I've never done anything with neural networks and almost nothing with automatic differentiation and gradient descent.)

Physics-informed neural networks (PINNs) are an interesting approach to numerical solution of partial differential equations: you train a neural network to map (x, y) or (x, y, z, t) values to the value of the PDE solution. The training procedure involves running some sample points through your candidate network and calculating the derivatives of the output with respect to the input coordinates, thus giving the derivatives that you're trying to impose conditions on, and then calculating a loss based on how far the conditions are from being true. You typically need to make sure you have a number of points on your boundary in your training set in order to sample the boundary conditions.

A potential question here is how to decide which points to pick, because it may be the case that it's easy to get a solution that works correctly most places but is way off in a few crucial places. A solution that seems promising to me is to train a GAN ("generative adversarial network") to generate (x, y, z, t) tuples from some random number; the GAN is optimized to find tuples that produce a large loss for the PINN.

An interesting thing here is that, unlike more typical ways to numerically solve PDEs, there's no sample grid. The trained network represents the solution in a fully continuous fashion; you feed it any arbitrary (x, y) pair and it tells you what it thinks the value is at that point.

It occurs to me that you ought to be able to use the same approach to ray-trace a scene or film: train a network to map an (x, y, t) triple to an (r, g, b) triple in such a way as to minimize the error from some "ground truth" raytracing. You feed the same (x, y, t) tuple into a real raytracer written in the normal way to get the "ground truth" pixel; by applying automatic differentiation to the raytracer you can get the color gradient and movement at the sample, and by applying it twice we can get a Wronskian (?) that tells us how these gradients are varying. Then we can compare these results to the corresponding results from the neural network to compute the loss.

By training a GAN to generate difficult coordinates we can focus our optimization efforts on the places in the image which are particularly hard to approximate well, or at any rate particularly poorly approximated so far.

You might get better results by training a couple of stages of the raytracing network separately: for example, one stage that maps (x, y, t) tuples to (x, y, z, t) tuples where the ray intersected something, then a second stage that transforms these tuples into something like (x, y, z,

u, v, oid, t), and then a third stage that transforms that into the actual color. The benefit here is that you can use the traditional raytracer to train these intermediate tuples.

It might be possible to solve the rendering equation spatially by this method as well, deriving a neural network to approximate the light field: at any given point in space, looking in a particular direction, you see a particular color. In free space this color is the same that you would see if you moved in that direction; on a diffuse surface, looking into the surface, you see the color at the surface illuminated by the color you'd see integrated over all possible viewing directions; etc.

For numerical integration, maybe you could train a neural network (or other universal approximator, such as a spline) to approximate the indefinite integral of the function you want to integrate, generating random (or adversarially generated) points at which to compare the derivative of your approximation with the original function to compute your loss. It's hard to imagine how such an approach could ever be cheaper than just doing Gaussian quadrature in one dimension, but maybe if you have multiple independent variables, or if the limits of integration or a parameter of the function vary?

Another way to apply the PINN idea to rendering is to sample some pixels from a "real" raytracer, either the conventionally implemented raytracer or a universal approximator as described above, and then try to extrapolate the rest of the image from those pixels, in the same way that a PINN extrapolates the rest of the field from its boundary conditions. That is, you train an image-generating network to generate a visually plausible image that has the correct values at the sampled pixels, computing its loss from the error at the sampled pixels and a canned GAN discriminator network (probably a convnet) that judges visual plausibility. A second adversarial network can be used to decide which pixels to sample, looking for pixels with a large error, since you can sample more "test set" pixels once your image-generating network is trained.

This might be faster if you start with an image-generating network that already generates visually plausible images.

Normally you train a PINN simultaneously to satisfy both its constitutive PDEs (which in the above case are replaced with a discriminator) and its boundary conditions. You might be able to get a speedup on this by starting with a PINN pre-trained for the same PDEs and retraining it with new arbitrary boundary conditions, but a different approach is to include some samples from the boundary conditions among the PINN's inputs, along with (x, y[, z, t]). If this works, it gives you a PINN that solves an entire class of PDE problems instead of just one, allowing you to change the boundary conditions without retraining the network. To get a precise solution, you still might have to retrain the network.

Training a PINN to produce the SDF of a scene might be an interesting approach; the SDF is constrained to have value zero at objects' surfaces, negative inside them, positive outside, and to have a gradient with magnitude unity *almost everywhere*, in the sense that the cusps in the SDF (where the gradient has some other value) have measure zero, unless the surface geometry is fractal. So, if you're just

sampling at random, you'll find those cusps with probability zero.

A different way to use a PINN as an SDF is as a cheap-to-compute lower bound, training it to produce the tightest lower bound you can. Using interval arithmetic you can exhaustively evaluate the PINN and the real SDF over various parcels of space and find a bound on the worst case where the PINN drops below the true SDF; by adding this number to the PINN's output, you get a true lower bound. You can evaluate this cheap function for most SDF probes, only falling back to the true SDF (or maybe a small part of the true SDF) when the conservative approximation falls below 0.

A third approach to render images with a PINN is holographically: to look for solutions to a wave equation representing the propagation of waves through the scene. I think this can be a static field (i.e., a 3-dimensional problem rather than 4-dimensional) if the state variables at each point are complex rather than real, thus encoding not only amplitude but phase. For everyday macroscopic objects, diffraction effects normally only become noticeable at dramatically smaller scales than we normally look at (micron-scale, say), so the wavelength of the waves can usually be considerably longer than that of light. With a finite-element or sample-grid representation, this would reduce the computational effort enormously, but I'm not sure if it will matter for a PINN. If it *doesn't* matter much, that would be a *huge* advantage for computational holography, which unavoidably must use light's real wavelength.

Simulating polarization, for example for compound Fresnel-equation reflection, probably requires more than the two reals suggested above per point in the field; I don't know how many you need. Doing color probably requires doing three separate simulations.

It seems likely that three-dimensional or four-dimensional convolutional neural networks are likely to be useful for PINNs, but perhaps not as intermediate layers on their own; rather, you might need some intermediate layers that have convnets *in parallel with* conventional fully-connected layers.

The standard rendering problem is, given scene geometry (and materials, etc.), compute one or more 2-D images. From a certain point of view, vision is exactly the opposite problem: given one or more 2-D images, compute the scene geometry. Gradient descent and other generic optimization algorithms are thus applicable to turn any rendering algorithm into a vision algorithm, and they can additionally be guided by a neural network that is trained to produce geometries that are more probable (an approximate prior over world scenes).

Topics

- Graphics (p. 1177) (10 notes)
- Optics (p. 1209) (6 notes)
- Numerical modeling (p. 1229) (5 notes)
- Artificial neural networks (p. 1307) (3 notes)
- Generative adversarial networks (GANs) (p. 1366) (2 notes)

Aqueous scanning probe microscopy

Kragen Javier Sitaker, 02021-11-12 (updated 02021-12-30)
(7 minutes)

STMs and AFM can achieve deep subatomic resolution (10 pm is common), but STMs are limited to conductive materials, and in air they are limited to those that don't form a nonconductive oxide: mostly gold and graphite. Anything else requires not just vacuum but pain-in-the-ass UHV, worse even than an STM. And, as I understand it, their failure mode is to crash the probe if there's insulating crud on the surface, potentially destroying it.

Optical microscopes are normally limited to about 200'000 pm (a nominal wavelength of 600 nm divided by twice an oil-immersion NA of 1.5), four orders of magnitude worse. If you can see something at all in a visible-light optical microscope, it's probably at least 400 atoms across, which means it contains 64'000'000 atoms: seven orders of magnitude coarser than the atoms you can see with an STM. Ultraviolet microscopy can get partway into that region, but at a wavelength below 124'000 pm you run into the wall of vacuum ultraviolet, to which all gases and all liquids are opaque, so you're stuck around 40'000 pm, about 80 atoms across, 512'000 atoms or so per particle.

Can't we do anything to get into this region? Well, scanning near-field optical microscopy can help us with going under this limit; it can reach 20 nm (20'000 pm) with evanescent-wave illumination bringing it to life, but that's still more than three orders of magnitude away from STM/AFM resolution, 64'000 atoms or so. And it's limited to fluorescent samples, for which there are a number of other techniques available.

Here's a possible alternative for conductive samples, which includes anything we can sputter metals onto. If we have a *convex* conductive sample, we can immerse it in a fluid of high permittivity, such as water, glycerol, or propylene glycol, and set up an alternating low-voltage electrical field between the sample and some "reference electrode" in contact with the same liquid some distance away. The contour surfaces of constant voltage that form in the fluid can then be measured with a needle probe that is heavily isolated with a low-permittivity dielectric such as teflon, polyethylene, or beeswax, except at the tip. Assuming the resistivity of the sample is much lower than that of the fluid, one of these contour surfaces will be the surface of the sample itself, and others will be nearby; this should permit scanning the probe over the surface while maintaining a fixed distance, without crashing it, and without especial concern around the formation of insulating oxide films on the surface, etc.

The reason for the relative permittivities of the fluid and the probe insulation is that the potential gradient through the fluid (the electric field) should be fairly weak, while the potential gradient through the insulating sheath should be very strong indeed, so that the voltage we measure on the other end of the probe, somewhere outside the liquid,

which is the same as the voltage at the probe tip, is the same as the voltage that would be present if the probe were absent. This requires minimizing the capacitive coupling between the shaft of the probe and the liquid it passes through.

An electrolyte liquid, such as saline water, can be used instead of a pure dielectric, if its conductivity isn't too high and the voltage is low enough to avoid destructively large amounts of electrolysis or other reactions at the surface.

If we stick the probe inside a cavity in the sample surface, though, the potential gradient should entirely disappear. To correct this problem, we can use a second scanning probe as the reference electrode, so that we can insert it into the cavity at the same time. By shortening the distance, this method also greatly increases the potential gradient (which is to say, the electric field strength) we can apply, so that our microscopy resolution is limited not by the electrode potentials of potential electrolysis reagents but by the avalanche breakdown of the high-permittivity fluid.

Water's dielectric strength is sometimes cited as being around 70 MV/m, but such numbers strongly depend on the timescale; it can be enormously higher over short (subsecond) timescales, or much lower over long (multi-hour) timescales. Also, I think the Paschen minimum happens with avalanche breakdown in things that aren't gases as well, so the effective dielectric strength at submicron distances might be smaller. 70 MV/m is 70 mV/nm, and 70 mV is not a terribly challenging voltage to amplify (my stereo is faithfully amplifying submillivolt signals as I write this), so subnanometer resolution is probably attainable with this method.

At high frequencies high permittivity shades into conductivity; capacitors pass high frequencies, and if the dielectric is lossy enough, the current comes into phase with the voltage. The conventional value for the resistivity of deionized water is 18.2 megohm cm, which would give you about 200 teraohms (2e14) over a 1-nm channel with a square nanometer of cross-sectional area. Using a relative permittivity of 80, we get a capacitance of 7e-19 F for the same dimensions ($C = \varepsilon A/d = 80 \times 1\ \text{nm}^2 \times 8.85\text{e-}12\ \text{F/m} / 1\ \text{nm}$) and a reactance ($X = 1/2\pi fC$) which becomes smaller than the resistance at about 1 kHz and gets down to 200 megohms a bit above 1 GHz.

So on one hand the intuition that the water will polarize in such a way that it acts mostly capacitively is correct, but on the other hand detecting the current through such a tiny capacitance would be very challenging, if possible at all. Even at $1\mu\text{m}^2$ of tip area positioned $1\mu\text{m}$ away from the workpiece we only get 0.0007 pF.

However, I'm confident that if we load up the solution with enough ions, we'll be able to detect the voltage from the ionic current. Maybe a porous tip, or a dendritic tip, or one with lots of micro-slots cut into it, would enable a larger contact area with the ion-rich liquid. And you might have to use a lowish frequency to give the ions time to move around. The final distribution of ions will probably give a very nonlinear voltage distribution, but that should be okay if we're running the tip along a voltage contour.

Topics

- Contrivances (p. 1143) (45 notes)
- Physics (p. 1157) (18 notes)
- Sensors (p. 1191) (8 notes)
- Scanning probe microscopy (p. 1242) (4 notes)

Redundancy in self-replicating systems such as hundred-eyed chickens

Kragen Javier Sitaker, 02021-11-12 (updated 02021-12-30) (4 minutes)

A complex self-replicating system such as a hen contains a large number of somewhat unreliable subsystems, such as an intestine. If the intestine ruptures, the chicken will die without being able to produce another chicken; if the ovary dies, the hen will survive but will produce no further eggs. (Hens normally have only one functioning ovary.) As the number of such SPOFs grows, the chances that one of them will fail prior to self-replication also grows; for the total fertility rate to remain above the replacement threshold, the reliability of each of these systems must also increase.

The replication rate of hens is somewhat complicated to calculate: they start laying at 18–24 weeks of age, up to 250 eggs per year, maybe 75% fertile, requiring 21 days of incubation time. Commercial broiler operations kill their hens at one year of age, because fertility declines below 50% at a year, and egg operations at one or two years but otherwise chickens will typically live 3–7 years, laying less eggs each year: maybe 250, 200, 175, 150, 125, 110, 90. Half the eggs will be roosters. A hen can incubate 12–15 eggs at a time, and normally only does this ("goes broody") once a year, and never more than three times a year, so the figures below will assume artificial incubation.

A simple simulation (with a slightly simplified version of that model) reveals that this works out to something like a rate of increase of 3.3% per day, a doubling time of about 21 days. Naïvely, this would suggest that, disregarding infant mortality, as long as the hens' MTBF is more than 21 days, they would still produce replacements, but I don't think that's actually true; only 1 in 101 would live to 20 weeks, and on average would produce less than 21 offspring. I think the actual crossover point (without calculating it) is an MTBF of just over 32 days, at which point more than 1 in 21 hens survive to reproductive age.

If hens have a single SPOF, then, such as an ovary, it needs to have an MTBF of over 32 days to reach replacement fertility. If they have two SPOFs, such as an ovary and an intestine, one or both of them needs to have an MTBF of over 64 days. If they have 32 SPOFs, all but one of them need to have an MTBF of over 1024 days.

A little bit of redundancy can help somewhat here, but without regeneration, it has rapidly diminishing returns. Hens have two eyes, and usually die quickly if they go blind, so getting to 32 days of MTBF only requires each eye to have about 21 days of MTBF. If they had ten eyes, each eye would only need to have 11 days of MTBF. For Argos hens with 100 eyes each, to reach 32 days, each eye only needs to have 6.2 days of MTBF. Actually, though, the situation is much worse than that, because the number of surviving sighted hens drops off much faster than the usual exponential distribution;

with 6.2 days of MTBF for 100 eyes, less than 1 of every 1000 hens survives to 14 days, and less than one in a million to 21 days.

Alternation of generations and multiple genders are a significant subject here. In most animals the alternation of generations is subtle and easy to dismiss, but plants like ferns make it much more visible. Multiple "genders" of manufacturing plants might be specialized to produce particular parts or materials, which then must all be combined.

Topics

• Self replication (p. 1204) (6 notes)

DSLs for calculations on dates

Kragen Javier Sitaker, 02021-11-14 (updated 02021-12-30) (1 minute)

https://github.com/mvrozanti/dte is a DSL for doing calculations on dates, but it doesn't attack my biggest pain point, which is (as described in https://news.ycombinator.com/item?id=29136554) timezones. I want to know what local time is 5 PM US Eastern or 11 AM US Pacific, or how long it is from now until 8 PM UTC. Occasionally I want to know how long it is between 02021-11-15 21:55 and 02021-11-17 05:00, or convert to or from Unix timestamps, or to know how many days it is since 01983-05-21, or until Christmas. Occasionally I'd like to know the tzolkin date, the phase of the moon, or the time of sunset. Most of dte is useless to me; I *never* want to convert `23h:23` to `23:23:00`.

Timezone selection is the hardest thing to do in a DSLish way, because any conceivable way to do it or debug it noninteractively involves memorizing *something* about every timezone you want to use. You really want some kind of interface that gives you a list of the possibilities.

Topics

- Human-computer interaction (p. 1156) (22 notes)
- Domain-specific languages (DSLs) (p. 1260) (4 notes)
- Command-line interfaces (CLI) (p. 1378) (2 notes)

Some notes on reading parts of Reuleaux's engineering handbook

Kragen Javier Sitaker, 02021-11-17 (updated 02021-12-30)
(7 minutes)

Reading F. Reuleaux's *The Constructor*, in Henry Harrison Suplee's 01893 English translation of the German fourth edition, 340 pp. including the covers and title pages. Apparently the first edition dates from 01861! But Suplee says his 01893 translation is the first English edition.

I think it's reasonable to see *The Constructor* as mostly a 19th-century version of the *Machinery's Handbook*, the 01924 sixth edition of which has been in the public domain since January 1, 02020 , except that Reuleaux focuses almost exclusively on the machines to be built, not the processes of building them. (Before the *Handbook*, Oberg started out by writing a toolmaker's handbook for small tools.) It is only one tenth the size of the *Handbook* (or a little over one fifth the size of its 1610-page 01924 edition) and it is enormously more readable.

Guiding tables

An introduction briefly covering Reuleaux's theory of kinematics and "phoronomics" is given on pp. vi–xv (12-21/340). A table of contents is present on pp. xvi–xviii (22-24/340). An alphabetical index occupies pp. 303–312 (327–336/340). An introduction to strength of materials, with various kinds of stresses and strains, but excluding the buckling of Euler columns, is given on pp. 1–18 (column buckling is covered on pp. 82–84), and a graphical method of calculation ("graphostatics") is given on pp. 22–38. The remainder of the book, except for a short series of mathematical tables on pp. 291–301, is Section III, mostly organized as a catalog of different kinds of mechanisms and machine construction, accompanies with both lucid theoretical explanations and data tables.

Not included are optics, chemistry, metallurgy (except in the broadest sense), machining processes (such as drilling, grinding, lapping, and boring), metrology, feedback control, vibrations and resonances, fluid mechanics (?), ceramics, cams, testing, thermodynamics, textile processes such as weaving and spinning, lithography, and electricity; but substantial attention is given to hydraulics (though not to sliding seals), pneumatics, and tension members such as belts and ropes, as well as some other sorts of manufacturing processes such as rolling. And of course no attention is given to polymers except for rubber, since that art was but little developed in 01893.

Listing of chapters of section III (add 24 to page numbers to get offset in the PDF):

- **Riveting** (strength, sheet metal gauges, boilers, etc.) pp. 39–44.
- **Hooping** (by shrinkage, cold, dimensions, etc.) pp. 45–46.

- **Keying** (longitudinal, cross, for propellers, securing, etc.) pp. 47–50.
- **Bolts and Screws** (Whitworth, Sellers, metric, threadforms, etc.) pp. 50–60.
- **Journals** (lateral, thrust, friction of, etc.) pp. 60–67.
- Plain **Bearings** (lateral, thrust, wooden, etc.) pp. 68–78.
- **Bearing supports** (columns, etc.) pp. 79–84.
- **Axles** (circular, annular, ribbed, wooden, etc.) pp. 85–92.
- **Shafting** (line, deflection of, journals for, etc.) pp. 92–94.
- **Couplings** (rigid, flexible, clutch, automatic, etc.) pp. 95–101.
- **Simple levers** (cast iron, rock-arm, strength of, etc.) pp. 101–103.
- **Cranks** (wrought iron, cast iron, multiple, hand, etc.) pp. 104–109.
- **Combined levers** (walking beams, scale beams, etc.) pp. 110–111.
- **Connecting rods** (for crank pins, round, cast iron, etc.) pp. 112–118.
- **Crossheads** (free, for link connections, for guides, etc.) pp. 118–121.
- **Friction wheels** (rolling-element bearings, as gears, etc.) pp. 122–126.
- **Toothed gearing** (spur, cycloidal, pin, hyperboloidal, etc.) pp. 127–150.
- **Ratchet gearing** (spring, multiple, checking, continuous, etc.) pp. 150–171.
- **Tension organs** (hemp, wire, stiffness of, sheaves, etc.) pp. 172–185.
- **Belting** (self-guiding, pulleys, efficiency, lacing, etc.) pp. 186–194.
- **Rope transmission** (specific capacity of, hemp, wire, etc.) pp. 194–206.
- **Chain transmission, strap brakes** (efficiency, etc.) pp. 211–216.
- **Pressure organs** (pumps, valves, hydraulic tools, etc.) pp. 216–241.
- **Pipes** (iron, copper, steam, lead, stuffing boxes, pistons, etc.) pp. 242–259.
- **Tanks** (cast iron, riveted, steam, air, etc.) pp. 260–273.
- **Valves** (lift, flap, round, spiral, rotary, etc.) pp. 273–289.

Incomplete listing of figures:

- Fig. 132, common wrought-iron rivet, p. 39 (63/340)
- Fig. 971a, Siemens Geyser pump.
- Fig. 975, the common lift and suction pump and force pump, p. 223 (247/340)
- Fig. 976, Muschenbrœck’s pump for moderate lifts, Donnadieu's pump for deep wells, and the Althaus telescope pump, p. 223 (247/340)
- Fig. 977, differential pump, Rittinger’s pump, and Trevethick’s pump, p. 224 (248/340)
- Fig. 1134, bell valve, p. 276 (300/340)
- Fig. 1135, a set of 19 valves as used in the Heidt shaft at Hermsdorf, p. 276 (300/340)
- Fig. 1136, a cone of ring valves, p. 276 (300/340)
- Fig. 1167, p. 286 (310/340)
- Fig. 1169, Cuvelier’s valve, p. 286 (310/340)
- Fig. 1170, double-seated valves, mislabled Fig. 1770, p. 286 (310/340)
- Fig. 1171, p. 286 (310/340)
- Fig. 1172, an oscillating valve by Wilson, p. 286 (310/340)
- Fig. 1173, Wilson’s balanced valve, p. 287 (311/340)
- Fig. 1174, the balancing of the valves of the Porter-Allen engine, and Sweet's balanced valve, p. 287 (311/340)

• Fig. 1175, the direct and inverted siphons, p. 287 (311/340)
• Fig. 1176, a water trap in a pipe, p. 287 (311/340)
• Fig. 1177, p. 288 (312/340)
• Fig. 1178, Hoffman's furnace, p. 288 (312/340)
• Fig. 1179, Wilson's water gas furnace, p. 288 (312/340)
• Fig. 1180, Hero's Fountain, p. 288 (312/340)
• Fig. 1181, the water trap apparatus of Morrison, Ingram, & Co., p. 288 (312/340)

Translation

Terminology has changed substantially since Reuleaux's time. "Latent forces" are now, I think, called "stiffness" or "rigidity". Most confusingly, Reuleaux's "strain" S is what we now call "stress". His "tensile strength" is a stress, and I think it is the ultimate strength, not the yield stress. "Modulus of resistance" T seems to mean yield strength, and "modulus of rupture" K seems to be the ultimate strength, same as "tensile strength", suggesting I'm misunderstanding one of them. "Modulus of elasticity" E is the familiar Young's modulus of elasticity. "Theoretical resistance" is, I think, the *force* (as opposed to the stress or strain) exceeding the yield strength. "Toughness" has its modern meaning.

(The 01924 edition of the *Machinery's Handbook* already uses the modern terminology, so possibly Suplee rather than antiquity is to blame for the nonstandard terminology.)

In many cases Reuleaux tried to introduce new terminology that was not adopted: he considers valves to be "ratchets for pressure organs" (or rather the pawls thereof), pipes to be "conductors for pressure organs", etc.

Topics

• Contrivances (p. 1143) (45 notes)
• History (p. 1153) (24 notes)
• Mechanical (p. 1159) (17 notes)
• Reading (p. 1244) (4 notes)

A simple 2-D programmable graphics pipeline to unify tiles and palettes

Kragen Javier Sitaker, 02021-11-18 (updated 02021-12-30)
(6 minutes)

The problem of a CRTC is, in some sense, to compute a function from screen coordinates to color, where the function is largely given by a pipeline of some lookup tables. In the bitmap or TrueColor case, you have just a single lookup table, the framebuffer, which directly converts from screen coordinates to color. In paletted modes, aka pseudocolor, there are two lookups: first from screen coordinates to palette index, then from palette index to color. In a character generator, there are also two lookups, but they are slightly different: first from *truncated* screen coordinates to glyph index, then from glyph index combined with differently truncated screen coordinates to color. Or, sometimes, to a foreground/background selector, which is then used in combination with the truncated screen coordinates to index a table of character colors. In a tiled video game console like the NES, the situation is fairly similar to the colored-character-generator case, but there's also a palette for each tile, an offset added to the screen coordinates for scrolling, and some sprite compositing as well.

Nowadays there is less call for this sort of elaborate stuff for 2-D graphics, because human eyes are still the same resolution they were in the 01960s, while computer memories have gotten much larger and CPUs faster, and by storing the scene to be drawn in a framebuffer, you can draw whatever you want in the framebuffer. For 3-D stuff it's much more elaborate still, with shader programs doing arbitrary GPU computations that might take an arbitrary length of time.

I was thinking that an interesting sort of intermediate level of complexity would be a sort of reconfigurable systolic array (maybe the kind sometimes called a "diastolic array", though I'm not sure of the distinction). Essentially the idea is that you set up an APL-style array computation on the whole vector of screen coordinates to produce the desired colors, and then the CRTC executes this computation in a pipelined fashion, producing each pixel value as it is needed. Each node in the abstract syntax tree is assigned to some computing resource, such as a memory, an adder, or a FIFO, and they all communicate through some sort of routing fabric. In the most general case, this could be a sort of crossbar scheme, but even a simple fixed-function pipeline where each stage has the option to pass through its input would be useful in many cases. More flexible routing fabrics and data processing units permit more efficient assignments of operations to nodes, but a limited degree of flexibility might be sufficient for many uses.

I was thinking that a particularly interesting kind of node for this might be a round-robin memory, for example for fonts and colors. It might appear as, for example, eight LUT resources in the

computational fabric, consisting in fact of a counter, eight output latch registers, and an input multiplexer multiplexing the address bus among eight inputs, controlled by a counter. On each cycle, the counter latches the current memory word into the current output register and advances to the next input. In this way, a single memory could be dynamically divided among different functions, such as tile palette and glyph atlas, which vary less often than once per pixel. Thinking further, though, I'm not sure this kind of processing element is actually useful at all, unless more than one of the functions can usefully use the same data in memory (for example, for compositing multiple layers of text); eight memories that each contained about one eighth the amount of data would occupy almost exactly the same amount of chip space, and they wouldn't be limited to changing their value every eighth cycle (or fifth, or whatever, if you set the counter to reset more frequently). You'd have eight address buses and eight data buses, but they'd each be connected to one eighth as much memory, so that's not actually worse. The only drawback is really the static partitioning.

Even static partitioning could be overcome to some degree. Suppose you have an incoming 11-bit index and your memories only have 512 words. You can feed the low 9 bits of the index to four different memories, and then in a subsequent processing stage or two use the other two bits of the index to demultiplex one of the four memory outputs. This is energy-inefficient and adds a cycle of latency, but it doesn't reduce throughput over the case where you had a single 2048-word memory.

Unlike in the strict array-processing paradigm, the system could easily accommodate stateful processing. The simplest example might be generating pixel indices modulo 5 for a 5-pixel font; this can be done with a counter whose low three bits cycle to 0 and generate a carry after reaching 5, and which is synchronously reset to all 0 at the beginning of each scan line. A perhaps more interesting example is a boxcar filter using a prefix-sum node, a FIFO, and a subtractor. Doing the same thing vertically requires a buffer of the size of a scan line.

Topics

- Electronics (p. 1145) (39 notes)
- Graphics (p. 1177) (10 notes)
- Systems architecture (p. 1205) (6 notes)
- Displays (p. 1261) (4 notes)
- Tiled graphics (p. 1269) (3 notes)

Interesting works that entered the public domain in 02021, in the US and elsewhere

Kragen Javier Sitaker, 02021-11-20 (updated 02021-12-30) (15 minutes)

Writing Some notes on reading parts of Reuleaux's engineering handbook (p. 1019) I learned that the 01924 Sixth Edition of the *Machinery's Handbook* has finally served its sentence of copyright and graduated into the public domain. It occurred to me to check what else had recently thus graduated, in 02021 and 02020.

Evidently there was a 7-volume *Machinery's Encyclopedia*, also by Oberg and Jones, but unfortunately only Volume 7 seems to be in the Archive so far, and that one is scanned by Google, though it was less ineptly carried out than the great mass of Google's book scans. It is the "Index and Guide to Systematic Reading". Strangely, the *Encyclopedia*, or at least that volume thereof, seems to be devoid of illustrations.

Oberg died in 01951, but his coauthor Franklin Day Jones didn't die until 01967. As I understand it, this means that, in countries that sentence copyrighted works to life plus 70 years, and do not observe the rule of the shorter term, even the 01914 First Edition of *Machinery's Handbook* will not be set free until 02037. Argentina and the EU do observe the rule of the shorter term, for example, but Germany does not.

Edgar Rice Burroughs's *Tarzan* and *Barsoom* stories call attention; though they were published since 01925, Burroughs has been dead since 01950, so in countries that limit copyright to life plus 70 years, they are now fair game. Similarly Shaw; Orwell's *Animal Farm* and *1984*; Olaf Stapledon; Rex Ingram's *The Prisoner of Zenda*, the first Ruritanian fiction; all of Korzybski's œuvre on General Semantics; Edna St. Vincent Millay; and Schumpeter. Gurdjieff died in 01949, and so did Margaret Mitchell (*Gone with the Wind*) and Richard Strauss.

In countries that limit copyright to life plus 50 years, the work of everyone who died in 01970 has now graduated into the public domain, notably Carnap, Russell, E. M. Forster, Jimi Hendrix, Mishima, Nasser, Rube Goldberg, and the *Death be not Proud* of Gunther. Adorno, Eisenhower, Gropius, Kerouac, Meher Baba, and John Wyndham died in 01969. Unfortunately of these only Nasser is actually *from* such a country, so this doesn't help those of us laboring under more oppressive copyright regimes, even if they observe the rule of the shorter term. Taiwan, New Zealand, Thailand, Qatar, the UAE, Tunisia, Bolivia, Iran, and the PRC are notable bright spots here, and Mexico observes life + 50 for deaths before 01944-01-01, and Russia and Switzerland, 01943-01-01. India and Venezuela are currently life plus 60 years.

And Tom Lehrer effectively released all his songs to the public

domain in 02020, using his own website.

The rule of the shorter term has liberated all US works published up to 01925 (and, less significantly, all works published up to 01925, for users in the US); in 02021, that gave us *The Great Gatsby*, Woolf's *Mrs. Dalloway*, and Agatha Christie's *The Secret of Chimneys*, and in 02020 Burroughs's *Tarzan and the Ant Men*, Agatha Christie's *The Man in the Brown Suit*, *Doctor Dolittle's Circus*, Gershwin's *Rhapsody in Blue*, and W.E.B. Du Bois's *The Gift of Black Folk*. *The Boxcar Children* was published in 01924, and *The Velveteen Rabbit* in 01922. Sinclair Lewis's Babbitt also seems to be popular, and his *Main Street* was published in 01920. Not until 01926 would *The Sun Also Rises* and *Winnie-the-Pooh* appear (which last was published first in England), and we don't get *The Sound and the Fury* or *A Farewell to Arms* until 01929.

In nonfiction, 01925 gave us Ronald A. Fisher's *Statistical Methods for Research Workers*, in which frequentism was born; Walter Lippmann's *The Phantom Public*, on Mussolini's manipulation of public opinion, proposing the agent/bystander dichotomy; Napoleon Hill's *The Law of Success*, from which the entire genre of soft-headed modern self-help books derives; *The Bolshevik Myth*, by anarchist Alexander Berkman, who committed suicide in 01936, who was the lover Emma Goldman's, who died in 01940, written after he was deported to Russia, following Goldman's *My Disillusionment in Russia* and *My Further Disillusionment in Russia*. 01924 gave us Haldane's *Daedalus*; Crookshank's virulent pseudoscientific racist *The Mongol in our Midst*; and the *Lessons of October* and *New Course* by Trotsky, assassinated 01940. 01923 gave us Santayana's epistemological Scepticism and Animal Faith, Henry Smith Williams's ten-volume Story of Modern Science, the first two volumes of Churchill's World Crisis, Bernays's *Crystallizing Public Opinion*, and Jepson's Manual of the Flowering Plants of California.

01922 gave us Liddell's *Handbook of Chemical Engineering* and Born's _Einstein's theory of relativity.

H.G. Wells was English, mostly published first in England, and died in 01946, so his works have mostly been in the public domain since 02016, since the UK sentences works to life + 70; this notably includes his 01919 *Outline of History*, which rejects racism. In 01924 Wells published *A Year of Prophesying*, a compilation of his columns. *The Everlasting Man*, published in 01925, was by Chesterton, who was also English, but who died in 01936; it is a rebuttal to Wells's *Outline*.

The US doesn't observe the rule of the shorter term for works published in 01926 to 01977 unless they were in the public domain in their source country on 01996-01-01, so I think Wells's post-01925 works are not yet free in the US, but Chesterton's are. (Churchill and Eliot didn't die until 01965.)

Other public-domain works that catch my eye in the Archive:

- *East of the Sun and West of the Moon*, 01922, published in New York, a book of folk tales I enjoyed as a kid
- *History of the Cherokee Indians and their legends and folk lore*, published

in Oklahoma, 01921
• *The Babur-nama in English*, published in London, 01922, translated by Annette Susannah Beveridge, née Akroyd, who died in 01929
• *The Poetic Edda*, translated by Henry Adams Bellows (died 01939), published in New York, 01923
• *English Industries of the Middle Ages*, by Louis Francis Salzman, died 01971, published at Oxford, 01923 (not public domain outside the US)
• *The Joy of Cooking*, first published in the US in 01923 with unchanged reprintings up to 01957; Irma S. Rombauer died in 01962, but her daughter Marion Rombauer Becker didn't die until 01976. So the original editions are public domain in countries observing the rule of the shorter term, but not those that sentence foreign works to life + 70 years.
• *Fuel and Refractory Materials*, poorly scanned in Bangalore, by Alexander Humboldt Sexton, published in Glasgow and Bombay in
• Sexton seems to have been born in 01853 and retired from the Chair of Metallurgy in 01909, so he almost surely died before 01950, but I have not found death records yet.
• *A History of Magic and Experimental Science During the First Thirteen Centuries of Our Era, Volume I*, by Lynn Thorndike (died 01965), published 01923 by Columbia University Press (New York and London) and the Macmillan Company, describing the evolution of science from Roman times (Pliny, Seneca, and Ptolemy) up to the 13th century (Roger Bacon, Raymond Llull, Albertus Magnus).
• *Among the Ibos of Nigeria*, published in London, 01921, by George Thomas Basden, died 01944, about the Igbo, covering marriage customs, slavery, religion, etc., with many high-resolution black-and-white photos.
• *A course of pure mathematics*, by Godfrey Harold Hardy (died 01947), Third Edition, published at Cambridge in 01921, a textbook of basic analysis (complex numbers, continuous functions, derivatives, integration, circular functions, exponentials, infinite series).
• *Advanced laboratory practice in electricity and magnetism*, published in New York and London in 01922 by McGraw-Hill, by Earle Melvin Terry, died 01929. Covers units, switches, rheostats, galvanometers, Ohm's Law, Weston cells, Kelvin balances, ammeters, voltmeters, wattmeters, capacitance, resonance, electron-tube oscillators, alpha rays, beta rays, gamma rays, optical pyrometers, etc., with exercises. Scanned in color at 400 dpi with numerous schematics and engravings.
• *Servant of Sahibs*, excerpts from the travel diary of famed Ladakhi explorer Ghulam Rassul Galwan (died 01925), who named the Galwan river, published in London in 01923. This edition seems to be devoid of illustrations.
• *Mazes and Labyrinths: a General Account of Their History and Developments*, by William Henry Matthews, who died in 01948, published in 01922 by Longmans, Green, & Co., in London, New York, Toronto, Bombay, Calcutta, and Madras., dedicated to his daughter Zeta (01914–02000), with 151 photos and other illustrations. Though it focuses mostly on British mazes it also covers, for example, Pima and Mesa Verde mazes, and Troy.
• *Welsh Fairy Tales*, by William Elliot Griffis (died 01928), published in 01921 in New York by the Thomas Y. Crowell company. A book

of children's stories.
• *A Treatise on Probability*, by John Maynard Keynes, reprinting of 01921 first edition by Macmillan and Company. I didn't know Keynes had written a textbook on probability; on skimming, this seems to spend as much time on epistemology as on mathematical proofs.
• *The Prophet*, published 01923 in the US, by Kahlil Gibran (died 01931), poorly scanned by Digital Library of India.
• *Heraldry and Floral Forms as Used in Decoration*, published 01922 in London, by English book illustrator Herbert Cole, died 01930. Abundant engravings scanned at 500 dpi.
• *Sir Isaac Newton's Daniel and the Apocalypse*, by Sir William Whitla, died 01933, published 01922 in London, with the Latin translated into English.
• *The Inscriptions of Asoka*, by Eugen Hultzsch, died 01927, published at Oxford 01925, with many black-and-white photos of the inscriptions, beautifully scanned at 300 dpi, along with transcriptions, transliterations, and translations.
• *Leonhari Euleri, Opera Omnia (Opera Mathematica Volumen Septimum): Commentationes Algebraicae: Ad Theoriam Combinationum et Probabilitatum Pertinentes*, published by Louis Gustave Du Pasquier, died 01957, published 01922, 644 pp., evidently mostly in French, with French notes by L. G. D., including manuscripts published here for the first time; though some Latin is present. I'm not sure whether Euler's French is Euler's or a translation by L. G. D. This is the 7th volume of the *Complete Works of Leonhard Euler* (*Oeuvres complètes de Léonard Euler*). The scan is not very good because it's by Akce-Universal Digital Library, but it's not totally fucked up like their other scans. L. G. D. taught at the University of Neuchâtel in Switzerland. I'm not sure where this was published but I'm pretty sure it wasn't the US, so it's probably not in the public domain elsewhere.
• *Selected stories from O. Henry*, died 01910, including *The Gift of the Magi* (first published 01905), edited by his friend C. Alphonso Smith, "late head of the Department of English at the United States Naval Academy", died 01924, published by the Odyssey Press, New York, in 01922.
• *The Story of Little Black Sambo*, by Helen Bannerman (died 01946), published in New York in 01923 by the Frederick A. Stokes Company, following a London publication in 01899. Frequently criticized for racism, the story itself contains no racist stereotypes, but the illustrations do; around World War II the name Sambo became a byword for racism, though perhaps originally it derives from Foulah *sambo* "uncle" or Hausa *sambo* "second son".
• Lyman Churchill Newell's 01922 extremely basic introductory textbook for children, with exercises; originally published by D. C. Heath & Co. (Boston, New York, Chicago), and he died in 01933. Scanned in color at 500 dpi, with 215 or so grayscale photographs and other figures. The 560-page PDF is compressed very badly, inflicting fatal damage on most of the hundreds of photographs and other illustrations, but the original JP2 files are available and are not so corrupted. Organic is relegated to "fuels and illuminants", "other carbon compounds" and "food", pp. 260-315, which I suppose in part represented the state of knowledge at the time. Imperial units are

used throughout except for temperature. "The topics suggested by the College Entrance Examination Board and the Board of Regents (New York) have been incorporated." The low information content of this book is hard to overstate; Gibbs free energy is not mentioned, and the notion of equilibrium is relegated to p. 152.
• In Praise of Folly, by Desiderius Erasmus, in which he castigates royalty and the Church, published in New York by Peter Eckler Publishing Co. in 01922, with numerous engraved illustrations by Hans Holbein. Peter Eckler himself wrote the preface, which is followed by a brief biography of Erasmus, who died in 01536. The book seems to be printed entirely in English, but the biography and translation are uncredited. Erasmus did presumably speak English, as he lived in England from 01510 to 01515, but the first edition was published in Latin and Greek (mostly Latin), with engravings by Holbein, who evidently died in 01543. Eckler had previously published Walt Whitman's *Drum-Taps* in 01865, and *The Canon of the Bible* in 01877, so he must have been very old by 01922! My best guess is that Eckler was reprinting an earlier English translation, but not John Wilson's 01668 translation, which is notably different and worse.

Topics

• History (p. 1153) (24 notes)
• Reading (p. 1244) (4 notes)
• The United States of America (USA) (p. 1314) (2 notes)
• Law
• Copyright

At small scales, electrowinning may be cheaper than smelting

Kragen Javier Sitaker, 02021-11-21 (updated 02021-12-30) (25 minutes)

I was thinking that on small scales (sub-meter, especially sub-millimeter) it might be more economical to reduce metals from ores by aqueous electrowinning than by smelting, because maintaining large thermal gradients is very difficult.

If the things being constructed are themselves small, the strength of materials is not very important, because at small scales even very weak materials are strong enough to hold together except at very large accelerations. Metals, however, have some other interesting properties: they can conduct electricity, they have very low vapor pressures and so can withstand exposure to space, and they can be readily shaped by electrochemical machining.

Macroscopically, hardness is very important for abrasion or cutting, but I suspect that these shaping processes, like sliding-contact joints, will not be very usable at small scales because of the rapidity of surface wear and the comparatively large forces involved in surface contact. However, at scales above where this is true, hardness is still important, because it determines what can cut what else.

Casting and molding are also very important shaping processes at the human scale. At submillimeter scales, the same thermal problems that impede XXX

pH, CO_2, H_2O, O_2

pressure

Thermal versus electrical insulation: what about *not* electrowinning?

Consider the Ellingham diagram for iron, which shows that smelting iron requires a temperature of at least 700°, more practically 1000° or more. If outside the smelting apparatus the temperature is 25° then we have some 975° of temperature difference. If we have a meter of refractory insulation, that's 975 K/m. Vermiculite's insulating value is about 16–17 K m/W, a conductivity of about 0.06 W/m/K, giving about 60 W/m² with that gradient, a heat flux which is, in the steady state, uniform throughout the thickness of the material. Aerogel is about three times as good, insulating firebrick about three times worse, and most other insulating materials are in between. The worst insulator of all, diamond, is about 1000 W/m/K.

Now suppose we scale the apparatus down by a linear factor of 1000, cutting the insulation thickness to 1 mm. Because the thermal gradient has increased by a factor of 1000, we are now losing 60 kW/m². This poses a real difficulty inside the apparatus. Because the surface area covered by insulation has increased by a factor of a million (say from 600 m² to 600 mm²) so we are dissipating only one thousandth as much power as before to maintain the kiln at smelting

temperature; but the volume over which that power must be generated has diminished by a factor of a billion (say to 1 milliliter), requiring a million-fold increase in the power density of our heating elements, say to 35 W/ml, which is achievable but problematic. Scaling down further rapidly becomes impossible; at 1 micron thickness we are losing 60 MW/m^2, which for a 10-micron cube amounts to 36 milliwatts.

It also poses a difficulty outside the apparatus, because removing 60 kW/m^2 requires either radiation at an uncomfortably high temperature ("60 suns", as they say for solar concentrators) or a lot of coolant, but this is a less serious problem.

Scaling in the opposite direction, we would reach a point where even shitty insulating materials would thermally insulate adequately.

High-temperature processes are possible in a low-temperature environment at the micron scale if they can be carried out very quickly and intermittently. For example, a cubic micron of material, weighing on the order of 5 picograms, can be heated to 2000° for a short period of time with an energy on the order of 10 nanojoules. It cools off through conduction with on the order of a milliwatt, so several milliwatts is required to reach this temperature, which then cools off on a timescale on the order of a microsecond. Lasers and electron beams are straightforwardly capable of being switched with submicrosecond timescales and delivering such power densities.

Resistance heating is also straightforward. 10 milliwatts at 10 volts is a milliamp and thus 10 kΩ. If our joule heater is amorphous carbon at 6×10^{-4} Ω m, a 1-micron cube of it would give us 600 ohms; we could either increase the current to 4 mA and reduce the voltage to 2.4 volts, or we could increase the aspect ratio of the heating element, but either way it seems clear that we will have no trouble reaching the desired temperatures on the desired timescale with easily constructed circuitry.

Electric arcs and pseudosparks are another candidate method for achieving such temperatures rapidly enough.

By contrast, no metal needs as much as ten volts to reduce it. Common insulators have electrical resistivities of 10^{11} Ω m and up, the best ones exceeding 10^{23} Ω m, while conductors are in the neighborhood of 10^{-9} Ω m. Ten volts across a millimeter of a 10^{15} Ω m substance like sulfur or dry wood produces a current of about 10 pA/m^2 and thus 100 pW/m^2, almost 15 orders of magnitude less than the thermal leakage calculated above through a thermal insulator for the temperature needed to smelt iron. If the linear approximation for conductivity were accurate this far down, a 1-nanometer-thick layer of such an insulator would permit only 0.01 mA/m^2 (and 0.1 mW/m^2) of conduction. In fact breakdown voltage becomes a much more significant concern than energy loss to conduction; fused silica can withstand some 500 volts per micron, but other materials are closer to 10, so they'd need over a micron of insulation. At small scales vacuum becomes the best choice of insulator, since most metals don't suffer field emission until over a gigavolt per meter, which would be 0.01 microns of insulating vacuum.

A micron-thick wire at 10^{-9} Ω m has 1.2 kΩ/m of resistance, which is an almost entirely insignificant 1.2Ω/mm. So electrical

transmission is not perfectly efficient but it does not pose feasibility problems for micron-scale electrowinning in the way that thermal conductivity does for micron-scale carbothermic reduction.

Metal selection

The eight ancient metals are iron, gold, copper, lead, tin, silver, mercury, and, in India, zinc. Today I think the most important metals are aluminum, iron, copper, zinc, tin, tungsten, nickel, chromium, lead, cobalt, molybdenum, vanadium, magnesium, titanium, platinum, gold, zirconium, and the semimetals carbon and silicon. LME's "non-ferrous" category includes aluminium, copper, zinc, nickel, lead, tin, aluminium alloy, NASAAC ("North American Special Aluminum Alloy Contract"), "aluminium premiums", alumina, and aluminium scrap; "precious" is gold, silver, platinum, and palladium; and "EV" ("electric vehicle") is cobalt, molybdenum, and lithium.

Of course many other metallic elements are widely used, in an oxidized form, such as calcium, sodium, and potassium, and there are niche uses of almost all of the metals. But what I'm mostly concerned with here is *reducing* metals from their oxidized form.

Aluminum

Aluminum is resistant to corrosion in air, nearly as abundant as iron, and although it is not as strong as steel per volume, it is stronger per weight, much easier to shape, and more conductive per mass than copper. It also has an astoundingly high boiling point, 2470°, and an extremely useful oxide. 65 million tonnes are mined per year, and it costs about US$2/kg.

Unfortunately, there is no known way to electrowin aluminum in an aqueous solution; metallic aluminum has a -2.33-volt standard electrode potential to reduce to hydroxyls, while hydrogen is only -2.23 volts, so aluminum will steal oxygens from hydronium. Instead aluminum is electrowon by dissolving alumina in cryolite Na_3AlF_6, which requires a temperature around 1000°; neat cryolite melts at 1012°, but the eutectic is only 960°.

Of my list of "important metals" above, magnesium, titanium, and zirconium have the same problem, but the others should *all* be electrowinnable with low-temperature processes.

Alternative processes for reducing aluminum might include plasma electrolysis, mass spectrometry, electron-beam reduction *in vacuo*, and simple carbothermic reduction using intermittent heating.

Iron

Iron is one of the most abundant and strongest metals, and it can withstand moderate heat (1500° or so without oxygen, much more than aluminum or brass, though not in the same ballpark as sapphire, graphite, tungsten, molybdenum, etc.). It's the main metal used for construction and machinery, having mostly displaced the more expensive bronze and brass as the humans improved their techniques for shaping the more stubborn iron. A couple billion tonnes of it are mined per year, and I think scrap iron costs about 25¢/kg

(US$213/ton in 02020).

Electrolytic iron is commercially used in cases that require especially high purity or small particles, such as cereal fortification, powder metallurgy, or high-coercivity powdered-iron magnetic cores.

US Patent 4,134,800 from 01979, by Prasanna K. Samal and Erhard Klar, describes one process, using a bath of ferrous sulfate (36–40 g/l of iron ion) and ammonium sulfate (24–28 g/l of ammonia ion), with 1.4–1.6 grams of iron per gram of ammonium, a pH of 5.6–6.0, a temperature of 38°–49°, and 18–26 amps per square foot (194–280 A/m²), which they say isn't critical. Their declared aim was to make the iron more brittle so it could be ground, which they hoped to achieve by iron hydroxide formation. As a "prior art bath" they gave as an example 50 g/l ferrous ions, 13 g/l ammonia ions, pH 5.4, 38°–43°, 22 A/ft² (237 A/m²). They carefully didn't mention their voltage, electrode spacing, agitation, aeration, electrolytic cell size (1 liter or 1 tonne?), or Faraday efficiency, and they didn't mention any other additives, which hopefully they didn't have.

If you had sulfate, you could presumably digest iron ores with it and then follow this process. In fact, you could probably continuously digest iron oxides in the sulfate electrolysis bath.

Samal and Klar cite patents 2,464,168 (Fansteel, 01949), 2,481,079 (Chrysler, 01945), and 2,626,895 (Fansteel, 01944). A little further searching turns up patents 1,782,909 (Pike, 01930), 2,464,889 (Pike and Schoder, Tacoma Powdered Metals, 01949), 2,503,235 (Cain, Sulphide Ore Process Co., 01950), 1,162,150 (Estelle, 01915), 2,538,990 (Trask, Buel Metals, 01951), 3,041,253 (Audubert and Lacheisserie, 01962) and, for nickel, patents 3,414,486 (Nordblom and Bodamer, ESB, 01968) and 483,639 (Strap, 01892).

The Estelle patent is particularly interesting for being over a century old and claiming to make iron pyrite an economic source of iron, which it is not at present (though the name of Cain's company above suggests it used to be). He was electrolyzing ferrous chloride, formed by digesting the pyrite with muriatic acid, and then recycling the resulting ferric chloride solution into muriatic acid and ferrous chloride by reducing it with sulphuretted hydrogen (produced in the first step), producing sulfur as a byproduct. He says that nickel, cobalt, and zinc can be co-precipitated with the iron, but the zinc is easily enough driven off.

Cain's patent is especially helpful in telling us that at the time (01946) there were two main processes for electrodeposition of iron, one involving the dissolution of an iron anode and one that doesn't (because it's digesting an oxide or something similar); and that usually you use an asbestos anode bag to contain the crap formed on the anode. He says it's good to keep the pH below 2 with muriatic acid. (You'll pardon me if I prefer polyethylene or polyester to asbestos.)

Audubert and Lacheisserie (concerned with fine particle size) say you can use most ferrous salts, but sulfate and chloride are best, and that if you're getting oxidized iron, either you have oxygen dissolved in the bath or you have too much ferric iron, and that they use 0.65 volts.

Anyway, so it seems like it's slightly tricky, but not nearly as tricky as you'd assume from the negative standard electrode potential of iron. And I guess it would have to be not that tricky for Edison's nickel-iron battery to be rechargeable.

Copper

While iron is crucial for moderate temperatures and strength, the much less abundant copper is crucial for electrical conductivity, low-friction bearing surfaces for iron parts, corrosion resistance in oxygen atmospheres, and high thermal conductivity for heat exchangers. 25 million tonnes are produced per year; it costs US$6.20/kg.

Copper is so easy to electrodeposit (and electro-etch) that it's easier to enumerate the cases where it *won't* work: where you're trying to form an adherent deposit on an electrode that copper will spontaneously oxidize, such as iron, and when the anions in your electrolyte don't form a soluble copper salt (among the usual suspects, these are iodide (mostly), cyanide (without enough ammonia), thiocyanate, hydroxide (i.e. bases or just water), oxalate (again, without enough ammonia), and phosphate). The USGS says that there are currently 3 electrolytic refineries for copper in the US and 14 electrowinning facilities.

Zinc

Zinc is used to add corrosion resistance to iron in oxygen atmospheres (its main industrial use today), in Zamak, as an alloying element for copper to form brass, and in its oxidized form, as a white pigment. It has a remarkably low boiling point, 907°. 12 million tonnes are produced per year; it costs US$2.40/kg.

Despite the name "galvanization", zinc coating was originally done not as electroplating but as a hot-dip process, which is still the most common way to do it today. But electroplating zinc is also a common thing to do, and there's lots of historical work on producing zinc powder electrolytically.

"Zamak" is a family of low-temperature zinc-based casting alloys, some of which have strength comparable to steel; Zamak 2 (4% aluminum, 2.7% copper, 0.04% magnesium) has a tensile strength of 330 MPa, a Young's modulus of 96 GPa, and melts over the range 379–390°. Unfortunately the aluminum is a necessary component, and slight lead impurities will wreck Zamak with zinc pest.

Brass

In modern practice, brass (about 20% zinc, US$5.40/kg) has mostly been displaced by steel, which is stronger, harder, stiffer, lighter, and cheaper (more than 20× cheaper by weight), and, in high-carbon cases, can be hardened by heat treatment. But brass still has many small-volume niches.

It is enormously easier than steel to cast or, especially with a bit of lead, to cut.

It's more corrosion-resistant in oxygen atmospheres and in water, especially salt water; "admiralty brass" is 70% copper, 29% zinc, and 1% tin (see below) and is an especially good formulation for this.

Brass has higher thermal and electrical conductivity than steel, and so in particular it lasts much longer for EDM electrodes.

It has much lower friction on steel than steel does, so it can be used for plain bearings (journals), as a cheaper and less durable alternative to bronze (though babbitt is often better still).

It's used as a solder to join steel parts ("brazing"), which allows a stronger connection than bolts, with lower temperatures and less distortion than welding, and it can join a wider collection of materials than welding, including tungsten carbide (see below).

Because it's softer than steel, brass doesn't produce sparks and doesn't mar steel surfaces, so in some environments and for some purposes brass hammers and other tools are preferred to steel.

Finally, its yellow color is often used for aesthetic purposes. With just zinc and copper, you can make silver (zinc), red (copper), and yellow (brass).

Galvanizing

Galvanized steel, steel coated with zinc, has mostly replaced tinplate as an anti-corrosion coating. Zinc is somewhat toxic in food (the oral rat LD50 of the highly soluble zinc chloride is 350 mg/kg, and it's also used topically to induce skin necrosis in "black salves") and produces toxic fumes when heated near its boiling point, so this isn't done for tin cans or cooking pots, but it's widespread for things like buildings. As mentioned above, this is usually done as a hot-dip thing, but it can be done through electrodeposition.

Tin

Tin is crucial for soldering electronics; alloyed with copper it is bronze; alloyed with copper and antimony it is babbitt; coating steel it prevents corrosion; and it melts at only 232°. The largest of its many uses today is as a nontoxic anti-corrosion coating for steel in "tin" cans. Bronze can withstand both higher temperatures and more stress than brass, while retaining brass's easy castability. Babbitt, which makes the best plain bearings, is tin with 2.5–5% copper (occasionally as high as 8.5%) and 4–8.5% antimony. Some 0.3 million tonnes of tin are mined per year, and it costs about US$18/kg.

You might think its numerous oxidation states (2+ (stannous) and 4+ (stannic), sometimes + and 3+, as well as neutral and negative states) would make it difficult to electrowin. The sulfate, bromide, chloride, and fluoride, all divalent, are water-soluble; the iodide is mildly so, and the bromide is additionally soluble in donor solvents like DMSO. There are also a tetravalent bromide, chloride, fluoride, iodide, sulfide (sphalerite), and nitrate; the tetravalent chloride is a liquid that mixes with all kinds of nonpolar liquids, and the tetrabromide is also water-soluble. The nitrate is, unusually, unstable in water. The sulfate is preferred when stannic ions are undesired, because there is no stannic sulfate.

Tin electroplating is widely practiced using acid baths (I'm guessing sulfuric), alkaline baths (I'm guessing stannate; you can get sodium stannate by digesting tin with lye), and methylsulphonic acid baths. It's often codeposited with lead, copper, silver, zinc, and/or bismuth.

Tungsten

Tungsten has the highest melting point of any metal (3422°), almost as high as carbon's sublimation temperature of 3642° and the melting points of tantalum hafnium carbide (3990°), tantalum carbide (3880°), and hafnium carbide (3928°), though well short of tentative results for hafnium carbonitride (4200°). Tungsten also has the highest boiling point of all elements, an astounding 5930°. It's an essential ingredient in high-speed steel, though vanadium and molybdenum can replace it to some extent, and tungsten carbide (the main current use of tungsten) has largely replaced high-speed steel in modern steel-cutting practice. It's also essential to TIG welding and important in vacuum tubes and incandescent lights. Some 84000 tonnes are mined per year, 80% in China, but I don't know what it costs.

Carbides of vanadium, molybdenum, niobium, and the titanium-group metals are possible substitutes for tungsten carbide.

The current industrial process for smelting tungsten is long and involved, but the main article of commerce is tungsten trioxide, which is then either carbothermally reduced or reduced with hydrogen.

Experiments have been made in electrowinning of tungsten at 1080°, but also US patent 2,384,301 (Harford, 01944) and others describe electrodeposition methods for reducing tungsten. Harford recommends complexing your tungsten with 25% ethylenediamine in water, using 25 A/ft², but he explains that people previously just used cyanide.

The titanium group

I think low-temperature electrowinning of titanium, zirconium, and hafnium is basically a lost cause with current electrochemistry. This is a real shame, because titanium is as strong as iron and much lighter.

Perhaps even more interesting than the metals, though, are the carbides, nitrides, borides, and oxides of this group, which are outstanding materials in many ways: ultra-high temperature ceramics, superhard, transformation-toughened, solid electrolytes, photocatalysts, super-high-kappa dielectrics, resistant to chemical attack, high-conductivity semiconductors, etc. They are often produced from the metals, but for example zirconium diboride can be made from refined zirconia, boria, and metallic magnesium, or from boron and zirconia, or boron carbide and zirconia. Nitrides can be made by reacting the oxides with ammonia or nitrogen, etc.

However, of the oxides, only titania (rutile or anatase) occurs in nature. Zirconia (mixed indiscriminately with hafnia) is obtained from zirconium silicate (zircon or jargoon) by calcining.

Electrowinning to separate metals

In most cases it's difficult to electrodeposit alloys; metals tend to get separated from each other by the process. Sometimes this is because of differing solubilities; lead sulfates, for example, are insoluble, so lead won't electrodeposit from a sulfate bath.

(Chromium has both soluble and insoluble sulfates, and of course barium and calcium have insoluble sulfates, but they're too reactive to electrodeposit from water.) But that's not unique to electrochemistry; that's just regular heap-leach mining chemistry.

The much more interesting fact is that by setting the voltage low enough, you can generally electrodeposit *just a single metal* from an electrolyte containing different kinds of cations, because no two metals have exactly the same electrode potential. This is potentially *very* interesting: it's a high-throughput, high-efficiency, small, low-temperature way to separate many different ionic species. It won't work for every case, because of considerations like those mentioned above for iron. But it will work in many cases.

By the same token, it's often possible to dissolve just one metal out of an alloy anode by setting the voltage at the right level.

Single displacement and the Tree of Saturn

In general if a metal can be electrowon it can also be precipitated by a single displacement reaction from a more reactive metal. Standard electrode potentials include:

solutes	metal	E°/V	electrons
$Li^+ + e^-$	Li(s)	−3.0401	1
$Na^+ + e^-$	Na(s)	−2.71	2
$Mg^{2+} + 2e^-$	Mg(s)	−2.372	2
$Al^{3+} + 3e^-$	Al(s)	−1.662	3
$Ti^{2+} + 2e^-$	Ti(s)	−1.63	2
$Zr^{4+} + 4e^-$	Zr(s)	−1.45	4
$V^{2+} + 2e^-$	V(s)	−1.13	2
$2H_2O + 2e^-$	$H_2(g) + 2OH^-$	−0.8277	2
$Zn^{2+} + 2e^-$	Zn(s)	−0.7618	2
$Ta^{3+} + 3e^-$	Ta(s)	−0.6	3
$Fe^{2+} + 2e^-$	Fe(s)	−0.44	2
$Co^{2+} + 2e^-$	Co(s)	−0.28	2
$Ni^{2+} + 2e^-$	Ni(s)	−0.25	2
$Sn^{2+} + 2e^-$	Sn(s)	−0.13	2
$Pb^{2+} + 2e^-$	Pb(s)	−0.126	2
$2H^+ + 2e^-$	$H_2(g)$	0	2
$Cu^{2+} + 2e^-$	Cu^+	+0.159	1
$Cu^{2+} + 2e^-$	Cu(s)	+0.337	2
$O_2(g) + 2H_2O + 4e^-$	$4OH^-$	+0.401	4
$Cu^+ + e^-$	Cu(s)	+0.52	1
$Ag^+ + e^-$	Ag(s)	+0.7996	1
$Au^{3+} + 3e^-$	Au(s)	+1.52	3

So, if you have some divalent lead salt such as lead acetate in water, and you put a less noble metal into the water, such as aluminum, titanium, zirconium, vanadium, zinc, tantalum, iron, cobalt, nickel, or even tin, you should expect the lead to precipitate, dissolving the other metal into the water; this is the famed Tree of Saturn of the alchemists, and when instead done with a soluble salt of silver, it is the Tree of Diana. The same thing explains the immersion plating of silver ions onto copper with brief immersion at 50° to 60°, immersion plating of gold onto copper at 80° to 90°, immersion

plating of gold onto nickel, and so on.

As I understand it, the difficulty in electrowinning aluminum, magnesium, and the titanium-group metals is precisely that they have a more negative electrode potential than hydrogen, so they form an "immersion plating" of hydrogen, consuming the water and the metal. Normally they are protected from this reaction by an impermeable oxide layer, so they don't dissolve spontaneously in water the way lithium and sodium do.

So, in theory, you ought to be able to precipitate out any of the nobler metals from solution by starting with a hunk of zinc.

Topics

- Materials (p. 1138) (59 notes)
- Electrolysis (p. 1158) (18 notes)
- Frrickin' lasers! (p. 1168) (12 notes)
- Aluminum (p. 1180) (10 notes)
- Minerals (p. 1210) (6 notes)
- Steel (p. 1222) (5 notes)
- Small things (p. 1223) (5 notes)
- Copper (p. 1234) (5 notes)
- Refining (p. 1335) (2 notes)

Micro ramjet

Kragen Javier Sitaker, 02021-11-22 (updated 02021-12-30)
(3 minutes)

Most of the fuel of a rocket is its oxidizer. For the air, ramjets are an appealing alternative: just carry the reducer, squirt it into a combustion chamber, and let the hot compressed incoming air maintain the combustion!

The autoignition temperature of heptane is 223°, and it's nearly as energy-dense as diesel or regular jet fuel: gasoline is 34 MJ/l versus diesel's 39 or kerosene's 35. So if the incoming air can get over 250° or so, a bit less than doubling its input temperature, we're golden; from there it's just a matter of adding the heptane or whatever gradually enough to avoid cooling the air below ignition temperature.

How much compression does that need? For an ideal gas, $PV = nRT$; in an isothermal process, where nRT is constant, $PV =$ some constant C. So the short answer is that we need to double PV. But when we increase P, V decreases. By how much?

Well, in adiabatic heating and cooling, $PV^n = C$, where n is the adiabatic index, 7/5 for diatomic gases. So, I guess, if the volume is cut in half, then the pressure needs to increase by a compensating factor of $2^{7/5} = 2.64$, which means that the product PV and therefore the temperature increased by 32% ($2^{2/5} = 1.32$). So to double the temperature we need to decrease the volume by $2^{5/2} = 5.66$, which will increase the pressure by $5.66^{7/5} = 11.314$, and $11.314/5.66 = 1.999$.

(I had to write 24 lines of Python to figure that out.)

So we need about 11 atmospheres of pressure on the front of the ram in order to run the jet. How fast is that?

As I understand it, in isentropic compressible flow, the stagnation pressure is $(1 + ½(n-1)M^2)^{(n/(n-1))}$ times the static pressure of the surrounding air, at Mach M. Here n is 7/5, $n-1$ is 2/5, $n/(n-1)$ is thus 7/2, so this simplifies to $(1 + M^2/5)^{7/2}$. So, to get 11 times higher stagnation pressure:

$$11 = (1 + M^2/5)^{7/2}$$
$$11^{2/7} = 1 + M^2/5$$
$$5(11^{2/7} - 1) = M^2$$
$$M = (5(11^{2/7} - 1))^{½}$$

This works out to be about Mach 2.22, about 760 m/s at sea level, if I've calculated everything correctly. But I suspect that it isn't correct because Wikipedia talks about subsonic ramjets, and they surely aren't using fuel that ignites at a much lower temperature than heptane, right? Indeed, WP says that they've been run as low as 45 m/s, but need to run at at least Mach 0.5 to be self-sustaining.

A crucial thing here is that the stagnation pressure and thus the stagnation temperature doesn't depend on the scale or shape of the ramjet in any way; it's the same for a millimeter-wide ramjet or a kilometer-wide ramjet. I'm not sure if that's part of my error, though. The ideal-gas assumptions break down in the transonic region, as I understand it, but I don't think that's my problem.

Topics

- Physics (p. 1157) (18 notes)
- Facepalm (p. 1199) (7 notes)
- Small things (p. 1223) (5 notes)
- Flying (p. 1296) (3 notes)

Vernier indicator

Kragen Javier Sitaker, 02021-11-22 (updated 02021-12-30)
(6 minutes)

I was thinking of SunShine's flexure indicator 3-D printed from PLA, just using a couple of "blade flexures" that converge on an indicator needle, nearly at the same point, perhaps 0.1 mm apart; the needle is about 20 mm long. Although he doesn't have the thing calibrated, he was able to use it to detect the thickness of a 50-micron-thick sheet of paper, which produced about a millimeter of movement.

(I've put quotes around "blade flexures" because each "blade" is made up of a set of parallel wiggle bars in order to allow them to connect to the same bar at almost the same point without interfering; "we stagger the layers", as he says.)

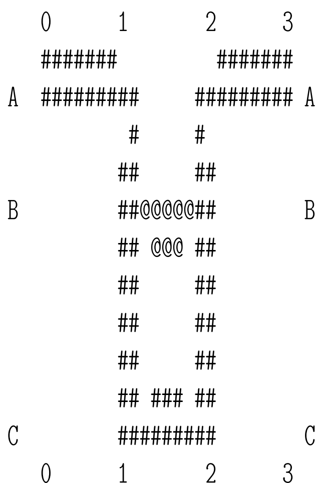
In SunShine's mechanism, most of the actual flexing takes place far away from the indicator itself, which unfortunately greatly reduces the amplification factor to only about 20:1; this could be remedied with a more rigid flexure design, at the cost of increasing plunger force, but a better flexure design is also possible. He also has sinned against flexures by making the plunger shaft a sliding contact with the 8mm indicator stem rather than using parallel blades or some similar prismatic flexure joint.

The total range of motion of his indicator needle is about 10 mm, which reduces the precision of readings available, even if you calibrate the device. It occurred to me that using the vernier principle it should be possible to make much smaller rotations easily visible. By printing a disc that rotates relative to a fixed disc with graduations at a slightly different frequency, you can visually see quite small rotations. Better still, I think, would be to print each disc with a series of holes or slits in it, rather than merely graduations on the surface, and place the two in sliding contact with one another, or a flexural approximation thereof, so that the place where the holes coincide moves around the dial much more rapidly than the holes themselves.

It ought to be possible to get 200-micron holes with conventional 3-D printing and laser-cutting processes, which ought to afford about 20-micron visible precision on the outer edge of the dial. If the mechanical advantage can be set to 50:1, this would provide 400-nanometer resolution.

As with a ruler or caliper, thermal expansion or contraction will introduce error in the measurement. However, uniform expansion either of the dials or of the plunger and stem poses no such risk, because such expansion doesn't change the angles; it's specifically the part of the plunger outside the stem, and to a much greater extent, the lever arm over which the plunger's translation is transformed into rotation, which determines the calibration. Thus it should be possible to incorporate a small piece of wood, invar, glass, sapphire, carbon fiber, or fused quartz into that part of the movement, or build it like a gridiron pendulum, to get a measurement tool that is immune to such problems, differential though it is.

A simpler way to cancel thermal expansion in one dimension than a gridiron pendulum is with the following structure:



Here the # represents a material with a large thermal coefficient of expansion, and the @ represents a (normally more of a pain in the ass) material with a smaller but still positive TCE. There are six flexural joints in this setup: A1, A2, B1, B2, C1, and C2; let's suppose that essentially all the flexion happens there, while the rest of the structure remains rigid. Consider the ratio of distances AB:AC. If this is the same as the ratio of TCEs between the two materials, then uniform heating will not change the distance A1A2. By putting B a little bit further down, we can get a *negative* coefficient of expansion for A1A2, which could be chosen, for example, to cancel the coefficient of expansion for A0A1 and A2A3, so that the distance A0A3 is invariant with uniform heating.

In this literal form the structure would not be very stable; in practice you would want to stiffen it. Making just B1 and/or C1 perfectly rigid would probably answer for many purposes, and even if C2 were rigid the structure might work adequately with the right corrections. If it could be arranged to be under constant compression, the low-expansion B1B2 member could possibly be a ball bearing (steel: 12 ppm/K), or a glass marble (8.5 ppm/K), perhaps held in two lengthwise V-grooves in the A1C1 and A2C2 members, so that any necessary rotation can happen by rolling, without stick-slip movement. Constant stress, whether compression or not, would probably rule out the use of low-melting and therefore high-creep materials like PLA.

Many materials might serve. Radial expansion for Douglas fir is given as 27 ppm/K, while parallel to the grain it is 3.5, but it is also sensitive to humidity. Brass is 19, aluminum 23, fused quartz 0.59.

Getting back to the indicator, a simple expedient might be to laser-cut the whole thing from one to three sheets of steel. Steel has significant thermal expansion and contraction, but it's much smaller than that of many alternative materials; polypropylene's TCE is given as 150, 12 times higher, and even PLA is about 40 ppm/K. And, unlike them, steel isn't hygroscopic and doesn't creep significantly at ambient temperature.

Topics

- Contrivances (p. 1143) (45 notes)
- Precision (p. 1183) (9 notes)
- Hand tools (p. 1197) (7 notes)
- Metrology (p. 1212) (6 notes)
- Flexures (p. 1232) (5 notes)
- Length (p. 1356) (2 notes)

Some notes on Bhattacharyya's ECM book

Kragen Javier Sitaker, 02021-11-25 (updated 02021-12-30)
(11 minutes)

Reading Bhattacharyya's 02015 book on electrochemical micromachining. Seems like the guy invented a significant part of the field.

By chapter

This section of my notes is organized more or less in parallel with the book, which is to say, it's not organized. The idea is to add a later synthesis section below (or above).

Things I'm hoping to see but might not:

- Flexures.
- Recursion.
- Control loop characteristics.
- Machining speeds.
- "Speeds and feeds" (coolant speed, current, voltage, vibration speed and amplitude, current waveforms, electrolyte choice).
- Alternating ECM with selective electrodeposition.

Preface

I'm surprised to see that you can electrochemically machine semiconductors (p. xviii), since they aren't held together by metallic bonds. I look forward to learning more.

"EMM", "EMST" ("electrochemical microsystem technology"), and "ENT" ("electrochemical nanotechnology") are new terms to me. "SECM" is mentioned next to "STM" and "AFM", and might mean "scanning electrochemical microscopy".

The English of the preface is lamentably somewhat broken, but I guess Bhattacharyya earned his merit by building things, not writing English poetry, and he chose to publish his book through Elsevier instead of a publisher who actually has editors. But it means the next couple hundred pages will be a bit of a slog.

At the end of the preface there's this tantalizing note:

> To assure that the reader is exposed to wider coverage of EMM, the book includes EMST and ENT for updating further applicability of anodic dissolution or deposition which promises significant advances not only in micromachining but also for nanofabrication as well as nanotechnology applications.

1. Introduction

The English gets worse:

> In prehistoric age, fragments stone, bone and wood were first used as tool by human beings for shaping the material to fulfill their urgent needs of day-to-day life. Progress in machining technology started from those early days. It was in about 4000 BC that use of drilling and cutting tools started in ancient Egypt...

Clearly there's some imprecision being introduced here due to the

poor editing, because that would imply that ancient Egypt lacked cutting tools such as hand axes for the previous two million years.

Ugh, and on p. 3 he misspells the title of Drexler's book as "*Engine of Creation*".

It's astounding to see him equate (also p. 3) top-down nanotechnology with nanometer-scale subtractive manufacturing. (He repeats the error in his Figure 18 diagram on p. 22, where his "bottom up approach" category includes "Rapid prototyping (RP)", by which I assume he means 3-D printing, CVD, PVD, electroforming, and "electron beam direct writing".)

On p. 6 he's careless about scale, suggesting that the universe is on the order of 10^{15} meters in scale, when actually that's only about 0.1 light years, so it's too small by 11 orders of magnitude.

Page after page of this carelessness is making me wish I was reading *Nanosystems*, which has the unfortunate drawback that none of the systems it describes have been built.

This is a boner too (p. 7):

> When the size of the microcomponent becomes smaller to atomic scale, it is not possible to utilize the top-down approach. However, developments are taking place to improve some of the techniques such that machining and fabrication can be successfully made at the molecular level and may be extended even to subatomic scale.

Oh *really*.

On p. 8 we have a taxonomy of micromachining processes: TBM, USMM, AJMM, AWJMM, WJMM (mechanical micromachining); EBMM, LBMM, EDMM, EDMM again, IBMM ("thermal beam based micromachining"); PCMM and EMM (chemical and electrochemical micromachining); and ECSMM, ECG, EDG, and ELID (hybrid micromachining). No expansions are given for any acronym.

On p. 9 we have the first useful assertion: that the cutting-edge radius on micro-scale conventional tools can be "up to 10 micron", with a diagram showing the removal of a chip that's much thinner than the tooltip radius. "The main drawbacks of this process are high tool wear, rigidity requirement of the machine tool, and heat generation at the tool-work interface." I'd've thought the surface finish would be a bigger drawback!

I think AJMM, AWJMM, and WJMM are supposed to be abrasive jet micromachining, abrasive waterjet micromachining, and waterjet micromachining. However, he seems to have forgotten to cover AWJMM and WJMM, and TBM is not mentioned but maybe means micro-scale conventional cutting tools.

Micro-USM (p. 10) ultrasonic machining might be the meaning of the enigmatic "USMM" on p. 8. P. 10 also has a nice (though badly pixelated) diagram of abrasive-jet machining at 0.2-0.8 MPa with 500-1000 nm abrasive particles and a photoresist film to abrade selectively, none of which is mentioned in the text.

On p. 11 he says boron carbide is "often chosen as the abrasive [for ultrasonic machining] for almost all materials except diamond due to its cost effectiveness and ease of use." I wonder if maybe it's actually boron nitride, which he hasn't mentioned.

There are some useful figures on USM on p. 11: abrasive grain size from 200 to 20000 nm, 100 to 20000 nm vibration amplitude, 0.1 to 1 N force, and 20–40 kHz (p. 10). This information is very valuable, but I wonder if it's as unreliable as the other information presented previously.

"EDG" is "electrodischarge grinding" (p. 12).

"LBM" is "laser beam machining" (p. 12), so maybe LBMM is "laser beam micromachining".

"PAM" is "plasma arc machining" (p. 13), which wasn't mentioned in the taxonomy diagram; maybe "PCMM" is intended to mean it. The diagram is just a more pixelated version of a standard plasma cutting torch.

"IBM" is "ion beam machining" (p. 14) but the diagram is actually a diagram of e-beam machining (the next section) because it contains no ion source. I'm guessing "IBMM" is "ion beam micromachining", although, really, ion-beam machining pretty much has to be "micro" in order to be useful at all.

"EBM" is "electron beam machining" (p. 14), so maybe "EBMM" is that too.

On p. 15 he talks about "micro-CM" or "chemical micromachining (CMM)" which he describes as the way "microdevices like semiconductor devices, ICs, etc.," are made. I don't think I've ever heard chip fabrication called "CMM" before.

Aha! On p. 18 there's a chart of speed-and-feed stuff explaining what sets EMM apart from regular ECM. It answers immediately one of the things that's been puzzling me about ECM, namely, why people haven't been using it to make ridged mirrors: accuracy of ±0.02–0.1 mm, down to 0.01 mm for EMM, which is two orders of magnitude worse than what you need for optics.

2. Electrochemical machining: macro to micro

On p. 26 he gives a historical overview, which I'm hoping is actually accurate:

> In 1929, the Russian researcher W. Gusseff first developed a process to machine metal anodically through electrolytic process. In 1959, Anocut Engineering Company of Chicago established the anodic metal machining techniques as a commercially suitable technique. After 1 year, Steel Improvement and Forge Company followed with a commercial application of this technique, based upon research by the Battelle Memorial Institute. The technique was applied mainly for machining of large components made of advanced and difficult-to-cut metals in the 1960s and the 1970s, particularly in the gas turbine industry. Electrical discharge machining at that time was a more accurate technique and was preferred over ECM, because ECM was less accurate and its waste is hazardous to the environment. But ECM was able to achieve much higher machining speed.

It would be nice to get some specific information about the environmental hazards so we can mitigate them. If this was a major industrial consideration in the 01960s they must have been *amazing* environmental hazards; that was the period when they were investigating open-cycle nuclear-powered jet engines, chlorine trifluoride rocket fuel, and borate zip fuel, and the EPA and Superfund didn't exist yet.

On p. 26 his account of the effects of changing the process gap implicitly assume a constant-voltage source.

On p. 27 his diagram suggests that the standard way to separate the sludge is with a centrifuge.

On p. 28 there is a list of four major recent improvements: vibrating axes permit maintaining a 100-micron process gap (because the electrolyte can flush out during the other part of the vibration cycle); pulsed current rather than constant direct current; microfiltration for electrolyte regeneration; and CAD for cathode tool profiles. After having waded through 40 pages of tiresome sales pitches for ECM I sure hope the book explains how to do these things at some point.

On p. 29 there is a fundamental error:

> During electrical conduction through electrolyte … Distribution [sic] of anion and cation remains uniform; hence the electrical potential at all points in the electrolyte is also uniform. Application of an external electric field causes migration of one ion species with respect to other. [sic]

When the electric potential at all points in the electrolyte is the same, no current flows. I hope this is just a careless error and not something he really believes. (In Fig. 2.11 on p. 42 he gives a correct, if only qualitative, diagram of how the electrical potential differs at different points in the electrolyte.)

On p. 30 he gives Faraday's constant as "96.485 C mol^{-1}", which is correct if we read the "." as a thousands separator.

I'm not sure about his description of electrolysis on pp. 30–33. I need to come back and reread it. But at least it's real information instead of the sales pitch.

On p. 33 the concept of chemical equilibrium is incorrectly contrasted with a description of chemical equilibrium:

> When no current is flowing, the electrochemical changes occurring at an electrode are in steady state, i.e., atoms leave the electrode and become ions and the ions move to the electrode and becomes [sic] atoms. The process continuous [sic] till [sic] equilibrium is reached. A potential difference exists between electrode-electrolyte interfaces [sic], which is known as "electrode potential."

On p. 34 there is a description of different anodic dissolution regimes (pitting, polishing, both) that I need to reread.

Topics

- Materials (p. 1138) (59 notes)
- Electrolysis (p. 1158) (18 notes)
- ECM (p. 1186) (9 notes)
- Reading (p. 1244) (4 notes)

Chording commands

Kragen Javier Sitaker, 02021-11-26 (updated 02021-12-30)
(7 minutes)

Emacs has some key commands involving pressing multiple keys at once that are sometimes described as "chord commands", which can be pretty inconvenient to type. I'm using C-M-v to scroll down the other window, for example, and M-{ and M-} to move by paragraphs, which require the shift key. I also use M-< and M-> to move to the beginning or end of file pretty often. M-^ (M-shift-6) joins lines together, which I do regularly. M-% (M-shift-5) is search and replace, which I use pretty often. M-| (M-shift-\) passes the region to a shell command. C-M-w does "append-next-kill", which is occasionally useful. M-: (M-shift-;) is the eval-expression command, which I probably use as often as M-x. C-@ and C-_ are common commands that would be similarly inconvenient, but fortunately C-SPC and C-/ do the same thing. I occasionally use C-M-left and C-M-right to move over parenthesized expressions.

Key sequences like the infamous C-x 8 RET, C-x 8 _ a, C-x RET C-\, and so on, are also inconvenient; of course, like the chords, they're hard to discover; but also you have to type them in the right order, which slows you down, it's a real pain to do one of them repeatedly, and they amount to a short-lived modal interaction, which causes mode errors.

So in some ways the chords are preferable, but they cause repetitive stress injury. Also, in one way, even the chords are not real chords: on a piano it doesn't matter if you hit the G key 4 milliseconds before the C key or 4 milliseconds after, but Emacs definitely cares a lot about whether you press A and then Ctrl, or Ctrl and then A. So even the chords are slower than they need to be.

But there are about 132 keybindings in global-map and another 32 in esc-map, plus more commands provided by one mode or another (`apropos` finds 3685 commands currently loaded), and only about 88 keys on this keyboard, most of which normally have to be used for writing. Most of the ones that aren't for writing (Esc, F3, Insert, the arrow keys, etc.) are in very inconvenient places. So it would seem that inconvenient chords or key sequences are inevitable.

But can we do better?

The home row has 12 keys on it, if we omit the nonstandard position of \, and there are another 24 keys almost as easily reachable above and below, plus the space bar, the)} key, and on this keyboard the >< key, for a total of 39. In a spectacular feat of perversity, this doesn't include the number keys, Esc, Enter, Backspace, Ctrl, Alt, or the left Shift key. But it does include Tab and the worthless Caps Lock. (I usually press Alt with my spacebar thumb, so maybe we have 40 convenient keys.)

It occurs to me that a much more manageable sort of chord would be one where you simultaneously press some magic "command key" and one *or more* keys for the command. So, for example, Alt-Q might be one command (and should act identical regardless of whether you

press the Alt first or the Q first), and Alt-Q-O might be another command (the same command as Alt-O-Q). So any permanent effects of the command wouldn't take effect until you started releasing keys. (They could totally change your view, though, since that's reversible.)

Here are the keys conveniently reachable by each finger:

- left pinky (6): tab, capslock, <, q, a, z
- left ring (3): w, s, x
- left middle (3): e, d, c
- left index (6): r, f, v, t, g, b
- left thumb (2): leftalt, space
- right thumb: nothing except space
- right index (6): y, h, n, u, j, m
- right middle (3): i, k, ,
- right ring (3): o, l, .
- right pinky (6): (,), ;, ', /, rightshift (disregarding the nonstandard \)

If the left pinky is tied up with capslock, then there are $3 \times 3 \times 6 \times 2 \times 6 \times 3 \times 3 \times 6 = 34992$ possible capslock-based chord commands, and there's an even larger number of chords accessible with Alt as the command key, but we probably want to limit ourselves to chords that don't involve too many fingers, both due to human limitations and due to key jamming and ghosting. Even chords that involve only two fingers can be awkward; try Alt-E-X, for example.

A very simple sort of first-level command set is the number of chords that involve Alt, one key from the left hand, and one key from the right hand, since those are all guaranteed to be easy to type. There are 17 left-hand non-thumb keys (15 if we remove capslock and the Spanish-keyboard-only <) and 18 right-hand non-thumb keys, giving 306 commands. This is a promising number of commands, and I think that key jamming and ghosting won't be a problem at all on any reasonable computer keyboard with Alt plus two regular keys.

You'd think that in 02021 keyboards would handle at least three, but in fact even without Alt, this keyboard gets key jamming between Q and S when A is depressed, between A and W when S is depressed, between Z and S when X is depressed, and so on. Perhaps more alarmingly, FGH, HJK, CVB, and VNM also form such jamming triples, even though they're physically all in separate columns of the keyboard. It may not be a coincidence that these all include pairs of keys that are supposed to be pressed with the same finger (FG, HJ, VB, and NM respectively); the keyboard may be designed (using the term generously) for correct touch-typing technique. Still, it makes me worry that some common keyboard out there will jam on easy-to-type combinations like FJK.

I think that generally the difficult-to-type chords are difficult to type because they have fingers of the same hand two rows apart: QV, ZT, WC, XR, Y., but not QF, AV, ZG, AT, WD, SC, SR, XF, YL, or J.. I think this gives a large number of four-key chords in consisting of Alt with the thumb, one key or two keys in adjacent rows or the same row on the fingers of one hand, and one key or two keys in adjacent rows or the same row on the fingers of the other

hand. Crudely I guesstimate that this is about five thousand combinations.

Topics

- Human-computer interaction (p. 1156) (22 notes)
- Editors (p. 1257) (4 notes)
- Keyboards (p. 1289) (3 notes)
- Emacs (p. 1298) (3 notes)
- Chording

Exotic steel analogues in other metals

Kragen Javier Sitaker, 02021-12-01 (updated 02021-12-30)
(8 minutes)

Iron is a pretty economically attractive material: extremely abundant (5.6% of Earth's crust, 32% of Earth), moderately refractory (doesn't melt until 1600°), can form carbides called "cementite" that make it into a very hard and strong "alloy" called steel (arguably really a cermet), and can be heat-treated to increase its hardness further. But could we make "steels" based on other similar metals, if we had enough of them? I think we could.

Ridiculously oversimplified steel

Steel is a very complex system, and I am doing it a bit of an injustice by simplifying its behavior to just "derives its strength from cementite".

Regular iron cementite (Fe_3C) has a "hardness" of 7–11 GPa, which I take to mean that its ultimate tensile strength is 7–11 GPa. Jiang and Srivilliputhur calculate ideal tensile strengths for cementite between 15 and 30 GPa in different directions, though I haven't really read their paper. When steel changes phase from austenite to ferrite, the solubility of carbon in the iron phase drops greatly, precipitating submicron-thickness layers of cementite alternating with ferrite in a structure called "pearlite". (I don't think they're thin enough to be below the flaw-tolerant critical size, which is estimated at 30 nm for goethite fibers such as those found in limpet teeth, but which exists for any material; nacre actually uses 200–500-nm-thick crystals, similar to the thickness of cementite layers in some pearlite. The cementite layers in bainite might be thin enough to be flaw-tolerant.) As a consequence, pearlite wires can reach tensile strengths over 6 GPa. But cementite is unstable above 723°, so steels become soft and malleable when they transition to the austenitic phase.

Tungsten and the chromium group

Mushet steel, the nascent form of high-speed steel, includes 1.5–2.5% carbon, 4–12% tungsten, which also forms a hard carbide with carbon, and 2–4% manganese. As I understand it, the manganese makes it austenitic at room temperature, allowing it to be air-hardened without quenching and to have much greater toughness than earlier hardened steels. I think that most of the carbon in Mushet steel and similar high-speed steels ends up in tungsten carbide rather than carbides of iron or manganese, and that nearly all the tungsten does.

Tungsten carbide's tensile strength is normally cited as being only around 0.4 GPa, but because it's a brittle ceramic, I suspect that number is dominated by flaw-sensitivity. It doesn't decompose until 2800°, and I think this is why Mushet steel and the modern high-speed steels that are based on tungsten remain hard at high

temperatures.

I think you could probably make a "steel" consisting of tungsten and a little carbon. The solubility of small amounts of carbon in tungsten rises up to 2715°, so you could probably get some kind of interspersed pearlite-like microstructure by quenching tungsten down to a lower temperature. It wouldn't have to be anywhere near room temperature; quenching it to 1091° in molten magnesium or 907° in molten zinc would be just fine. (There's a second tungsten carbide, but it only becomes important at higher concentrations.) But at room temperature tungsten is kind of brittle, and of course tungsten itself is a very rare element (0.17 ppm of Earth by weight, 1.25 ppm of the crust), which sometimes matters.

Tungsten is a group-6 transition element, along with chromium, molybdenum, and the wildly radioactive and presently irrelevant seaborgium.

Chromium (4700 ppm of Earth, 100 ppm of Earth's crust) is a bit more refractory than iron (melting at 1907°), is the base of some stainless steels, and is commonly plated on top of steel to make it harder, shinier, and more corrosion-resistant. In many ways it's reasonable to think of it as a sort of half-assed tungsten. So what about its carbides?

Well, chromium has three carbides, which are indeed refractory (1895°, so less refractory than metallic chromium), very hard, and corrosion-resistant, and they're commonly used to improve the wear and corrosion resistance of metals. So far so good. I should look up the chromium–carbon phase diagram.

Molybdenum is damn near as refractory as tungsten itself, melting at 2623°, but also damn near as rare: 1.7 ppm of Earth, 1.2 ppm of its crust. Its oxide is a lot more volatile than tungsten's, limiting its refractory usefulness in applications exposed to air, but it also has a very hard refractory (2687°) carbide. I should look up the relevant phase diagram.

Covalent binder alternatives to carbon

So far, we've looked at carbides of iron, tungsten, chromium, and molybdenum, all of which are very hard and refractory due in part to the somewhat covalent character of their bonding. But there are other elements that can play a similar role: boron, oxygen, nitrogen, and sulfur, as well as oxoanions like phosphate. Some of these are counterproductive in iron itself: the oxides, nitrides, and sulfide of iron are all weaker and less refractory than iron itself. But iron boride is somewhat of a hit (Vickers hardness of 15–22 GPa, melts at 1389°, already in use as a steel ingredient for hardness and used for surface hardening), and iron tetraboride is superhard, and, in combination with other metals, some of these elements produce very interesting ceramics.

Sticking to just the metals so far mentioned, tungsten borides have Vickers hardnesses of 20–30 GPa and WB_4 is described as "an inexpensive superhard material" because you can make it with just arc melting from the elements; chromium borides are also very hard and strong and can be made by SHS, especially if you add aluminum; and molybdenum borides are predicted to be superhard but apparently

nobody has managed to produce them in volume yet.

As for the oxides, I've mentioned the iron oxide goethite above (not usually thought of as superhard, but the limpets manage); tungsten trioxide has 5–7 GPa hardness at 800°; chromium oxides include chromia (viridian) which melts at 2435° and has Mohs hardness 8 as the mineral eskolaite; and, though it melts at only 802°, molybdenum trioxide has a hardness of 18.7 GPa, though as a mineral it's only Mohs 3–4.

Nitriding, carbonitriding, and nitrocarburizing are commonly used as a surface hardening process for steel, chromium, and molybdenum, and nitriding has been used to harden iron since antiquity, with urine, leather, and hooves being preferred case-hardening ingredients. Tungsten nitride is also hard, but it decomposes in water, limiting its use in air-contact applications.

Many elements also have interesting oxynitrides, oxyborides, borocarbides, boronitrides, borocarbonitrides, carbonitrides, and oxycarbonitrides. Oxycarboborides and oxyboronitrides seem to be either neglected or too difficult to make, and although some "oxycarbides" are reported (including a molybdenum oxycarbide), many more are just carbonyls or oxalates, which are neither hard nor refractory.

Other metals

What about nickel, cobalt, vanadium, manganese, titanium, zirconium, hafnium, silicon, niobium, and tantalum? They also form carbides! That makes 110 more candidate ceramics to investigate!

Topics

- Materials (p. 1138) (59 notes)
- Strength of materials (p. 1164) (13 notes)
- Steel (p. 1222) (5 notes)
- Ceramic-matrix composites (CMCs) (p. 1265) (4 notes)

Simplest blinker

Kragen Javier Sitaker, 02021-12-01 (updated 02021-12-30)
(9 minutes)

There are lots of LED blinker circuits around, most driven by a 555 or a transistor-based astable multivibrator. But there are enormously simpler options.

In some sense the simplest LED blinker is two red LEDs in antiseries, in parallel with a capacitor and resistor, in series with another resistor, powered by a voltage around 9–48 V. That is, 9V-R1-(->|-|<- || R2-C)-GND. When the reverse-biased LED goes into avalanche discharge at *Vbr*, around 5 V, the capacitor discharges down to the minimal voltage necessary to sustain avalanche conduction in the backward-biased LED, after subtracting the forward voltage drop of the forward-biased LED, and with time constant R_2C, with a pulse energy $\frac{1}{2}Vbr^2C$, lighting the forward-biased LED. When the current drops too low to sustain avalanche conduction, the reverse-biased LED begins to block again, and the capacitor recharges toward the power supply voltage with time constant $(R_1 + R_2)C$.

(*Most* reverse-biased PN junctions have that kind of bistable avalanche behavior, and it's an annoying source of noise when using avalanche diodes as voltage references, but not all. Note that the plots in Infineon's appnote linked above do not exhibit this bistability; it is the source of MOS latchup. You could probably force such recalcitrant circuits into oscillation with sufficient series inductance: 9V-R1-L-(->|-|<- || R2-C)-GND.)

This is very similar to the basic neon lamp flasher, except for the R2 current-limiting resistor (which may not be necessary!), the lower voltage, the potentially much higher speed, and the complication that the LED that provides the circuit's bistability doesn't light (avalanche-mode LEDs do emit, but at two orders of magnitude lower radiative efficiency), so a second LED is needed.

In theory you could make R1 small enough that the LED would just stay lit until the avalanching LED burned out, but that seems unlikely in practice. And in theory you could leave out R2 and allow the LED to set its own current, since the energy of the pulse will be limited to the energy stored in the capacitor, but even arbitrarily short high currents can damage semiconductors through non-thermal damage mechanisms, so it might be better to leave it in.

Let's calculate some values so that this circuit is likely to flash visibly. The current through the LEDs should probably be around 20 mA to be brightly visible without much risk of damage, and I think the reverse voltage drop of an avalanching LED is pretty small, maybe around 1 volt. The forward voltage drop of the illuminated LED should be around 1.6 V, so we have about 5 - 1 - 1.6 = 2.4 volts across R2, which means something like 100Ω is appropriate, which would give us 24 mA. Let's shoot for about 1 Hz overall repetition rate. I suspect that even 1% duty cycle (10 ms) would be bright enough to be visible, but let's shoot for 50 ms (5%) to be safe.

At this point I realize I don't have any idea how much current is necessary to maintain avalanche conduction. If I pretend that the avalanche sustaining voltage and the LED forward voltage drop are constant with respect to current, and any amount of current is sufficient to sustain the avalanche, the result is that the capacitor voltage asymptotically approaches the $1 + 1.6 = 2.6$ V and never reaches it, and the circuit never turns off, so this is not a useful approximation. I'm sadder but no wiser after reading a couple of application notes about avalanche breakdown.

So, just guessing, maybe you need four RC time constants (54× lower current) to turn off. 50 ms ÷ 100Ω = 500 μF, so a 470 μF electrolytic ought to work. Then we just need to set R_1 to get an off-time of around a second.

The off-time is determined by the time to charge from the avalanche cutoff voltage (plus the forward-biased LED's voltage drop) up to the breakdown voltage on its way up to the source voltage: from 2.6 V up to 5.0 V out of 9.0 V, which is to say, the remaining voltage should drop from 6.4 V to 4.0 V, a factor of 0.625, about $\sqrt{e}$, so half a time constant. Thus a time constant of about two seconds would be right, which works out to about 4.3 kΩ, so a 4.7kΩ resistor ought to work.

The visual brightness of the LED should mostly depend on how much charge goes through it, not how long it takes. If we were going to discharge 2.4 volts out of a 470 μF electrolytic at a constant 20 mA, it would take about 56 ms, which is plenty long enough to see the LED flash. So I think probably the LED flashing will be visible. The energy of each flash is about 2.3 mJ, about half dissipated in the resistor, with the other half split almost equally between the two LEDs, so it's unlikely that the LEDs will be damaged by heating.

The reverse-biased capacitance of the LED is down in the picofarads, so the stored energy is in the dozens of picojoules, insufficient to damage it much.

So the final circuit is:

```
9V-4k7-(->|[red LED]-|<[red LED]-||100Ω-470μF)-GND
```

Such a circuit is sensitive to every electronically relevant aspect of its environment: temperature, voltage, light, EMI, and radioactivity. Its off-time is inversely proportional to the difference between the input voltage and the turn-off voltage, though its on-time varies little with that. The avalanche breakdown voltage increases with temperature (Infineon's appnote above says that in silicon MOS it varies about 5% per 100°), so its both its off-time and its on-time would increase with temperature; but the threshold voltage of the forward-biased diode also changes with temperature, and I think that voltage *decreases*, which may have a larger effect on the on-time (though it isn't immediately obvious to me which way). Both diodes are photosensitive, so the circuit can be triggered with a flash of light, and its frequency will vary with the ambient light level. EMI can also advance the timing of the oscillator, so under some circumstances it can phase-lock to weak signals coupled in capacitively or inductively; but even in the absence of EMI, avalanche breakdown is somewhat

random, perhaps because it detects radioactive decay as well.

An interesting question is whether this sort of behavior is useful for anything. Of course, if you can mostly isolate the circuit from variations in voltage, light, EMI, and radioactivity, you can use it to measure temperature; if you can mostly isolate it from variations in temperature, voltage, EMI, and radioactivity, you can use it to measure light; etc. But digital logic seems potentially more interesting.

In Implementation and applications of low-voltage Marx generators with solid-state avalanche breakdown? (p. 960) I've argued that such triggerable oscillators can be used as clocked logic elements. With ordinary LEDs you ought to be able to get up into the MHz, but with avalanche diodes built on an IC, tens of GHz ought to be attainable. Large resistances or capacitances on chip require large areas, but by running with a supply rail closer to the breakdown voltage and increasing the duty cycle from 5% up to maybe 20%, you ought to be able to use equal or nearly equal resistances for R_1 and R_2, so neither needs to be particularly large. On chip it might be reasonable to use $C \approx 20$ fF, $R_1 \approx R_2 \approx 2k\Omega$, with a frequency around 10 GHz.

Why would you use this kind of dynamic logic instead of regular CMOS? As I understand it, a regular CMOS flip-flop needs 6 (or 8) transistors, and a CMOS NAND gate requires 4. Such an oscillator requires a capacitor, two diode-connected transistors, and two resistors, so it might permit higher density than regular CMOS, but be less power-hungry than four-phase logic (see Snap logic, revisited, and four-phase logic (p. 115)). (But really that's not attacking the crucial aspect of CMOS density, which is routing.)

Topics

- Electronics (p. 1145) (39 notes)
- Pulsed machinery (p. 1167) (12 notes)
- Physical computation (p. 1208) (6 notes)
- LEDs (p. 1286) (3 notes)

Capacitive linear encoder sensors

Kragen Javier Sitaker, 02021-12-11 (updated 02021-12-30)
(7 minutes)

I was thinking about the kind of capacitive linear encoder sensor used by digital calipers today. It occurred to me that it's a wonderful way to increase the precision of machinery, stage by stage, and it ought to be straightforward to manufacture a bootstrapping version by hand.

The basic primitive is that you have a couple of "combs" with the same spacing on two different circuit boards. Each comb consists of a sequence of parallel wide conductive lines with wide spaces between them, which are all connected electrically together with some sort of conductor elsewhere. The boards are placed on top of one another, with the lines parallel, with a thin dielectric between them. When the two combs on the different boards are in phase, which is to say spatially aligned, there is a large capacitance between the two combs. When they are out of phase, so that the lines of one are adjacent to the spaces of the other rather than its lines, the capacitance is instead very small. By measuring the capacitance between such a pair of combs, you can detect when they come in and out of phase.

To get a usable linear encoder, you put three interleaved combs on one board instead of one. When the comb on the other board is coming out of phase with one of the three combs, it is coming into phase with a second one, so the total capacitance is always the same, or nearly so. You need only make a ratiometric measurement of the three capacitances to uniquely identify the phase relationship of the two boards.

To be more concrete, suppose the combs are 10 mm wide, 100 mm long, drawn on two fired-clay ceramic surfaces, and separated by a 50-micron-thick layer of paint with a relative permittivity of about 5. The area of each comb is about 300 mm², so we can calculate the peak capacitance we see when two combs are in alignment at about 270 pF, while the parasitic capacitance between two out-of-alignment combs is probably only about 1 pF. If we assume the graphite itself has a resistance on the order of a kilohm, then the circuit becomes primarily resistive above about 600 kHz. Above that frequency it becomes harder to detect small changes in the capacitance because the graphite's resistance hides them. (Measuring the time constant of a step function response is probably a more practical measurement method, whether with analog or digital circuits.)

If we can reliably measure an 0.1% ratiometric change between two of these capacitances, for example with an ENOB-10 ADC, we should be able to measure the phase to about a milliradian. So if the lines on the combs are 2 mm wide, separated by a 1 mm space, and thus have a full cycle every 9 mm, a radian is 1.4 mm, and a milliradian is 1.4 microns, maybe 350 times more precise than the manufacturing precision required for the original combs. However, this is more precise than is realistic, as will be seen below.

This setup is particularly appealing because of how many forms of

manufacturing error it tends to cancel out or average out. It really only cares about the surfaces being flat or cylindrical (so there's a constant separation), the right total length, and having the right number of lines distributed over that length, so that the *average* spacing of a comb is correct. If all the teeth on one circuit board or the other are too big or too small, that affects the absolute capacitance but almost doesn't affect the ratio in a given position at all.

Random errors in the shape of a single comb tooth are averaged out across all the comb teeth; in the above setup, there are 33 total teeth on the interleaved plate, 11 in each of the three combs, so this doesn't help that much: an 0.5 mm standard deviation in edge location averages out to an 0.15 mm standard deviation in average edge location. However, with larger numbers of teeth, this is very powerful. Consider, for example, 30-micron-wide teeth separated by 15-micron gaps, presumably with a thinner dielectric. In 100 mm, there are 2220 total teeth, 740 per comb, so a 1-micron standard deviation in edge position for each edge works out to 34 nm standard deviation for the average.

So in practice it can only give errors on the order of 30 times smaller than the average local manufacturing error, not 350 times smaller; to get to 100 times smaller, you would need ten thousand teeth per comb. And the *global* manufacturing error is only slightly diminished: if you make the nominally 100-mm-long combs 101 mm long instead, or if thermal expansion makes them grow by that much, then all your displacement measurements will be off by 1%. Still, a 1% error in a 1-mm displacement is only 10 microns.

(1% error in steel or concrete at 12 ppm/K is 830°. In a 100 ppm/K material like many plastics, it would be only 100°.)

By putting such a comb around the outside of a circle rather than on a linear slide, you can get a pretty good relative angular measurement rather than a displacement measurement (a rotary encoder rather than a linear one), and you also avoid the variation in amplitude as more or less of the boards overlap. A 100-mm-diameter circle is 314 mm in circumference. If we have 314 teeth per comb, making each tooth about 300 microns wide, and the random error in each tooth edge has a standard deviation around 50 microns, the resulting random error for the whole dial will be about 2.8 microns, which is 56 microradians. This is an error well under a micron for things near the center of the dial, at which point the uncertainties in the bearings are a bigger source of error than anything in the electronic realm or in the ruling of the dial.

At a smaller scale, such sensors should provide even better precision. If you have a 10-mm-long comb with 10-micron tooth spacing, 30 microns per comb, then you have 999 total teeth, 333 per comb, and the necessity to position the combs within a micron or two without touching. If the standard deviation per tooth is 1 micron, then the standard deviation of the average should be 55 nanometers. Capacitive and micromagnetic feedback systems can measure smaller displacements (for decades now) but over a much shorter total travel.

See also tiltmeter.

Topics

- Electronics (p. 1145) (39 notes)
- Ghettobotics (p. 1169) (12 notes)
- Bootstrapping (p. 1171) (12 notes)
- Precision (p. 1183) (9 notes)

Two finger multitouch

Kragen Javier Sitaker, 02021-12-11 (updated 02021-12-30) (3 minutes)

If you have some slips of paper on a table, you can translate and rotate them around the table with two fingers. But you can't move the fingers further apart without tearing the paper or slipping on it, and if you move them closer together, the paper buckles and maybe wrinkles.

In typical early multitouch demos, this two-finger drag gesture was used to arrange photos or other objects on a surface, resizing them in the process. I've also seen people use it in SnapChat to place text on a photo while resizing it. Mostly it seems to have fallen out of favor, though.

What if you had an infinite canvas on which you could rotate and move objects, like these slips of paper, but also interact with them by moving your fingers nearer and further? For many objects it isn't that useful to expand and contract them, or even to rotate them. You could use these extra degrees of freedom in the two-finger touch to control one or two continuously variable parameters of the object or to invoke actions on it.

The single-finger drag has become an idiomatic way to scroll in multitouch environments, and single-finger taps either select an object or invoke a button. So a two-finger drag as a universal way to move movable objects seems like a promising interaction paradigm. It potentially conflicts with the idiomatic pinch-zoom interaction, but I think that you can still support zooming by pinching on the background.

What these objects should be or do depends on the application, but one of the most appealing features of a potentially universal UI paradigm like this one is that it permits the modeless coexistence of objects from different applications.

Container objects can do a few different things. The simplest thing they can do is to move the objects they contain when they are themselves moved, maybe rotation too. If they support a copy operation, they can copy the objects they contain in the process; similarly for a hide operation, perhaps provided by a tabbed view widget. They can do layout for the objects they contain, for example radial or table layout. They can also publish and subscribe to communicate with the objects they contain; those inner objects can then be used as tools to alter the outer object, or vice versa.

Tool objects that can alter other objects are a major way to extend this approach, since without interaction between objects, each object is limited to only two parameters or some sort of menu system. But if you can target a tool object at it, for example bringing it under some sort of magnifying glass or selecting it to bring up its attributes as separate on-canvas objects, you have arbitrary freedom.

Topics

- Human-computer interaction (p. 1156) (22 notes)
- Composability (p. 1188) (9 notes)
- GUIs (p. 1216) (6 notes)
- Multitouch

The Habitaculum: a modular dwelling machine

Kragen Javier Sitaker, 02021-12-13 (updated 02021-12-31)
(16 minutes)

Suppose we have a bunch of Lego-like bricks that can be attached together to make a temporary or permanent dwelling space. They might include LEDs for lighting, 240VAC outlets, speakers, mattresses, cabinets, drawers, carpets, mirrors, ovens, stove burners, air conditioners, refrigerators, toilets, etc. If the size and connectors are standardized, they can be composed in a variety of different Minetestish ways to make temporary or permanent furniture or dwelling spaces, perhaps inside of an existing suboptimal dwelling space.

You could imagine locking together a few dozen such independently wheelable blocks into a modular “dwelling machine” or “habitaculum” inside an arbitrary apartment, perhaps occupying some 10 m² of floor space and containing bed, desk, and shower, standing off the floor on short legs.

(I read a book from the 01970s recently that mentions in passing that the author’s friends did something like this in their shitty apartment in La Boca in Buenos Aires. I think it might have been Hennessey and Papanek’s *Nomadic Furniture 2* or Papanek’s *Design in the Real World*.)

Some of the deficiencies of existing dwelling spaces that can be thus remedied:

- Inadequate air conditioning. Here in Argentina many windows have roll-down shutters above them which permit wind to blow right through, and the regulations require 100 cm² openings in the walls where gas appliances are installed to prevent gas buildup. The resulting quantity of airflow on a hot windy day makes adequate air conditioning very difficult to achieve. A small, relatively sealed inner space without gas appliances can be free of such unwanted air leaks and can be air conditioned much more cheaply.
- Air filtering. The covid pandemic has made us aware that our indoor air is generally of very poor quality, and that this is a major risk not only for contagion of respiratory diseases like covid, but also for health in general. One of the biggest problems, especially in big cities like Buenos Aires with lots of high-sulfur diesel, is particulate pollution; pollen and mold spores are problems for sensitive people even in other places. Again, this is a difficult problem to solve without being able to limit the ingress of bad air, but an easy one to solve with controlled airflow.
- Thermal insulation, which makes air conditioning much more reasonable. Most buildings here are made of concrete with single-paned windows. The 10 m² habitáculo suggested above might have 45 m² of surface area (25 m² of walls and 20 m² of ceiling and floor). If this is all sheathed with 100 mm of 0.3 W/m/K insulating material, maintaining a 20° difference through the wall (e.g., from 21°

within to 41° without, or 21° within to 1° without) requires 2700 W of heating or cooling power; with a typical CoP of 3, this is an air conditioner of only 900 W. Thicker insulation than this may be feasible, cutting the required power further, and usually the temperature delta is much smaller.

• Storage. Many apartments and houses do not have much storage built in, being designed for consumers or socialites rather than craftspeople or the self-sufficient.

• Flooding. I just found a roll of toilet paper that had been water-damaged under the sink today; I guess the pipes are leaking again. I've had my clothes, neatly stacked in the closet, turn moldy from water leakage from a poorly constructed bathroom. When it rains, the roof here has leaked in two different rooms, creating big puddles on the floor before the landlord fixed the roof. In the apartment where I lived in La Boca, there was such a big hole in the roof that there were mushrooms growing down from the ceiling. Sometimes there's widespread flooding when rains outpace the storm sewers, although here in Buenos Aires the last big floods were about 10 years ago before they fixed the storm sewers. Most commonly only a centimeter or two of water gets onto your floor, but that's enough to ruin your computer, short out your power strip, ruin your mattress (if it's on the floor), destroy your tatami, and demolish your books (if they're on the floor). All of this can be avoided if everything is up off the normal floor, especially if the habitaculum itself is watertight.

• Lighting. Most buildings are inadequately lit (50–500 lux), which possibly causes seasonal affective disorder, widespread depression, and sleep phase disorders. Full daylight is 10–25 kilolux in the shade, about 100 times brighter, and direct sunlight is 100 kilolux. A lux is a lumen per square meter; to illuminate a black room to 1 kilolux requires a kilolumen per square meter, while a white or mirrored room may be about five times as efficient. For our example 10 m² habitaculum with 45 m² surface area, if we paint it white, we might need 9 kilolumens per kilolux. Both Samsung and Philips are shipping 200-lumen-per-watt LEDs since 02017, ordinary fluorescent tubes can hit 100 lm/W, ordinary LED bulbs are in the neighborhood of 75 lm/W, and sunlight is 93 lm/W. So reaching 10 kilolux might require 90 kilolumens and thus 450–1200 watts powering the lights, or less in a smaller area, and about US$20 worth of fluorescent tubes or US$500 worth of LEDs.

• Mosquitoes. In the Boca apartment with mushrooms growing from the ceiling, we constantly had mosquitoes because, as with most Argentine houses, there were no screens on the windows. (For a while I dated a woman who had moved into an apartment in terrible condition with screens on all the windows. As part of her renovation, she removed the screens because she didn't like how they looked.) Excluding mosquitoes from a small space without windows is easier than excluding them from a large space with windows.

• Spaces totally without amenities. There are numerous industrial and storage spaces, both in Argentina and in the rest of the world, that have electrical power, or can get it, but lack running water, climate control, floors, sewage treatment, and so on. These spaces are often much cheaper to rent than more livable spaces in the same area. An easily portable modular habitaculum could provide a convenient

alternative to a camping trailer.
• Noise. If you have many centimeters of some kind of sound-absorbing material between you and the outside world, you will be less annoyed by the superchargers on buses interrupting your conversation.
• Portability. Moving out is a huge pain in the ass. This was a major motivation for Papanek's *Nomadic Furniture and _Nomadic Furniture 2.*

• Coziness. A small space can be much more comfortable than a large one if properly organized.
• Saving wear and tear on the property. When you move out, it's nice if you don't have to repaint.

Standard residence doors range from 625 mm up to maybe 900 mm in width, and from 2000 mm on up in height. Standard residence ceilings are about 2.6 m high, though occasionally there are small areas with lower ceilings; you need at least 2 m of height inside the habitaculum for most people to stand up comfortably, and probably more like 2.2 m. The oppressive feeling of such a low ceiling can be relieved by covering most of it in mirrors, doubling the visual height of the space to 4.4 m.

For individual modules to fit through a standard door under the power of a single person, they ought to have wheels built into them or trivially attachable, be no wider than 500 mm (to permit passing through a 620-mm doorway at up to a 36° angle), be no taller than 2000 mm (they can be tilted over to go through low doors), and be no longer than 1000 mm in order to be manageable in size for a single person. This ½ × 2 × 1 m size is roughly that of a stack of three single-bed-sized mattresses, or half of two Minetest blocks. To be comfortably pivotable on a corner, they ought to weigh no more than 80 kg, so you never have to lift more than 40 kg; a weight of 40-60 kg, or even less, would be much better.

This presumes that the modules should have the shape of a rectangular paralellepiped. While aesthetically and in terms of structural strength this might leave something to be desired, walls are traditionally vertical so as to be in a local gravitational stable equilibrium, and floors, beds, counters, and tables are necessarily horizontal. So polyhedra of higher degrees of symmetry like the Platonic dodecahedron, or of lower numbers of connection points like the tetrahedron, would seem to be impractical.

If your wall modules are exactly 2 m tall, your ceiling or floor module will need to have some extra space to get up to 2.2 m inside space; 2.0 m is rather cramped.

So a minimal construction set for a 3.5 m × 3.5 m space (12.25 m²) might consist of 16 outer wall modules (0.5 m × 2 m × 1 m), 4 ceiling corner modules (0.25 m × 2 m × 1 m), 4 non-corner ceiling modules (same), and another 8 floor modules, 32 modules in all, occupying some 24 m³ in storage and enclosing 3.5 m × 3.5 m × 2.2 m = 26.95 m³. Some of the modules might contain Murphy beds, others closet space, others air conditioners.

The minimal volume to merely enclose that space would of course be much smaller, consisting of thin cubicle-like panels (or panels similar to Symons forms), perhaps only 20 mm thick. That would

reduce the total volume when disassembled to something like 0.64 m^3.

The modules could connect together at the corners with twistlock-like connectors, modified to permit watertight connections between them, or with wedge connectors like the "wedge bolts" that connect concrete-form panels. Probably the walls will not have to withstand more than 2 kN of force, which probably will not have more than a 4:1 mechanical advantage against the corner connectors (spread across two corners), so with 250 MPa A36 mild steel connectors you only need about 16 mm^2 of tensile cross-section per connector to keep the modules together. So it ought to be easy enough to use snap connectors like those used in seatbelts. However, connectors should not protrude from the surfaces when not in use (that would make the modules uncomfortable to sit on or lean against), and because you might want to disconnect a module that's currently connected to two or more other modules, it needs to be possible to disengage the connections while the modules are still in contact; turning a knob should enable you to engage or disengage them. (Knobs are less likely to be actuated unintentionally than buttons, levers, and the like.)

The modules could probably be built as welded angle iron frames, plus panels to give them rigidity. Each wall module has 14 meters of edges (4 × (½ + 1 + 2)). Metals Depot will sell you hot rolled A36 12mm × 12mm × 3mm angle iron for US$5/m, and they say it weighs 570 g/m, so this would work out to 7.9 kg and US$70, which is a lot less than 80 kg but a bit pricey. 18-gauge A513 12mm × 12mm square tube is a little lighter but costs more than twice as much. They also sell ASTM A527 18-gauge (1.3 mm) galvanized steel sheet for US$73/$m^2$, saying it weighs 10.5 kg/m^2, which is a slightly lower price per kg.

On Mercado Libre, Almacen Techista here in Buenos Aires sells 30-gauge ("C30") galvanized (0.3 mm) for AR$2100 for a 1×2 m sheet, weighing 5.05 kg. At today's quote of AR$194.50/US$ that's US$10.80, US$5.40/m^2, or US$2.14/kg, a dramatically lower per-kg price than Metals Depot. They also sell 18-gauge cold-rolled (1.25 mm) for AR$6050 for 1×2 m, which would be about 20 kg, US$31, and US$1.58/kg, which is cheaper than the galvanized (at Metals Depot cold-rolled is dearer than hot-rolled, which is dearer than galvanized).

Also on Mercado Libre, Gramabi sells 20 mm × 20 mm × 1.25 mm square steel structural tubing for AR$1600 for 6 m, US$8.22, US$1.37/m. I think that's 1.19 kg, so US$6.90/kg, which is more similar to the Metals Depot prices. But this might be a better grade of steel than the galvanized roofing sheet metal.

Some of these modules include things that can be controlled or monitored electronically without a lot of bandwidth, such as LEDs, fluorescent lights, air conditioners, heat exchangers, speakers, or thermometers. If each face of a module where it can be latched onto another module has an electronic connector as well, and the microcontroller in each module can identify its module type, distinguish between the different connectors, and detect their orientation, then they can build a constantly updated model in

memory of the whole assemblage of modules and all of their hardware, permitting it to be controlled and monitored from a larger computer.

It's maybe not necessary for the modules to have electronic (and electrical power) connectors on every surface. Legos only have mechanical connectors on two of their surfaces, only four connectors in the simplest general case; tee connectors have only three ports and are capable of being assembled into two- and three-dimensional networks. (Think of a cube that can couple to other cubes on three of its faces.)

This idea seems really cool: you could snap together some modules into a custom ad-hoc dwelling machine that is immediately reflected in the mirror world inside your laptop, from which you can control it.

Interestingly, although I had forgotten this, *Nomadic Furniture 2* profiles a system of modular cubes called "Palaset" by Ristomatti Ratia, which is in some ways very similar. They're polystyrene cubes designed for storage (shelving, drawers, cabinets) that you can also sit on, with a 13½" outside size and ¼" walls (345 mm and 6 mm), linked together with doublestick tape and by inserting asterisk-shaped plastic "dowels" into holes on their faces. Some of these are evidently still available 49 years later at palaset.com for about 20 euros each, though perhaps not as many as were profiled in the book.

The ClicBot educational robot kit works more or less the way I describe above, with sensors and connectors to create a VR model of the physical robot that you can then program, but it's for building tabletop robots, not dwelling-machines.

Topics

• Composability (p. 1188) (9 notes)
• Life support (p. 1251) (4 notes)
• Household (p. 1363) (2 notes)

Against subjectivism

Kragen Javier Sitaker, 02021-12-15 (updated 02021-12-30)
(36 minutes)

XXX note that fallibility is mentioned in the fallibility section and also the section on people confusing social constructions with real things

In a discussion recently, I saw someone say (slightly edited for readability and formatting):

> *The* truth? Engage in discussion in a productive way? These really don't seem to be my things. … I'm not interested in discussing things with people who believe in *the* truth (or that they *know* it). … In my opinion, truth is being *established* between people, within communities. Truth is not a given, it does not exist outside of a social context and it certainly does not inhabit any discussion before common ground is found.

I want to emphasize that this is not a straw man; it is a literal transcript of something that someone actually said to me, improbable as it sounds. Therefore I think it is worthwhile to clearly explain why it is untrue.

It struck me as a particularly clear statement of a malignant doctrine, a collective variant of metaphysical solipsism, that I've often seen in a more covert form. It's a hopelessly confused metaphysical doctrine which eliminates both the possibility of rational action and the possibility of any basis for agreement, other than submission or compromise; scholars call it "global alethic relativism".

The truth in global alethic relativism

First, though, let me explain what is *true* about this view.

People cannot distinguish social constructions from objective truth

There are *many* statements that people commonly accept as "true" that are in fact socially constructed rather than objectively true in any sense: my property extends up to such-and-such a line; masturbating in public is unacceptable; *The Da Vinci Code* is not yet out of copyright; saying that someone is "nice" is saying that they treat others well. Clearly the community could change its collective mind about where the property line is drawn, what sorts of behavior are acceptable and unacceptable, what causes bizarre human behaviors, the terms of copyright, and the definitions of words.

People nearly always act as if things like national boundaries, corporations, ghosts, demonic possessions, marriages, and land tenure rights *exist* in the same objective, material sense that stone walls or puppies do. This is very practically useful: you cannot run through a stone wall by disbelieving in it, and in the same way you cannot run through a heavily armed international border by disbelieving in it — even though the border is socially constructed, the armaments and the border guards wielding them are objectively, materially real.

But, in fact, when it comes to such things, there is indeed no such thing as *the* truth; the truth of a marriage, a corporation, a national

border, or a law is established between people, within communities. It does not exist outside of a social context and does not "inhabit any discussion before common ground is found." It is a shared opinion, not a fact about the material world.

The relevant difference is that if *the border guards* stop believing in the national boundary, or can be tricked about where it is, then you can cross it without changing anything in the material world other than people's beliefs and expectations. This does not work with stone walls.

People are fallible

Everyone has had the experience of being wrong about things, and dialogue with other people is usually the way we find out we are wrong — in cases like land tenure rights and national boundaries, *some* sort of communication with other people is the *only* way to find out, because those things don't physically exist in the material world, except as beliefs in people's minds. Everyone has also had the experience of encountering someone who is unwilling to consider the possibility that they are wrong.

Everyone has also had the experience of having beliefs that they *thought* were objective truth, which turned out to only be opinions — usually because they met someone with different opinions.

Given this background of experience, how can we justify any belief at all in *the* truth, or a truth that exists outside of any social context? How do we know that we aren't just confusing ourselves again, switching from one set of beliefs to a more popular one, without ever making any contact with objective reality?

People's understanding of the objective world is filtered through social constructions

A different aspect of this proposition is that people are not mentally equipped to grapple directly with the immense mass of brute facts in the objective world, so they retreat to what Walter Lippmann terms "fictions":

> Now in any society that is not completely self-contained in its interests and so small that everyone can know all about everything that happens, ideas deal with events that are out of sight and hard to grasp. Miss Sherwin of Gopher Prairie is aware that a war is raging in France and tries to conceive it. She has never been to France, and certainly she has never been along what is now the battlefront.
>
> Pictures of French and German soldiers she has seen, but it is impossible for her to imagine three million men. No one, in fact, can imagine them, and the professionals do not try. They think of them as, say, two hundred divisions. But Miss Sherwin has no access to the order of battle maps, and so if she is to think about the war, she fastens upon Joffre and the Kaiser as if they were engaged in a personal duel. Perhaps if you could see what she sees with her mind's eye, the image in its composition might be not unlike an Eighteenth Century engraving of a great soldier. He stands there boldly unruffled and more than life size, with a shadowy army of tiny little figures winding off into the landscape behind.
>
> ...
>
> In all these instances we must note particularly one common factor. It is the insertion between man and his environment of a pseudo-environment. To that pseudo-environment his behavior is a response. But because it is behavior, the consequences, if they are acts, operate not in the pseudo-environment where the behavior is stimulated, but in the real environment where action eventuates. If the

behavior is not a practical act, but what we call roughly thought and emotion, it may be a long time before there is any noticeable break in the texture of the fictitious world. But when the stimulus of the pseudo-fact results in action on things or other people, contradiction soon develops. Then comes the sensation of butting one's head against a stone wall, of learning by experience, and witnessing Herbert Spencer's tragedy of the murder of a Beautiful Theory by a Gang of Brutal Facts, the discomfort in short of a maladjustment. For certainly, at the level of social life, what is called the adjustment of man to his environment takes place through the medium of fictions.

By fictions I do not mean lies. I mean a representation of the environment which is in lesser or greater degree made by man himself. The range of fiction extends all the way from complete hallucination to the scientists' perfectly self-conscious use of a schematic model, or his decision that for his particular problem accuracy beyond a certain number of decimal places is not important. A work of fiction may have almost any degree of fidelity, and so long as the degree of fidelity can be taken into account, fiction is not misleading. In fact, human culture is very largely the selection, the rearrangement, the tracing of patterns upon, and the stylizing of, what William James called "the random irradiations and resettlements of our ideas." [Footnote: James, Principles of Psychology, Vol. II, p. 638] The alternative to the use of fictions is direct exposure to the ebb and flow of sensation. That is not a real alternative, for however refreshing it is to see at times with a perfectly innocent eye, innocence itself is not wisdom, though a source and corrective of wisdom. For the real environment is altogether too big, too complex, and too fleeting for direct acquaintance. We are not equipped to deal with so much subtlety, so much variety, so many permutations and combinations. And although we have to act in that environment, we have to reconstruct it on a simpler model before we can manage with it. To traverse the world men must have maps of the world. Their persistent difficulty is to secure maps on which their own need, or someone else's need, has not sketched in the coast of Bohemia.

So these "fictions" or "pseudo-environments" are products of social contexts and evolve from the interplay and negotiation between people. Lippmann's "fictions", however, have "degrees of fidelity": they are referred to an underlying environment, and may be better or worse at it, so they can be more or less true in ways that have nothing to do with social consensus.

Why alethic relativism is false

But this extremist form of relativism does not stop at pointing out that people commonly accept social constructions as facts, are fallible, and understand their environment in a way that is profoundly altered and simplified by their social context; it calls into question even propositions such as whether there is a stone wall in front of you, the commutativity of integer multiplication, or whether Elvis Presley is still alive.

The obvious problem is that it affirms nonsense

At first glance it would seem to foreclose only the possibility of any basis for agreement other than mere popularity. It entails that geocentrism was actually true until Copernicus and Galileo made it unpopular, and heliocentrism was false, because the community of the Catholic Church had established the truth of geocentrism. It argues that whether there is or is not a stone wall in front of you is a truth that can only exist inside of a social context, established between people, within communities; the right social context would allow you to run through the stone wall as easily as you can run across a state line in the US. It entails that exterminating sparrows and deep plowing in the Great Leap Forward was to result in abundance rather

than famine, because the social context had established that it would. It asserts that the only difference between truth and lies is that people disbelieve lies.

That is, it cannot distinguish between truth and collective ignorance or self-delusion.

Worse, it can't even agree on which nonsense to affirm in which context

But, in fact, the problem goes much deeper! According to this radical form of relativism, it isn't even objectively true that the Catholic Church was preaching geocentrism, or that the Inquisition put Galileo under house arrest for teaching heliocentrism, or for that matter that our own community subscribes to heliocentrism. If saying that some proposition *X* is true means only that some community has accepted *X* as true, then it would be equally valid for the following "truths" to "exist" by "being established" within a different community or social context:

> The Catholic Church was teaching heliocentrism and Galileo, who was never put under house arrest, was teaching geocentrism. Nowadays most people accept geocentrism. The Great Leap Forward did not promote exterminating sparrows or deep plowing.

That is, the question of what a given community establishes, or established, as truth, is itself a question of objective fact, so relativists are in some sense smuggling in a hidden dependency on

Alethic relativism contradicts itself when we look at what people believe

In an ontological sense, this is incoherent, in a way shown by Plato; the SEP summarizes his argument as follows:

> Most people believe that Protagoras's doctrine is false.
> Protagoras, on the other hand, believes his doctrine to be true.
> By his own doctrine, Protagoras must believe that his opponents' view is true.
> Therefore, Protagoras must believe that his own doctrine is false.

Plato's version of this argument is considerably lengthier and is quoted in full at the end of this note; as the SEP notes, though, this argument begs the question, implicitly relying on a notion of absolute rather than relative truth. Protagoras must believe that his opponents' view is true *for them*.

This argument is contingent on what the actual social consensus is; in a world where everyone agreed with Protagoras, it would lose its force. Nevertheless, we do not live in such a world. People routinely make factual assertions based on an evident belief that some statements are true and others are false in a fashion that is independent of people's opinions.

And it makes predictions very much at odds with reality

Borges, in *Tlön, Uqbar, Orbis Tertius* writes (my translation below):

> Siglos y siglos de idealismo no han dejado de influir en la realidad. No es infrecuente, en las regiones más antiguas de Tlön, la duplicación de objetos perdidos. Dos personas buscan un lápiz; la primera lo encuentra y no dice nada; la segunda encuentra un segundo lápiz no menos real, pero más ajustado a su expectativa. Esos objetos secundarios se llaman *hrönir* y son, aunque de forma

desairada, un poco más largos. Hasta hace poco los *hrönir* fueron hijos casuales de la distracción y el olvido. Parece mentira que su metódica producción cuente apenas cien años, pero así lo declara el Onceno Tomo. Los primeros intentos fueron estériles. El *modus operandi*, sin embargo, merece recordación. El director de una de las cárceles del estado comunicó a los presos que en el antiguo lecho de un río había ciertos sepulcros y prometió la libertad a quienes trajeran un hallazgo importante. Durante los meses que precedieron a la excavación les mostraron láminas fotográficas de lo que iban a hallar. Ese primer intento probó que la esperanza y la avidez pueden inhibir; una semana de trabajo con la pala y el pico no logró exhumar otro *hrön* que una rueda herrumbrada, de fecha posterior al experimento. Éste se mantuvo secreto y se repitió después en cuatro colegios. En tres fue casi total el fracaso; en el cuarto (cuyo director murió casualmente durante las primeras excavaciones) los discípulos exhumaron —o produjeron— una máscara de oro, una espada arcaica, dos o tres ánforas de barro y el verdinoso y mutilado torso de un rey con una inscripción en el pecho que no se ha logrado aún descifrar. Así se descubrió la improcedencia de testigos que conocieran la naturaleza experimental de la busca... Las investigaciones en masa producen objetos contradictorios; ahora se prefiere los trabajos individuales y casi improvisados. La metódica elaboración de *hrönir* (dice el Onceno Tomo) ha prestado servicios prodigiosos a los arqueólogos. Ha permitido interrogar y hasta modificar el pasado, que ahora no es menos plástico y menos dócil que el porvenir. Hecho curioso: los *hrönir* de segundo y de tercer grado —los *hrönir* derivados de otro *hrön*, los *hrönir* derivados del *hrön* de un *hrön*— exageran las aberraciones del inicial; los de quinto son casi uniformes; los de noveno se confunden con los de segundo; en los de undécimo hay una pureza de líneas que los originales no tienen. El proceso es periódico: el *hrön* de duodécimo grado ya empieza a decaer. Más extraño y más puro que todo *hrön* es a veces el *ur*, la cosa producida por sugestión, el objeto educido por la esperanza. La gran máscara de oro que he mencionado es un ilustre ejemplo.

Centuries and centuries of idealism have not ceased to influence reality. Not infrequently, in the oldest regions of Tlön, lost objects are duplicated. Two people seek a pencil; the first finds it and says nothing; the second finds a second pencil no less real, but more in keeping with their expectations. These secondary objects are called *hrönir* and are, though only slightly, somewhat longer. Until recently the *hrönir* were accidental children of distraction and forgetfulness. Incredibly, their methodical production is only a hundred years old, so the Eleventh Volume says. The first attempts were fruitless. The *modus operandi*, however, is worthy of note. The director of one of the state prisons told the prisoners that in the riverbed were certain sepulchres, promising liberty to whoever should bring him a significant find. During the months preceding the excavation, they were shown photographic slides of what they were to find. This first attempt showed that hope and greed can inhibit; a week of labor with shovels and picks failed to exhume any *hrön* but a rusty wheel, of a date later than the experiment. This was kept secret and repeated in four workshops. In three the failure was nearly total; in the fourth (whose director died coincidentally during the first excavations) the disciples exhumed — or produced — a golden mask, an archaic sword, two or three amphoras of mud, and the greenish and mutilated torso of a king with a still undeciphered inscription on his chest. Thus they discovered the uselessness of witnesses who understood the experimental nature of the search... Mass research produces contradictory objects; now preference is given to individual and almost improvisational experiments. The methodical production of *hrönir* (says the Eleventh Volume) has provided prodigious benefits to archaeologists. It has permitted the interrogation and even modification of the past, which is now no less flexible and docile than the future. Curious fact: the *hrönir* of second and third degree — the *hrönir* derived from another *hrön* — exaggerate the aberrations of the first; those of the fifth degree are almost uniform; those of the ninth degree are confused with those of the second; in those of the eleventh there is a purity of line that the originals lack. The process is periodic: the *hrön* of twelfth degree begins to decay. Stranger and purer than any *hrön* is, at times, the *ur*, the thing produced by suggestion, the object educed by hope. The great mask of gold I have mentioned is an illustrious example.

Along the same satirical lines, Douglas Adams writes, in *Life, the Universe, and Everything*:

“Recreational Impossibilities” was a heading which caught Trillian’s eye when, a

short while later, she sat down to flip through the Guide again, and as the Heart of Gold rushed at improbable speeds in an indeterminate direction, she sipped a cup of something undrinkable from the Nutrimatic Drink Dispenser and read about how to fly.

The Hitch Hiker's Guide to the Galaxy has this to say on the subject of flying.

There is an art, it says, or rather a knack to flying.

The knack lies in learning how to throw yourself at the ground and miss.

Pick a nice day, it suggests, and try it.

The first part is easy.

All it requires is simply the ability to throw yourself forward with all your weight, and the willingness not to mind that it's going to hurt.

That is, it's going to hurt if you fail to miss the ground.

Most people fail to miss the ground, and if they are really trying properly, the likelihood is that they will fail to miss it fairly hard.

Clearly, it's the second point, the missing, which presents the difficulties.

One problem is that you have to miss the ground accidentally. It's no good deliberately intending to miss the ground because you won't. You have to have your attention suddenly distracted by something else when you're halfway there, so that you are no longer thinking about falling, or about the ground, or about how much it's going to hurt if you fail to miss it.

It is notoriously difficult to prise your attention away from these three things during the split second you have at your disposal. Hence most people's failure, and their eventual disillusionment with this exhilarating and spectacular sport.

If, however, you are lucky enough to have your attention momentarily distracted at the crucial moment by, say, a gorgeous pair of legs (tentacles, pseudopodia, according to phylum and/or personal inclination) or a bomb going off in your vicinity, or by suddenly spotting an extremely rare species of beetle crawling along a nearby twig, then in your astonishment you will miss the ground completely and remain bobbing just a few inches above it in what might seem to be a slightly foolish manner.

This is a moment for superb and delicate concentration.

Bob and float, float and bob.

Ignore all considerations of your own weight and simply let yourself waft higher.

Do not listen to what anybody says to you at this point because they are unlikely to say anything helpful.

They are most likely to say something along the lines of, "Good God, you can't possibly be flying!"

It is vitally important not to believe them or they will suddenly be right.

Waft higher and higher.

Try a few swoops, gentle ones at first, then drift above the treetops breathing regularly.

Do not wave at anybody.

When you have done this a few times you will find the moment of distraction rapidly becomes easier and easier to achieve.

You will then learn all sorts of things about how to control your flight, your speed, your manoeuvrability, and the trick usually lies in not thinking too hard about whatever you want to do, but just allowing it to happen as if it was going to anyway.

You will also learn how to land properly, which is something you will almost certainly cock up, and cock up badly, on your first attempt.

There are private flying clubs you can join which help you achieve the all-important moment of distraction. They hire people with surprising bodies or opinions to leap out from behind bushes and exhibit and/or explain them at the crucial moments. Few genuine hitchhikers will be able to afford to join these clubs, but some may be able to get temporary employment at them.

Trillian read this longingly, but reluctantly decided that Zaphod wasn't really in the right frame of mind for attempting to fly, or for walking through mountains or for trying to get the Brantisvogan Civil Service to acknowledge a change-of-address card, which were the other things listed under the heading "Recreational Impossibilities".

Such Wile E. Coyote techniques do not work in our world, individually or in groups, so we can conclude that alethic relativism is false; that there exists a truth, an objective reality, independent of our communities, outside of a social context, even before common ground is found.

Of course, this too begs the question — it relies on the presumption that it is *objectively true* that such events do not happen in our world, while a relativist might argue that *for us* they do not happen, while *for them* such events *do* happen. I have not yet met a relativist who has dared to make such an argument, but in this way alethic relativism could insulate itself from all possible disproof, just as ordinary solipsism does.

As the Wikipedia article on social constructionism complains, "It has been objected that strong social constructionism undermines the foundation of science as the pursuit of objectivity and, as a theory, defies any attempt at falsifying it."

Alethic relativism is malignant because it would render science, engineering, and the rest of philosophy impossible

So what? Why is it worth talking about? Many people believe many false and incoherent things, but we don't spend all our time constructing detailed refutations of each of them. What makes this doctrine so "malignant" it's worth our attention?

I have two major concerns with this sort of willful blindness to objective reality, which accepts popularity contests as superior to empirical evidence or logical reasoning: it is guaranteed to render disagreements unresolvable, and it makes rational action impossible. It's a collective variant of solipsism in the sense that, rather than proposing that *I* am alone in the universe, it proposes that *we* are alone in the universe; every other fact is taken to be merely a social consensus, only "true" relative to the community that established it, possibly "false" relative to other communities. By deluding its adherents into rejecting the universe, it tempts them to "establish truths" by collective partisan violence rather than by logic and evidence, violence that ultimately ends in self-destruction.

As Borges writes:

> Este monismo o idealismo total invalida la ciencia. Explicar (o juzgar) un hecho es unirlo a otro; esa vinculación, en Tlön, es un estado posterior del sujeto, que no puede afectar o iluminar el estado anterior. Todo estado mental es irreductible: el mero hecho de nombrarlo –*id est*, de clasificarlo– importa un falseo. De ello cabría deducir que no hay ciencias en Tlön –ni siquiera razonamientos. La paradójica verdad es que existen, en casi innumerable número. ... El hecho de que toda filosofía sea de antemano un juego dialéctico, una *Philosophie des Als Ob*, ha contribuido a multiplicarlas. Abundan los sistemas increíbles, pero de arquitectura agradable o de tipo sensacional. Los metafísicos de Tlön no buscan la verdad ni siquiera la verosimilitud: buscan el asombro. Juzgan que la metafísica es una rama de la literatura fantástica.

My translation:

> This monism or total idealism invalidates science. Explaining (or judging) a fact is uniting it to another; this linking, in Tlön, is a posterior state of the subject, which cannot affect or illuminate its prior state. Every mental state is irreducible: the

mere fact of naming it — that is, of classifying it — imports a falsehood. From this one could deduce that there are no sciences in Tlön — not even reasoning. The paradoxical truth is that they do exist, in an almost uncountable number. … The fact that all philosophy is from the beginning a dialectical game, a *Philosophie des Als Ob*, has contributed to multiplying them. Incredible systems abound, but of agreeable architecture or of a sensational type. The metaphysicians of Tlön do not seek the truth or even plausibility: they seek surprise. They consider metaphysics to be a branch of fantasy literature.

Of course, people commonly disagree on empirical questions like whether Elvis Presley is still alive or not, as well as logical or mathematical questions like whether it is true that every even whole number greater than 2 is the sum of two prime numbers, for example due to access to differing evidence or due to limitations in their reasoning.

And people commonly differ in their interpretation of language; one person may interpret a sentence as a proposition that is true, while another interprets it as a diiferent proposition that is false. For example, the sentence "every even whole number is the sum of two prime numbers" depends on the definition of "prime number"; it is trivially false according to our modern definition (in which 1 is not a prime number), because 2 is a example, but when Euler originally stated Goldbach's conjecture, he stated it that way because he was using a definition of "prime numbers" (or rather, "prīmī", using a Latin word) that included 1 as a "prime". Thus our interpretation of a sentence, for example in English, German, or Latin, depends on the socially constructed meanings of the words in it. ("Nice" used to mean "insignificant", too.)

But these sorts of disagreement presuppose the existence of an objective reality that could, in principle, resolve them. Alethic relativism, by contrast, claims that it is merely a matter of social consensus — opinion — whether Elvis is alive or dead, or whether 2 + 2 = 4. It claims that there is no objective sense in which Elvis is dead or alive, or in which 2 + 2 = 4, so there is no point in talking about it except to establish social consensus.

Alethic relativism as gaslighting

https://en.wikipedia.org/wiki/Chain_of_Command_%28Star_Trek:◦
◦_The_Next_Generation%29

Plato's version of the self-refutation argument

This version is somewhat more long-winded than the SEP version I quoted above:

Socrates Let us then get the agreement in as concise a form as possible, not through others, but from his [Protagoras's] own statement.

Theodorus How?

Socrates In this way: He says, does he not? "that which appears to each person really is to him to whom it appears."

Theodorus Yes, that is what he says.

Socrates Well then, Protagoras, we also utter the opinions of a man, or rather, of all men, and we say that there is no one who does not think himself wiser than others in some respects and others wiser than himself in other respects; for

instance, in times of greatest danger, when people are distressed in war or by diseases or at sea, they regard their commanders as gods and expect them to be their saviors, though they excel them in nothing except knowledge. And all the world of men is, I dare say, full of people seeking teachers and rulers for themselves and the animals and for human activities, and, on the other hand, of people who consider themselves qualified to teach and qualified to rule. And in all these instances we must say that men themselves believe that wisdom and ignorance exist in the world of men, must we not?

Theodorus Yes, we must.

Socrates And therefore they think that wisdom is true thinking and ignorance false opinion, do they not?

Theodorus Of course.

Socrates Well then, Protagoras, what shall we do about the doctrine? Shall we say that the opinions which men have are always true, or sometimes true and sometimes false? For the result of either statement is that their opinions are not always true, but may be either true or false. Just think, Theodorus, would any follower of Protagoras, or you yourself care to contend that no person thinks that another is ignorant and has false opinions?

Theodorus No, that is incredible, Socrates.

Socrates And yet this is the predicament to which the doctrine that man is the measure of all things inevitably leads.

Theodorus How so?

Socrates When you have come to a decision in your own mind about something, and declare your opinion to me, this opinion is, according to his doctrine, true to you; let us grant that; but may not the rest of us sit in judgement on your decision, or do we always judge that your opinion is true? Do not myriads of men on each occasion oppose their opinions to yours, believing that your judgement and belief are false?

Theodorus Yes, by Zeus, Socrates, countless myriads in truth, as Homer says, and they give me all the trouble in the world.

Socrates Well then, shall we say that in such a case your opinion is true to you but false to the myriads?

Theodorus That seems to be the inevitable deduction.

Socrates And what of Protagoras himself? If neither he himself thought, nor people in general think, as indeed they do not, that man is the measure of all things, is it not inevitable that the "truth" which he wrote is true to no one? But if he himself thought it was true, and people in general do not agree with him, in the first place you know that it is just so much more false than true as the number of those who do not believe it is greater than the number of those who do.

Theodorus Necessarily, if it is to be true or false according to each individual opinion.

Socrates Secondly, it involves this, which is a very pretty result; he concedes about his own opinion the truth of the opinion of those who disagree with him and think that his opinion is false, since he grants that the opinions of all men are true.

Theodorus Certainly.

Socrates Then would he not be conceding that his own opinion is false, if he grants that the opinion of those who think he is in error is true?

Theodorus Necessarily.

Socrates But the others do not concede that they are in error, do they?

Theodorus No, they do not.

Socrates And he, in turn, according to his writings, grants that this opinion also is true.

Theodorus Evidently.

Socrates Then all men, beginning with Protagoras, will dispute—or rather, he will grant, after he once concedes that the opinion of the man who holds the opposite view is true—even Protagoras himself, I say, will concede that neither a dog nor any casual man is a measure of anything whatsoever that he has not learned. Is not that the case?

Theodorus Yes.

Socrates Then since the "truth" of Protagoras is disputed by all, it would be true to nobody, neither to anyone else nor to him.

Protagoras's argument doesn't become any more coherent when applied to communities rather than individuals.

Topics

- Flying (p. 1296) (3 notes)
- Ontology (p. 1350) (2 notes)
- Galileo (p. 1367) (2 notes)

Solid rock on a gossamer skeleton through exponential deposition

Kragen Javier Sitaker, 02021-12-15 (updated 02021-12-30) (11 minutes)

Suppose you have made a lightweight model of something out of aluminum foil or thin aluminum mesh, for example by origami or stamping or bending, and you'd like to solidify that geometry. But the aluminum foil is very lightweight and flimsy (see Aluminum foil (p. 413), typically 10 μm) and so it can't withstand much force at all without deforming dramatically.

Extending this to thinner metal sheets such as gold leaf would be desirable but seems much more challenging.

One approach to this is cathodic deposition, whether of metals or of other minerals; see Fast electrolytic mineral accretion (seacrete) for digital fabrication? (p. 779). That file covers many candidate approaches to cathodic deposition of nonmetals. But there are some other approaches I want to explore.

Waterglass spray systems

Another thing you can do is to coat the foil with waterglass, for example by spraying. I think this is sort of the opposite of spray drying: as in spray drying, you want very fine droplets, under 10 μm, so that they don't collapse the aluminum with their weight, but you probably need the air to be sufficiently humid that the fine droplets do not dry before they can stick to the aluminum, where they are smoothed out by the surface tension of the solution and the hydrophilicity of the surface. (You may need to functionalize the surface to be more hydrophilic first.) With sufficiently outrageous pressure, you should be able to atomize even a fairly concentrated waterglass solution (≈30%) through a small orifice.

By this means you ought to be able to deposit an additional 10 μm or so of waterglass on the surface of the aluminum to stiffen it; upon exposing the object to dry, hot air, such a thin film should be able to dry fairly quickly. The resulting silica gel might be 5 μm thick on both sides of the foil, doubling its thickness. Silica xerogel is not very stiff, with a Young's modulus in the neighborhood of $E = 3$ GPa (compare to aluminum's $E = 70–90$ GPa).

If the surface is sprayed in the same way, either before or after the silicate, with a solution of a salt containing polyvalent cations that will not displace aluminum in the metal, with anions that will not decompose to oxidize the aluminum (such as the acetate or formate of magnesium, calcium, manganese(II), zinc, copper, or iron, especially ferrous but even ferric; or the acetate, sulfate, nitrate, iodide, or chloride of aluminum or magnesium; or the nitrate or iodide of calcium) it should immediately render the silicate insoluble upon contact, in the same way as in Keim's process for mineral painting (Keimfarben), but much more rapidly because of the thinness of the layer and because of carrying out the whole process at an elevated

temperature.

In addition, some of the resulting silicate compounds might be stiffer than a simple silica xerogel; *some* silicates of aluminum and magnesium are notable for their outstanding quartz-like hardness. I think aluminum and magnesium are also more advantageous in this respect because there is no danger that they will displace the aluminum metal, so they afford a wider choice of salts than zinc, copper, or iron, and because they contain more valence electrons per mass; I fear that the acetate, sulfate, nitrate, iodide, or chloride of zinc, copper, or iron might corrode the aluminum, though I think they normally are not sufficiently aggressive to corrode it in the time available.

(In general, lower alkalinity waterglasses will not only be able to solidify with smaller additions of polyvalent cations, but will also produce stiffer materials, because the silicate network tends to provide most of the mineral's strength.)

Alternative ways to rapidly solidify waterglass include carbonic acid gas, and alcohols such as methanol or ethanol, but these last are reputed to produce a rubbery effect which would be counterproductive in this context.

Aqueous phosphate spray systems

As an alternative to waterglass in this process, sources of soluble phosphate can be used, such as phosphoric acid and the soluble monobasic, dibasic, or tribasic phosphates of sodium, potassium, or azanium. These can be reacted with polyvalent cations in the same way as the soluble silicates to form insoluble mineral phosphates, some of which are competitive in hardness with the silicates. In many cases the reactions are not as calm as the corresponding silicate reactions.

Sufficient quantities of phosphoric acid can convert aluminum foil into the water-soluble acidic monoaluminum tri(dihydrogen)phosphate, though normally this reaction takes hours, while the more usual 1:1 aluminum phosphate is aggressively insoluble. Another possible disadvantage of phosphoric acid is that it would be much harder to dry out. I don't think the other phosphates are aggressive enough to attack aluminum foil.

The azanium phosphates are particularly interesting here because the azanium can be driven off by heating, leaving the anhydrous acid if this is done before adding the other reagent; the monobasic phosphate decomposes around 200°. In the case where the salt contributing the polyvalent cations is a muriate, fluoride, iodide, or formate, the heating step can remove the azanium-salt byproduct entirely after combining the two solutions, leaving only the desired mineral. Of these, the azanium muriate gas is probably the least objectionable.

Pyrophosphates or metaphosphates are likely alternatives to orthophosphates here; as with lower-alkalinity soluble silicates, these longer-chain phosphates may require smaller amounts of polyvalent cations and produce stiffer materials. If this effect exists, it would be much weaker than with the silicate systems.

General remarks on aqueous spray systems

Getting sprayed drops of the right size onto the aluminum is probably best done by producing the spray in a chamber with a slight updraft which will carry the smaller drops to the workpiece, while allowing larger drops to fall and be recycled. In the cases other than waterglass/carbonic acid, it would be best to use one such chamber for each liquid so that they do not react in the spray chamber and can be recycled safely.

Micron-sized filler particles, such as clay, talc, mica, silica (as in sol-silicate paint), or nanotubes (whether of carbon, BCN, boron nitride, halloysite, or some other material) could further enhance the stiffness of the resulting material and reduce the quantity of polyvalent cations required. These could be mixed into either of the two solutions.

As the object gets thicker layer by layer, it will become stiffer in proportion to the square of its thickness, so after a while it will be possible to deposit thicker layers.

Foamable systems

A possible alternative approach is to form your original shape out of not one but *two* layers of aluminum foil which are stuck together with drops of dried waterglass before being formed. Upon heating the formed shape, the waterglass softens, and the substantial amount of water locked inside its gel structure bubbles out, forming a foam, which pushes the two sheets of aluminum some distance apart. If the waterglass layer is continuous before foaming, this will badly distort the shape and quite likely rip the aluminum foil, but if adequate space is present laterally between the drops, they will have space to expand without damaging the foil or distorting the shape much, while still forming a continuous foam network and doing most of their expansion perpendicular to the surface. Once cooled, the resulting sandwich panel is potentially substantially more rigid than the original easily-formable material.

According to a preliminary test on a much larger scale (see Material observations (p. 633), section 02021-08-20) waterglass foamed by heating commonly expands in volume by about a factor of 10; so a layer of 10-μm-thick waterglass drops that is half waterglass drops and half air might expand 5× in thickness to 50 μm, making the total sandwich panel 70 μm and, I think, 9× its original rigidity ($(60/20)^2$). If you can manage full density, no spaces between drops, without ripping the foil, you can get to 100 μm and 30× the original rigidity ($(110/20)^2$).

By placing the waterglass drops along a pattern of crisscrossing lines, rather than uniformly distributed over the whole plane, it may be possible to use less total material at the expense of less increase in thickness and thus in rigidity.

If instead of a sandwich between two layers of foil we deposit the drops of waterglass on a wire mesh, they are more likely to chip off, but they will tend to distort the form less when heated, forming a solid foam piece.

If instead of waterglass we use drops of dried phosphates of azanium, heating will drive off azanium instead of water, melting the resulting phosphoric acid and allowing it to foam up with the

azanium gas. A slow-acting source of polyvalent cations, inert to phosphates of azanium at room temperature but reactive with warm anhydrous phosphoric acid, can be mixed in with the phosphates. Oxides of zinc, copper, aluminum, iron, or magnesium would probably work well for this with the grain size and grain surface structure adjusted to get the right level of reactivity.

In either case, including a small amount of a polyvalent cation in the waterglass or phosphate solution before drying, but not enough to precipitate on its own, might enable it to gel at a higher water content, thus providing a greater foaming structure.

Borax is another material that foams up at temperatures below the melting point of aluminum, because like waterglass it softens up and water is driven out, but it's not as easy to precipitate a water-insoluble material from. It might be possible to convert it into hydroboracite ($CaMgB_6O_{11}·6H_2O$) by reacting it with both calcium and magnesium ions, but this is far from the enthusiasm with which the phosphate and silicate systems form insoluble products, and even hydroboracite is not very hard.

Mixing pyrophosphates, orthophosphates, and metaphosphates together may be useful to encourage phosphates to form an amorphous gel (that can trap a lot of water) rather than crystals (which in a few cases can be quite hydrated, for example the decahydrates of di- and trisodium phosphate.)

Topics

- Materials (p. 1138) (59 notes)
- Digital fabrication (p. 1149) (31 notes)
- Phosphates (p. 1184) (9 notes)
- Foam (p. 1185) (9 notes)
- Waterglass (p. 1189) (8 notes)
- Aluminum foil (p. 1237) (5 notes)

3-D printing in poly(vinyl alcohol)

Kragen Javier Sitaker, 02021-12-15 (updated 02021-12-30)
(2 minutes)

Because polyvinyl alcohol can be (ionically) crosslinked with borate to gel, forming a much less water-soluble polymer (though still a thermoplastic), I think you could do a useful 3-D printing system by extruding a solution of polyvinyl alcohol in water, either spraying the surface with a solution of borax or boric acid in between layers to solidify it, or submerging it into a tank of the same.

The polymer is already widely available, sold as a mold release, for about US$50/kg, and is biodegradable and non-toxic. (Borate is also pretty nontoxic, but it's not in the same league as PVA.) It's quite strong and stiff, comparable to PLA.

This system would have the following potential advantages over the now-conventional PLA FDM process developed by the RepRap project:

- No heating is needed for the extruder.
- The polymer is crystal-clear, enabling the production of transparent objects; films of it are commonly used in optical devices.
- The hydrogel resulting from adding the borate is elastomeric and sufficiently hygroscopic to resist dehydration under ordinary conditions.
- Various functional fillers can easily be added. I have some PVA glue here that's full of glitter.
- When dry, it has extremely low oxygen permeability for an organic material.
- Because the polymer is mixed with a solvent, it can form a lower volume fraction of the final product; I think loadings under 10% are common in school glue. This should permit higher *filler* loadings in the final product than in filled PLA systems, which may permit better filler-induced properties, such as mechanical strength, stiffness, magnetism, conductivity, porosity, glitteriness, etc.

Alternatively, you could use it for a liquid-vat-hardening process similar to SLA, in which the liquid is selectively hardened not by selective exposure to light but by selective deposition of borate on the surface of a vat of liquid, using one or many nozzles, alternating with raising the level of liquid in the vat slightly. With the right crosslinking initiators, you could probably induce PVA to crosslink with light, too, which would probably form a covalently linked thermoset rather than the ionically crosslinked thermoplastic formed by the PVA/borate system.

There exist alternative crosslinkers for PVA that produce higher strength without UV, such as glutaraldehyde, which creates a covalently crosslinked material.

Topics

- Materials (p. 1138) (59 notes)
- Digital fabrication (p. 1149) (31 notes)
- 3-D printing (p. 1160) (17 notes)
- Poly(vinyl alcohol) (PVA) (p. 1245) (4 notes)
- Glutaraldehyde (p. 1294) (3 notes)
- Cross linking (p. 1377) (2 notes)

Ghetto electrochromic displays for ultra-low-power computing?

Kragen Javier Sitaker, 02021-12-16 (updated 02021-12-30)
(9 minutes)

Tungsten oxide, Prussian blue, and some other metal compounds can go through a reversible electrolytic redox reaction that changes their color or transparency; commonly this involves intercalating lithium ions into them. But of course you can also electrolytically oxidize silver to black silver oxide or, if sulfur ions are available, to sulfide, and then reduce it again by reversing the current; this sort of thing is also done by artisans to add contrast to copper objects, typically using liver of sulfur rather than electrolysis.

Display design

This suggests that by running, say, silver or copper "electrochromic" strips in one direction and thin wire counter electrodes over them in the other direction, filling the space between with a thin electrolyte (maybe a hydrogel), you could get a very simple electrochromic display. It might not be fast or last through many switching cycles but it should still be interesting. Like other electrochromic displays it would be fairly bistable and thus potentially very energy efficient for passive reading.

If you apply higher voltages to speed up the reactions, unless you use a per-pixel diode or transistor, you might get some bleedover into other pixels in the same row and column, as well as the rest of the display. If you're applying +2.1 V to a pixel, then any pixel not in the same row or column is in series with a pixel in the same row and a pixel in the same column with +0.7 V, -0.7 V, +0.7 V respectively. By a similar route, unless there are per-pixel diodes, different pixels will tend to drive currents through one another even when the driver is open-circuit, which will tend to equalize the charge and therefore the colors along each row and column.

With either per-pixel diodes or per-pixel transistors, the idea is that one of the two electrodes (let's say the counter electrode, though the electrochromic electrode would work too) is divided into one section per pixel. In the diode case, there are two insulated wires for that row or column, one with a diode from it to the electrode, which can thus make the electrode an anode, and the other with a diode from the electrode to it, which can thus make the electrode a cathode. Ideally these would be germanium diodes, Schottky diodes, or both, to reduce the voltage error.

In the transistor case, the channel of a FET switchably connects the electrode to a power-supply line, which itself can be brought low or high, so you still have two insulated wires but you no longer have a voltage error. We're using low enough voltages that the FET body diode probably doesn't matter; if it does, you might be able to use a silicon carbide MOSFET (which has a larger body diode forward voltage because of carborundum's 3.3 V bandgap, triple silicon's; the

MSC040SMA120B4 is rated for -4.0 V but the plot shows appreciable body-diode current at only -1.5 V, depending on Vgs) or I think you can get MOSFETs where the body terminal is brought out as a fourth pin, in which case you could tie that to a third power supply wire. (However, the 4-pin discrete MOSFETs I've been able to find use the fourth pin as a Kelvin-connection probe for sensing the voltage at the source on chip.)

The electrolytic reactions at the wire counter electrodes must also be taken into account; if they produce gas, for example, it will deplete the electrolyte, mechanically stress the device with gas bubbles, and may create an explosion risk. If the "wires" are, for example, transparent ITO strips, anything that forms on their surface will also be in the optical path; alternatively they could be the same metal as the electrochromic electrode, though they will probably have different overpotentials due to smaller surface area and thus higher current density.

You need the electrolyte to be on the same order of thickness as the pixel width in order to change the color of the whole pixel, though if the reaction passivates or "polarizes" the electrochromic electrode it might just be a question of how soon the color changes in each part of the pixel. That effect could be used to get, in effect, multiple pixels per intersection: whatever part of the electrochromic electrode is closest to the counter electrode would react first.

It may be useful to have reference electrodes that run along either rows or columns in order to control the voltage on the electrochromic electrode more precisely.

Such a device could presumably be used as short-term nonvolatile memory as well, using the thickness of the passivation layer thus formed to record a bit, measured by the ratio of resistive impedance to capacitive impedance by probing it at two frequencies.

Some materials have different extinction coefficients (opacities) for different wavelengths, so the color of their films depends on their thickness, quite aside from iridescence. For oxide layers that are not very opaque at any wavelength, the iridescence effect will tend to be stronger than the inherent color of the oxide formed, though it will be weaker in contact with water than with air, since the index of water is 1.33, close to common glasses. However, zinc oxide is 2.4, hematite is close to 2.9, tenorite is 2.9–3.1, titania is 2.6, and the strength of the reflection at the interface is roughly proportional to the square of the difference of the indices, so such materials would still have great potential for iridescence.

In general these devices will act faster at higher temperatures.

Copper oxides

The Pourbaix diagram for copper shows that above about pH 7 and above about +0.3 volts the equilibrium favors black cupric tenorite, CuO; as pH increases to about 12.5 the critical voltage decreases to about -0.2 volts. But there's a small region, for example from about -0.1 V to about +0.2 V at pH 8, where instead red cuprite, Cu_2O, is favored. (Different sources disagree on exactly how big this window is.) At more negative voltages, the equilibrium favors the reduction back to copper metal.

In this case the electrolyte would need to be slightly alkaline, and maybe you could get three colors: copper yellow, red, and black. Possibly turning a pixel red might take weeks.

There also exists an unstable olive-green copper peroxide, but I don't think you can make it in this way; you need pre-existing peroxide groups.

If the copper forms dissolved copper salts, they will of course be green, and when it redeposits as metallic copper it will often be yellow rather than shiny. Oxides of copper are very insoluble, though, so this presumes some other materials in the electrolyte.

Copper oxide itself is an electrochromic material and when it contains some cuprite it is reported to be somewhat reddish-gray even when only 60–500 nm thick.

Iron oxides and hydroxides

Iron oxides can have many different colors, especially with water hydroxylating them: in pottery commonly red, green, grey, or brown ; there are sixteen known oxides, including black Fe_3O_4 magnetite, black FeO wüstite, red Fe_2O_3 hematite, orange/brown FeOOH goethite which can be yellow to black depending on things including limonite at the yellow end, and green. This is another possible multicolored pixel, although you probably can't get *all* of those colors; the Pourbaix diagram for iron in water at 25° says that starting about pH 8.1, you get iron up to about -0.5 V, (green?) fougèrite up to about -0.3 V, black magnetite up to about 0 V, red hematite up to about 1.2 V, and then aqueous ferrate solution, "pale violet… one of the strongest water-stable oxidizing species known" (!). However, I suspect that most of these reactions are very slow.

Nickel oxides

Nickel is pretty passive most of the time, but nickel oxide is used in pottery to produce blue, grey, yellow, and black, and its usual NiO form is green, while the trivalent oxide-hydroxide is black. I'm not sure if you can form the divalent green compound in water; the Pourbaix diagrams I'm finding are contradictory.

Topics

- Materials (p. 1138) (59 notes)
- Electrolysis (p. 1158) (18 notes)
- Ghettobotics (p. 1169) (12 notes)
- Bootstrapping (p. 1171) (12 notes)
- Optics (p. 1209) (6 notes)
- Copper (p. 1234) (5 notes)
- Displays (p. 1261) (4 notes)
- Spatial light modulators (SLMs) (p. 1327) (2 notes)
- Silver (p. 1328) (2 notes)

Electrolytic 2-D cutting and related electrolytic digital fabrication processes

Kragen Javier Sitaker, 02021-12-16 (updated 02021-12-30)
(48 minutes)

As I've written about at some length previously, some of the most promising computational fabrication technologies at macroscopic scale are 2-D cutting processes like laser cutting, waterjet cutting, and CNC plasma table cutting. See Layers plus electroforming (p. 1100) for notes on the scaling laws. Cooper Zurad has prototyped an electrolytic 2-D cutting process using a needle-shaped cathode, but his process is very slow and imprecise because he's cutting at a single point, and because he's not doing the usual ECM things: closed-loop control of the process gap, using pulsed current, or vibrating the electrodes to reduce the tradeoff between flushing and cutting speed. Traumflug did similar experiments in 02011.

However, even without gap control, pulsing, and vibration, if you're cutting over a large area rather than just a point, you should be able to get a proportionally higher material removal rate. And if you're cutting a thin sheet of material, even a low material removal rate might be adequate.

Unlike other ways of cutting a thin sheet of material, this sort of electrolytic cutting leaves no burrs and does not produce heat-affected zones or mechanical stresses in the material being cut — though for very thin sheets the surface tension of the electrolyte may be big enough to plastically deform the workpiece, possibly requiring supercritical drying if the workpiece is ever to be removed from water.

In addition to cutting sheet metal, this process can be used to selectively remove a metal coating (as on a printed circuit board) or to etch or anodize a surface.

Cathode patterns

One way to do this is, as I wrote in Dercuano in 02016, to have an array of separate cathodes close to the anode workpiece, controlling the voltage or current of each cathode to either dissolve the workpiece near it, or not. A different way is to prepare the pattern of the cuts in a material form, for example as a printed circuit board, a network of wires on an insulating plate, or a pattern of apertures in an dielectric mask placed over a continuous-sheet cathode, and make this pattern the cathode. Then you can "print" this cut pattern on a series of sheets of metal.

One particularly interesting cathode-patterning possibility is to produce the insulating "mask" by laser-printing on paper, ideally paper that will not fall apart when soaked with the electrolyte. If the laser printing is sufficiently solid to be used for other toner-transfer methods, it should also work for this electrolytic sheet cutting approach; unlike the other toner-transfer methods, it might be

possible to get more than one metal copy from a single paper pattern.

Thermal wax printers may or may not produce a better dielectric pattern.

Separators between the cathode and anode

In either case, the cathode is placed very close to the anode workpiece; the most practical way of doing this is probably to separate them with either some sort of fabric, such as a paper towel or other nonwoven cloth or a woven cloth, or with a thin porous membrane full of holes, for example a thin porous layer of polyethylene. (In the case of laser printing on paper, the paper itself provides the separator.) Once the current is turned on, the cuts are all made simultaneously, though some may take longer to finish than others. Voltammetry should be adequate to determine when the process completes.

With this approach, the porous separator, plus any space it produces around the cathode, need to be able to absorb the metal salts produced by the cutting, since vibration or indeed flushing at all would be very difficult.

Containment: inner and outer cut contours

In cases where the cut pattern contains some cuts that are completely surrounded by others, for example holes "drilled" in a part to be cut out, there is a potential problem. If we feed the anode current in from the edge of the anode workpiece, it may not be able to reach the inner cuts if the outer cuts happen to complete first. There are several possible ways to solve this problem:

- We can do the cutting in two phases, first cutting out the inner contours and then later the outer contours. This is a common tactic in single-point 2-D cutting processes like those I mentioned above, because it prevents the outer piece from falling out of plane before you've cut the holes in it. This would result in slower cutting but prevents the problem. It requires setting up the cathode pattern in two or more electrically separate parts.
- We can feed in the current to the anode with an inert conductive backing anode, thus coupling in current to the workpiece over its entire back surface rather than only at the edge. This will reduce the efficiency of the process and perhaps eliminate the usefulness of voltammetry for measuring its completion.
- We can try to leave "tabs" around the outside of each cut, so that when the process is finished all the desired parts are still all connected together and must be removed from their support in a separate step. This is commonly done in casting, molding, and, especially in the case of thin materials, 2-D cutting by means such as lasers. It might be possible to calibrate the process with enough precision that the tabs are barely thick enough to maintain electrical continuity until the inner contours are all cut out, but continuing the process a little longer severs the tabs.

Example setups

- As one example, as I wrote in “Dead bugging” in Derctuo, it’s easy enough to find 100-μm-thick copper conductors (a) in stranded copper wire. You could paste these in the desired pattern (b) onto a high-impact polystyrene surface (c) with PVA glue (d), bridging disconnected parts of the cut pattern with fine varnish-insulated magnet wire (e). After allowing the PVA glue to dry, you make it water-insoluble by treating with a borax solution. You lay a paper towel (f) onto the pattern and soak it with sodium nitrate (g). You lay a sheet of 10-μm-thick household aluminum foil (h) onto the paper towel and press the whole assembly together, then apply a low voltage power supply for long enough to dissolve more than 10 μm of aluminum. The results should be, for example, if your current density is enough to cut 100 μm/s, that the cut is completed in 100 ms. Applying electricity for less time will result in etching the surface rather than cutting through.
- You can heat up example 1 to make the cutting go faster, using a heated press such as are commonly used in dye sublimation, laundry pressing, and vinyl transfer onto cloth.
- For the foil (h) in examples 1 or 2 you can substitute heavy-duty aluminum foil, commonly available in thicknesses such as 50 μm; aluminum flashing; aluminum sheet from aluminum cans (typically 100 μm) after cleaning nonconductive contaminants off of one of its two sides; aluminum sheet as is commonly available from metal vendors like Metals Depot, commonly in thicknesses as low as .032" (81 μm) or any other source of sheet aluminum. Thicker sheets will take proportionally longer to cut, produce less precise cuts, and, above a certain thickness, will also require a thicker separator (f). Cutting multiple stacked layers of metal (h) in a single run is a possibility that may increase the efficiency of the process in several ways, such as amortizing the setup time over multiple produced pieces, but will reduce the precision achieved.
- For the sodium nitrate (g) in examples 1, 2, or 3 you can substitute any other soluble salt whose anion forms a soluble aluminum salt or aluminate, such as sodium chloride, azanium acetate, iron sulfate, or potassium hydroxide, among dozens of other possibilities; particularly appealing are the chloride, acetate, sulfate, and hydroxide salts of azanium and the alkali metals, due to their high solubility and low toxicity; the corresponding salts of iron and zinc are also relatively safe and, except for the hydroxides, soluble. More toxic options include sodium perchlorate. In cases such as potassium hydroxide which are capable of corroding the aluminum rapidly without electricity, it will be necessary to stop the reaction, for example by washing the pieces thus produced with water, a buffer solution, or an acid that will not attack the aluminum. Salts which produce a passivating “anodized” layer on the aluminum at lower voltages may be preferable, because although they reduce efficiency, they will restrict the electrolytic etching to areas at sufficiently high voltages, improving the precision of the process. It is probably also useful to include additives such as metal borates, metal EDTA, metal tetrasodiumglutamatediacetates (GLDA) to prevent the formation of aluminum hydroxide, metal cyanides, SPS (CAS 27206-35-5), MPS (CAS 17636-10-1), ZPS (CAS 49625-94-7), polyethylene glycol,

polyvinyl alcohol, polyvinyl acetate, glycerine, propylene glycol, dipropylene glycol, DPS (CAS 18880-36-9), surfactants (such as SLS, alkali stearates, EN 16-80 (CAS 26468-86-0), or EA 15-90(CAS 154906-10-2)), UPS (CAS 21668-81-5), PPS (CAS 15471-17-7), NAPE 14-90 (CAS 120478-49-1), sodium benzoate, saccharin, coumarin, metal tartrates, metal citrates, metal sulfonates not otherwise mentioned, metal urates, thiazole, benzaldehyde, thiourea, quaternary azanium salts, phthalimide, metal methanesulfonates, metal ethylene sulfonates, depolarizers (such as manganese dioxide, metal sulfates, silver oxide, or metal chromates and dichromates, among many other possibilities), the acid forms of the anions mentioned here, or these anions' salts with organic cations or azanium, as well as other additives used in electrodeposition and ECM. Also, the solvent in which the salt is dissolved can be replaced with any other solvent suitable for the salts employed, such as DMSO, ammonia, ethyl acetate, THF, DCM, acetone, acetonitrile, DMF, formamide, acetic or formic acid, alcohols (such as methanol, ethanol, and isopropanol, among many others), organic carbonates (such as propylene carbonate, ethylene carbonate, diethyl carbonate, or dimethyl carbonate, among many others), glycerol, nitromethane, molten methylsulfonylmethane, deep eutectic systems, or other ionic solvents, among hundreds of others, or mixtures of these, with or without water; such substitution can permit the use of higher temperatures or electrolyte salts that either react undesirably in water or will not dissolve in it, and may be able to reduce the surface tension to less mechanically damaging levels. Generally the more important solubility consideration will be the solubility of the salts produced at the anode workpiece, since you cannot choose their cations, rather than the electrolytic etchant (g).

- For the paper towel (f) in examples 1, 2, 3, or 4 you can substitute any other porous material that will not be attacked too rapidly by the salts and is not too electrically conductive except ionically; for example, asbestos, fiberglass, carbon fibers, carborundum fibers, rock wool, basalt fiber, ordinary paper, buckypaper, nonwoven polypropylene, nonwoven polyester, nonwoven cotton, nonwoven rayon, onion-skin paper, other thin papers such as crepe paper and those used for tracing drawings and rolling cigarettes, perforated polyethylene film, perforated PET film, perforated polypropylene film, hydrogels (such as gelatin, agar, borated polyvinyl alcohol, or silica gel), woven textiles of the above-mentioned fibers, and porous ceramics such as glass frits or unglazed fired clays, among dozens or hundreds of other possibilities. Woven textiles will tend to add their weave pattern to the etched pattern, which may be considered a form of error in some applications. You can stack more than one such layer; for example, a layer of perforated polyethylene film can be used to separate a layer of borated PVA hydrogel from the cathode, preventing adhesion. Perforated boPET or polyethylene films can easily be made under 10 μm in thickness, a feature which might enable reproducing details not much larger than that.
- For the varnish-insulated magnet wire (e) in examples 1, 2, 3, 4, and 5, you can substitute wire insulated by other means such as thin layers of dielectric polymers, or you can pierce holes in the dielectric backing (c) to pass through conductors from a region devoid of electrolyte or at least separated from the workpiece by a dielectric or

by distance.

- For the borate-crosslinked PVA glue (d) in examples 1, 2, 3, 4, 5, and 6, you can substitute any other material that will hold the pattern conductors in place while permitting electrolytic access to them; for example, agar, gelatin, cross-linked starch, hydrogels used for contact lenses (such as silicone hydrogels, hydroxyethyl methacrylate), sodium polyacrylate as used in maxi pads, polyethylene glycol (perhaps treated to crosslink it into an insoluble gel as is commonly done for cell encapsulation), and porous ceramics such as glass frits or those made by unglazed fired clay. Alternatively, the separator material (f) can simply be bonded permanently to the cathode, which would require the electrolyte to be washed out between runs rather than merely replacing the separator as you would normally do. Alternatively, instead of holding the cathode pattern in place with any kind of continuous material, you can hold it in place with occasional thin fibers of dielectric material either bonded to the dielectric separator (c) or passing through it, as is done in embroidery or furniture decoration to hold certain kinds of thread or piping on the surface of the material.
- For the dielectric backing material (c) in examples 1, 2, 3, 4, 5, 6, and 7, you can substitute any other dielectric material that will not be too readily attacked by the electrolyte and in particular the alkaline solution that will tend to form in contact with the cathode, such as glass, polyethylene terephthalate, poly(methyl methacrylate), polymerized linseed oil, shellac, polyethylene, polypropylene, epoxy resins, teflon, fluorinated ethylene propylene, other polyester resins, aluminum oxide, or other metal oxides, among many others. A stack of such layers may be useful. Extremely inert backing materials such as teflon introduce the problem that firmly adhering the cathode to them with the PVA glue (d) or its alternative may be more difficult; stacking a readily adherable material such as HIPS on top of a more inert material such as polyethylene is one possible solution, and welding the backing (c) to the glue (d) will also improve adhesion in difficult cases.
- For the fine copper conductors (a) in examples 1, 2, 3, 4, 5, 6, 7, and 8, you can substitute nearly any other conductive material at all as long as it's sufficiently cathodically protected; for example, copper, aluminum, gold, silver, platinum, palladium, rhodium, tantalum, niobium, vanadium, molybdenum, graphite, glassy carbon, non-glassy amorphous carbon, nickel, stainless steel, chromium, lithium metal, sodium metal, or ordinary steel, or mixtures of these, among hundreds of other possibilities, in the form of fine wire, foil, thin film, or plating. Some of these possibilities rule out the use of certain electrolytes; for example, sodium metal probably cannot be used in contact with water regardless of how well it's cathodically protected. The use of nobler metals such as tantalum and gold does not affect the anodic dissolution process and permits the use of more aggressive electrolytes. Thinner metals such as gold leaf, especially together with thinner separator layers and thinner workpieces, permit finer patterning of the workpiece. Additionally, you can provide the pattern instead by using a continuous layer or mesh of any of these materials as the cathode, superposed on a selectively nonporous mask of some dielectric material, such as the laser-printer toner mentioned earlier or materials such as those listed in example 9 above.

- In examples 1, 2, 3, 4, 5, 6, 7, and 8, as an alternative to a pattern (a) supported on a dielectric backing (c) as described above, you can use a conductive plate (i) made out of any conductive material such as those mentioned in example 9 above with a selectively patterned impermeable dielectric “stop-off” or “mask” (j) on it, so that electrolysis can only proceed where the mask is absent or at least porous. For example, you can use sheet steel or any other sheet metal with nail polish selectively painted onto it; or a dielectric photoresist deposited on it and optically patterned in the way that is common for fabricating integrated circuits or printed circuit boards; or laser printer toner transferred onto it; or a dielectric coating selectively deposited by inkjet printing, perhaps then baked to improve the coating; or “permanent” marker ink; or “dry erase” marker ink; shellac (an idea due to Mina); cellophane tape; paraffin; powder coat paint, as commonly used for painting industrial machinery; glass, as in cloisonné; dried soluble silicates, if heated between uses to drive out excess water; a layer of a passivating compound formed from the surface of the conductive plate (i) itself, for example by heating or anodizing; polymerized linseed oil; photoresists; teflon; rosin; spray paint; shellac; or any other dielectric that is sufficiently resistant to the electrolyte. Many of these dielectrics can be applied in a continuous layer and then selectively removed by laser ablation, for example with a low-wattage laser cutting and engraving machine like those commonly used for cutting MDF, or by some other method such as stamping, grinding, abrasive jet blasting, or scraping. The mask (j) can be a separate removable layer rather than firmly adhered to the conductive plate, as in the earlier example of laser-printed paper; the screens used in silkscreening or the waxed fabric in batik would work well for this. A nonwoven thermoplastic cloth can combine the functions of the mask (j) and the electrolyte bearer (f) by being melted in the regions to be “masked”, rendering it nonporous, as is commonly done to join nonwoven thermoplastic cloths.
- In example 10, instead of protecting parts of the pattern electrode surface (i) with a solid dielectric, you can protect parts of the pattern surface by recessing them far enough that when the conductive pattern plate is brought into contact with the electrolyte-soaked porous material (f), the recessed parts are separated from it by an air gap.
- In example 10, instead of protecting parts of the pattern electrode surface (i) with a solid dielectric, you can cut spaces in the electrolyte-soaked porous material (f), or otherwise pattern it to fill only a part of the space between the two electrodes. For example, a thin stranded string of fiber can be shaped into the desired pattern, moistened with electrolyte, and squished between the two plates before applying the power.
- In examples 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10, if you use an electrolyte that can dissolve the aluminum workpiece at zero voltage, such as alkali-metal hydroxides or hydrochloric acid, you can reverse the voltage to cathodically protect the *workpiece* rather than the pattern. This will result in dissolution of the workpiece in a *positive* pattern (leaving workpiece where the pattern is present rather than where it is absent) rather than a negative one. In this case it may be convenient to first subject the whole workpiece to cathodic reduction (using it as a cathode with an unpatterned anode, possibly with a different

electrolyte) to eliminate possible passivating oxide films, before reversing the polarity. This approach can also result in the dissolution of the pattern, possibly in an uneven fashion resulting from parts of it remaining connected longer, as with the tabs mentioned earlier. (A similar consideration applies to ensuring the electrical continuity of the protected part of the workpiece until the end of the process.) This can be prevented by using a nobler material for the pattern than for the workpiece and operating at a moderate enough voltage to prevent the pattern from being attacked. Stainless steel wire or graphite is probably the most convenient pattern material in cases where copper is insufficiently noble. As an alternative to preventing this electrolytic pattern erosion, if the pattern is thick enough, you can alternate between patterning a workpiece cathode in this way, and electrodepositing new metal on the pattern to replace the lost metal.

- In examples 1, 2, 3, 4, 5, 6, 7, 8, 9, 12, and 13, as explained earlier, rather than using a dielectric backing (c) with a patterned electrode (a) on it, you could use an array (k) of independently controlled electrode pixels insulated from one another.
- In examples 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, and 12, rather than *dissolving* the workpiece, you can deposit oxides or other insoluble metal salts on its surface by a suitable choice of electrolyte (g), known as "anodizing". This can be used, among other things, to selectively passivate it for future use as a pattern electrode (as mentioned in example 10), to selectively passivate it to selectively resist some other etching process or reaction, to selectively harden the surface, to color it with opaque compounds, or to color it with iridescence by controlling the layer thickness, a process which is less effective for aluminum than it would be with some other metals. It may be possible to modulate the current density over time to spatially modulate the density of the oxide layer to form a rugate filter.
- In examples 1, 2, 3, 4, 5, 6, 7, 8, 9, and 14, instead of a dielectric backing (c) with a patterned cathode, you can use one or more movable cathodes (m) that electro-etches the anode workpiece (h) where it touches the porous material (f) and not elsewhere. Some useful forms of patterned cathode for this purpose might include one or more a narrow rollers like pizza cutters, which cut along a line rather than at a single point while exerting minimal friction on the porous material (f); one or more needles which are touched to the surface of the porous material (f) at different points at different times; a metal ball like that used in ballpoint pens and ball bearings, which can roll like the pizza cutters; outlines of various forms, such as circles and semicircles of different diameters, the edges of razor blades, the whole shapes of parts, logos, letters, cartoon characters, and halftone patterns, which can be placed at different points on the material at different times and etched to varying depths. These "stamps" can be made in many different ways, including engraving or etching a solid metal or graphite surface, and bending wire. A soft wire brush is another candidate cathode, as in brush electroplating. The roller approach and the seal approach can be combined in a rolling seal. A wire or metal tape can be used to etch a straight line of variable length all at once, either by hand or under the control of a machine similar to an old automatic wire-wrap machine.
- In example 15, the porous material (f) can be attached to the

movable cathode or cathodes (m) rather than to the workpiece (h), and the pattern can be in the porous material rather than the cathode, as in example 12.

- In examples 1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, or 16, the aluminum sheet (h) can be replaced by a sheet of almost any other metal or conductive ceramic with a corresponding change of electrolyte (g), though platinum and a few related substances are considered impossible to dissolve anodically. This opens up more interesting anodizing possibilities, such as bluing steel, or depositing iridescent layers on metals with transparent oxides with high refractive indices such as titanium. This cutting process is particularly appealing for very hard metals, ceramics, and cermets.

Patterning the "paper"

In the cases where the porous material (f) is held fixed relative to the workpiece (h), the result of the process includes not only the etching of the anode (h) but also metal ions impregnated into parts of the porous material (f). In some cases this directly produces a visible pattern on the paper or similar material, but whether it does or not, onchrome or afterchrome dyeing can be used to produce a permanent, colorfast pattern. Mordants commonly used for dyeing include salts of copper, tin, iron, aluminum, chromium, and tungsten, so electro-etching anode metals containing these metals can be used to selectively mordant a textile in this way — either one that is dyed afterwards ("onchrome") or one that is pre-impregnated with the dye ("afterchrome") and then merely washed to reveal the pattern. For this it is necessary to use a dye that cannot be effectively mordanted with whatever cations are present in the electrolyte before electro-etching; I think azanium ions are safe for all dyes.

Ferrous (Fe^{2+}) ions from the photodecomposition of organic ferric (Fe^{3+}) complexes (ferrioxalate, ferricitrate, ferric oxalate, ferric tartrate) are commonly used in this way to tone siderotypes with various kinds of vegetable pigments, as explained in Mike Ware's *Cyanomicon* §3.5 (p. 77). It also explains the possibility of using the ferrous ions to "reduce the compounds of a 'noble' metal, such as platinum, palladium, silver or gold, to the metallic state", which also leaves an indelible mark in the paper. Below about pH 9, the Pourbaix diagram for iron says there's a wide stability range for ferrous ions; unfortunately this range extends below 0 V, to about -0.6 V, so if we look only at the equilibrium, this range can only be used directly in the examples above in the "positive" process, where the workpiece is a cathode that is dissolved *except* where it is cathodically protected. However, I think that in practice the spontaneous reaction rates may be low enough to permit the use of the "negative" process where the iron electrode is anodically dissolved, particularly if there's a little bit of postprocessing to cathodically draw some of the dissolved iron out of solution.

It would be exciting to be able to do the same trick with an ion that could be further oxidized to reduce something a little less noble (and expensive) like copper. But I can't think of anything.

Imaging is not the only way of using this selective ion impregnation. The polyvalent cations thus obtained can selectively catalyze other reactions if the porous material is supplied with the

reagents, and they can (noncatalytically) harden soluble silicate and phosphate solutions, which can thus selectively stiffen the porous material, thus forming a ceramic/fiber composite object with flexible blades of fibrous material joining stiff blades of fiber-reinforced ceramic.

Ware's book also reports (§4.2 "Pellet's process", p. 89):

> The new feature that Pellet introduced to solve the problem of fixing the positive-working process was based on earlier observations by Alphonse Poitevin in 1863, that ferric salts cause gum and similar colloids to harden and become insoluble in water, whereas ferrous salts do not.

In §4.10 on p. 113 he explains the gel lithography process:

> Gel lithography was also variously known in its hey-day by several proprietory [sic] names: Lithoprint, Ferro-gelatine, Fotol, Ordoverax, Velograph or Fulgur printing. The method took an over-exposed, but unprocessed negative blueprint image as its source, which was lightly squeegeed into contact with a matrix of moist gelatine, known as a "graph", containing a ferrous salt. Diffusion of excess potassium ferricyanide out of the lightly-exposed and unexposed regions of the cyanotype (the image shadows) formed Prussian blue in the gelatin matrix, with the effect of hardening it locally (see §4.1.1), where it became receptive to a greasy lithographers' ink, which was still repelled by the moisture in the unhardened regions of the gelatin. After a minute or so, the cyanotype was peeled off and the jelly surface inked with a roller. About 25 positive copies of the original image could be 'pulled', using little pressure, from the jelly, which was re-inked between each.

I'm not sure whether the effect of hardening the gelatin was due to the complexed ferric ions in the ferricyanide (but Ware asserts that it is, and we can probably trust him) or some other effect of forming the Prussian blue.

Faraday efficiency and energy usage

Aluminum has only one common oxidation state (3+), so the Faraday efficiency of the setup should be near perfect with aluminum. It's 27.0 g/mol. At 96485.34 coulombs per mole, times three, we have 10.7 megacoulombs per kilogram:

```
You have: avogadro 3 e / (aluminum g/mol)
You want: MC/kg
        * 10.727928
        / 0.093214641
```

At an ordinary electroplating current density of 10 A/dm² and 2.70 g/cc this is almost a year per meter:

```
You have: avogadro 3 e / (aluminum g/mol) / (10 A/dm^2) * 2.70 g/cc
You want: days/m
        * 335.24776
        / 0.0029828685
```

This works out to only 35 nm/s, which would take 300 seconds to cut through a 10 μm household aluminum foil. We ought to be able to use a much higher current density for electrochemical machining because we don't have to worry about forming dendrites, but the 1000 A/dm² we'd need to do the cut in three seconds sounds pretty

extreme. Is it?

Considering the 100-μm-diameter copper wires I was talking about at the beginning, how much current are we talking about? Suppose one such wire is 100 mm long; it then covers an area roughly 100 μm by 100 mm on the porous medium (f), 10 mm². Then 1000 A/dm² would be just 1 amp. Suppose we're feeding it from both ends. At a nominal copper conductivity value of 58 siemens m / mm² (from definitions.units) this works out to 2.20 Ω/m and thus 0.11 Ω in the 50 mm; if all the current came out in the middle it would be 500 mA in each side through that 0.11 Ω, with a resulting voltage drop of about 55 mV, which is probably bearable. But actually the current is hopefully coming out evenly along the length of the wire, so the situation is a little better, with that 500 mA though 2.20 mΩ/mm initially dropping 1.1 mV/mm, but linearly dropping to 0 mA and thus constant voltage in the middle of the wire. Without actually doing the algebra, I think this works out to a voltage drop of 27 mV.

This is a very reasonable voltage drop. I think it also works out to about 7 mW, which normally would be a large enough power to worry about in a tiny wire like this, but maybe not when it's immersed in water.

The standard electrode potential of reducing $2H_2O$ to H_2 and $2OH^-$ is -0.8277 V (per electron), and that for oxidizing Al to Al^{3+} + $3e^-$ is -1.662 V (per electron). If I understand this stuff right, which I might not be, that means you need at least 834 mV between the electrodes before you start electro-etching the aluminum. This is a very easy voltage to supply and implies that the overall power needed to do these cuts is only about 800 mW, plus whatever gets wasted on Joule heating of the electrolyte and the cathode (about 3.4% in the electrode in the above example). If you have something in the electrolyte that's more likely to deposit on the cathode than sodium or aluminum — copper, say — then you might not have to pay even that much; but then your cathode becomes less precise.

If we use the conductivity of seawater, 50 mS/cm and an electrolyte path of 100 μm, we get 2 Ω:

```
You have: 100 um / (50 mS/cm * 100 um 100 mm)
You want: ohms
        * 2
        / 0.5
```

At 1 A this would be a joule-heating voltage drop of 2 V, giving a total of 2.859 V: 2.000 V in the electrolyte, 0.832 V in the electrolytic interfaces, and 0.027 V in the wire. The conductivity is proportional to the ion mobility, the ion concentration, the ion charge, and the temperature (≈2%/°); with more concentrated solutions, and concentrations with highly mobile ions (hydronium beats sodium 7×), we ought to be able to get it down to 0.2 Ω and thus 0.2 V, so that even at 1000 A/dm² (100 mA/mm²) we spend 80% of the energy on electrolysis. And of course at lower currents the ohmic losses become insignificant.

At significantly higher currents the voltage drop along the wire would become sufficient to provoke different electrolytic reactions in different places, which is not the desired effect. This would also produce different current densities in different places, and thus reaction speeds, cutting speeds, and potentially cut widths; a higher-resistivity electrolyte will tend to avoid this problem, at the expense of wasting more energy as heat.

A power supply that can produce 3 V at 1 A is straightforward to cobble together from common components; in the most primitive form, two resistors and a power-transistor emitter follower can produce this from many USB chargers, though it would produce a lot of heat. A more efficient switcher design is also not very demanding and would be a lot safer.

So in fact cutting through hand-sized aluminum foil in a few seconds with submillimeter precision is eminently attainable, and should be reasonably efficient, using minimally 8 mJ per millimeter and realistically 30 mJ/mm. If you could manage a thinner kerf, it could be even more efficient. Scaling the cutting up to higher speeds, larger workpieces, or very complex cuts might start to be a challenge, though.

Workpiece materials

Different metals require somewhat different amounts of current, but the density of the electron gas you're sucking out of the metal doesn't vary nearly as much as other properties of metals such as hardness, toughness, mass density, and electronegativity; here are my calculations for a selection of metals including the common ones (excluding the air-unstable sodium, potassium, calcium, strontium, and barium and the brittle manganese):

metal	molar mass	density	valence	current required	melts
Silver	107.868 g/mol	10.49 g/cc	1	9.38 A/mm^2/(mm/s)	1234.93 K
Gold	196.967 g/mol	19.30 g/cc	1?	9.45 A/mm^2/(mm/s)	1337.33 K
Lead	207.2 g/mol	11.34 g/cc	2	10.6 A/mm^2/(mm/s)	600.61 K
Tin	118.710 g/mol	7.265 g/cc	2?	11.8 A/mm^2/(mm/s)	505.08 K
Zirconium	91.224 g/mol	6.52 g/cc	4	13.8 A/mm^2/(mm/s)	2128 K
Magnesium	24.305 g/mol	1.738 g/cc	2!	13.8 A/mm^2/(mm/s)	923 K
Titanium	47.867 g/mol	4.506 g/cc	4?	18.2 A/mm^2/(mm/s)	1941 K
Zinc	65.38 g/mol	7.14 g/cc	2	21.1 A/mm^2/(mm/s)	692.88 K
Iron	55.845 g/mol	7.874 g/cc	2?	27.2 A/mm^2/(mm/s)	1811 K
Copper	64.546 g/mol	8.96 g/cc	2?	27.2 A/mm^2/(mm/s)	1357.77 K
Aluminum	26.98 g/mol	2.70 g/cc	3	29.0 A/mm^2/(mm/s)	933.47 K
Cobalt	58.9332 g/mol	8.90 g/cc	2?	29.1 A/mm^2/(mm/s)	1768 K

Nickel 58.693 g/mol 8.908 g/cc 2? 29.3 A/mm²/(mm/s) 1728 K
Molybdenum 95.95 g/mol 10.28 g/cc 3? 31.0 A/mm²/(mm/s) 2896 K
Chromium 51.9961 g/mol 7.19 g/cc 3 40.0 A/mm²/(mm/s) 2180 K
Tungsten 183.84 g/mol 19.3 g/cc 6? 60.8 A/mm²/(mm/s) 3695 K

(The unit A/mm²/(mm/s) is equivalently A·s/mm³, GA·s/m³, or GC/m³, but I find these units less intuitive.)

This ordering more closely aligns with those of malleability, ductility, and hardness than with any other property I can think of: gold is the most malleable metal, very nearly the fastest cutting, and soft enough to dent with your teeth (as are lead and magnesium), while tungsten is the brittlest and nearly the hardest, and chromium is actually the hardest and also quite brittle.

If you wanted to design a material to be more rapidly cut by ECM, you'd probably want a composite of two or more phases, such that most of the volume of the material was in a discontinuous phase cemented together by a metallic continuous phase, and you could electrolytically cut the continuous phase without having to cut the discontinuous phase. The discontinuous phase might be a liquid or gas, making the material a gel or foam; it might be some other conductive substance, such as a metal with a more positive electrode potential, in which case it would need to be physically removed from the cut for it to proceed; or it might be an insulator. In any case the grain size of the discontinuous phase would need to be smaller than the desired cuts. A metal volume fraction of 15%, corresponding to a 6× ECM speedup, seems reasonable:

> There is some overlap between [metal matrix composites] and cermets, with the latter typically consisting of less than 20% metal by volume.

See below for notes on suitable solid nonconductive reinforcing discontinuous phase materials. Foams are appealing for increasing stiffness without increasing mass or cutting time.

Zirconium is particularly appealing as an electrolyzable matrix material here; though it is not as abundant as iron, aluminum, magnesium, or titanium, it is more abundant (in Earth's crust) than copper, zinc, nickel, chromium, tin, lead, molybdenum, or tungsten, on par with carbon or vanadium; even as a pure element it is about as strong as steel (230 MPa yield stress, 330 MPa ultimate tensile strength, with Young's modulus of 94.5 GPa; grade 705 is alloyed with 2.5% niobium to get 500 MPa yield stress, 600 MPa ultimate tensile strength, higher than Zircaloy); while being substantially less dense, having a higher melting point, and being biocompatible; and it should electrolyze twice as fast as iron, copper, nickel, or cobalt — assuming you can sufficiently disrupt its protective oxide layer during electrolysis, a problem which also arises with titanium. You could imagine a zirconium-cemented composite consisting principally of submicron grains of yttrium-stabilized zirconia (assuming cubic zirconia adheres as well to zirconium as the protective oxide layer does) that can be cut electrolytically five times as fast as steel. Zirconium also potentially supports the formation of

hardening carbide grains like those in steel, though I'm not sure if there's a way to form a pearlite-like structure in zirconium. (See Exotic steel analogues in other metals (p. 1050) for more thoughts on this theme.)

(Zirconia is notable for its electrical properties, but at room temperature it is an insulator, because its conductance is mediated by the mobility of oxygen ions.)

Metallic magnesium is also appealing here because it has not only a high electrolysis rate but also a very low standard electrode potential (-2.372 V) and many conveniently soluble compounds. It has alloys with reasonable strength: yield strength of casting alloys "typically 75–200 MPa, tensile strength 135–285 MPa … Young's modulus is 42 GPa." ASTM A36 steel, for reference, has yield strength 250 MPa, UTS 400–500 MPa, Young's modulus 200 GPa, so these alloys have a substantial fraction of steel's strength and (to a lesser degree) stiffness. (Pure magnesium is much weaker, only about 20 MPa, though another source says 65–100 MPa, and some wrought alloys are stronger, as high as 300 MPa yield strength.) Stiffness can be improved with discontinuous reinforcing fillers to a much greater extent than strength. Its greatest drawbacks are its inflammability, its intolerance of high temperatures (worse even than aluminum) and creep. (Fillers tend to eliminate creep.)

Suitable nonconductive reinforcing discontinuous phases

Ideally these would be in the form of submicron particles, especially submicron-length whiskers or laminae; they might include carborundum, carbon nanotubes, carbon fibers, halloysite nanotubes, other clays, boron nitride nanotubes, basalt fiber, goethite, asbestos, zirconia, zircon, sapphire, talc, cubic boron nitride, boron carbide, silicon nitride, topaz, diamond, silica, rutile, chrysoberyl, beryl, spinel, mica, aluminum magnesium boride, boron, or iron tetraboride. (Titanium nitride and zirconium nitride are too conductive.) Composites drawing most of their strength from such high-aspect-ratio functional fillers may actually benefit being bonded with a soft, malleable metal (like tin, magnesium, or zinc), rather than a harder, stronger metal (like tungsten, chromium, or cobalt), because, as with intentional weakening of the fiber–matrix bond in ceramic-matrix composites, it allows pullout, impeding crack propagation and distributing the load along the length of the fibers or plates. With this sort of nanostructure it should be possible to take advantage of the extra strength of reinforcement whose thickness is below the critical dimension for flaw-insensitivity.

Laminar functional fillers can enjoy flaw-insensitivity by having only one of their particle dimensions below the critical dimension, and can theoretically provide high strength in two dimensions, thus providing on average twice the strength of the same material as a fibrous filler, but high filler loadings for laminar fillers are only possible by aligning the laminae parallel. I saw a paper about 10 years ago which achieved this with bentonite and PVA (rather than a metal) by depositing them in alternate layers ("layer-by-layer (LBL) assembly"), but I haven't seen examples since then. (I think some of

steel's strength can be attributed to pearlite and bainite having precisely this structure, with ceramic cementite nanolayers alternating with soft metallic ferrite.) I posted the paper to kragen-fw with the headline "new high-strength composite made of "nanoclay" and PVA":

> Charles Griffiths told me about this October 4 article from Physorg, "New plastic is strong as steel, transparent":
>
> http://www.physorg.com/news110727530.html
>
> Apparently, by alternating layers of polyvinyl alcohol and "clay nanosheets", Nicholas Kotov and a bunch of other people at UMich (many from his own lab), plus some folks at Northwestern (in some earlier research; see below) have fabricated an extremely high-strength composite. It gets its strength from parallel layers of clay nanosheets glued together with thin layers (monolayers?) of PVA. ...
>
> http://www.sciencemag.org/cgi/content/abstract/318/5847/80
> doi:10.1126/science.1143176
>
> Science 5 October 2007: Vol. 318. no. 5847, pp. 80–83.
>
> The authors are Paul Podsiadlo, Amit K. Kaushik, Ellen M. Arruda, Anthony M. Waas, Bong Sup Shim, Jiadi Xu, Himabindu Nandivada, Benjamin G. Pumplin, Joerg Lahann, Ayyalusamy Ramamoorthy, and Nicholas A. Kotov, all of whom are from UMich and five of whom are from Kotov's lab.

In this work, for which Google Scholar finds 1563 citations, by crosslinking the polyvinyl alcohol with glutaraldehyde (widely sold as a disinfectant at 2–2.5% strength under names like Surgibac G and Sertex), they achieved 400 MPa strengths, stronger than many steels. They'd previously done the same thing with a mussel glue amino acid, L-3,4-dihydroxyphenylalanine, achieving lower strengths.

Electrodeposition would seem to offer a low-temperature codeposition route to fabricating such layered structures in bulk rather than a few nanometers at a time: first, compact the mass of filler to a high density, then electrodeposit a metal matrix in its interstices, similar to the molten metal infiltration technique for tungsten carbide , also used for Al/SiC metal matrix composites:

> AlSiC metal matrix composites are formed by pressure infiltrating molten aluminum into silicon carbide preforms. This method of casting is typically used in applications where solution requirements include high strength, lightweight, custom CTE and high thermal conductivity. PCC offers AlSiC with a composition varying between 30% to 74% silicon carbide by volume, depending on the application. This flexible material system allows PCC Composites to produce a part that can be tailored to exact solution requirements.

Conceivably electroless plating would work better.

For metal matrix composites or cermets, a crucial question is the adhesion of the metal matrix to the filler; as mentioned above, adhesion that is too strong can propagate cracks into the filler particles, eliminating their flaw-insensitivity, but of course in the limit of weak adhesion the composite is no better than a foam with extra dead weight.

The high filler loadings that would be ideal for electrolytic machinability are more similar to the area of practice generally known as "cermets" than to the area of practice generally known as "metal matrix composites". Nonconductive reinforcing discontinuous phases used in cermets seem to include sapphire, glucina, magnesia (periclase), zirconia, phosphates of calcium, fluoroaluminosilicate glass, rutile, boron carbide, carborundum, aluminum nitride, sodalite, and quartz.

A truly 2D material like graphene or a MXene would also make a great functional filler for this kind of thing if, like nitrides of titanium and zirconium or like the MAX phases, you could find one that isn't conductive. The problem with conductive fillers is that, once the surface of the metal is etched, they would screen the electric field from the metallic matrix surface in their interstices, so it would stop being etched.

Topics

- Digital fabrication (p. 1149) (31 notes)
- Electrolysis (p. 1158) (18 notes)
- Filled systems (p. 1161) (16 notes)
- Frrickin' lasers! (p. 1168) (12 notes)
- Aluminum (p. 1180) (10 notes)
- Composites (p. 1187) (9 notes)
- 2-D cutting (p. 1201) (7 notes)
- Magnesium (p. 1213) (6 notes)
- Anisotropic fillers (p. 1218) (6 notes)
- Poly(vinyl alcohol) (PVA) (p. 1245) (4 notes)
- Patterning (p. 1282) (3 notes)
- Glutaraldehyde (p. 1294) (3 notes)
- Codeposition

Layers plus electroforming

Kragen Javier Sitaker, 02021-12-16 (updated 02021-12-30)
(7 minutes)

2-D cutting (laser cutting, waterjet cutting, CNC plasma table, 2-D wire EDM, the electrolytic method described in Electrolytic 2-D cutting and related electrolytic digital fabrication processes (p. 1085), etc.) is a highly efficient way to make things, especially at large scales and when you have high precision. It occurred to me that by combining it with electroforming and anodic dissolution (ECM) it can be substantially more powerful.

Sheet lamination

If you want to 3-D print a 100-mm prolate spheroid that is 50 mm in its minor diameters, you need to 3-D print 131 milliliters of volume. With a typical 30% infill, this works out to 39 milliliters of actual plastic for the interior. If you use an 0.8-millimeter line width to print 3 perimeters around each layer, you have about 2.4 mm of thickness on the shell, which is 38 of those 131 milliliters, leaving only 93 mℓ for the infill, using 28 mℓ of plastic, for a total of 66 mℓ. At a typical layer height of 0.2 mm this is 138 meters of extrusion; at a typical 50 mm/s, this is 46 minutes of printing.

XXX recalculate that, it's calculated with a shell thickness of 2.4 mm which is wrong on the bottom and top

By contrast, if you instead print out the same object by sheet lamination ("laminated object manufacturing"), you only need to cut the perimeters. If you were to use the same 200-μm layer height, you would only need to make 78.5 meters of cuts, which at the same movement speed of 50 mm/s would only take 26 minutes. The resulting object is normally fully dense, which is an advantage in some contexts, since it makes it stronger and stiffer. In cases where it is not, if the inside contour of the object (where infill would have been placed) is not strictly defined, you can often hollow it out by nesting multiple layers one inside the other, avoiding the need for extra cutting time.

Often the movement speed is *not* the same; many 2-D cutting processes can run at much higher speeds than additive processes can usually manage.

This advantage increases for larger objects and decreases for smaller ones; for a meter-scale object instead of a factor of 1.8 it's a factor of 18. Larger objects tend to benefit more from being fully dense, because they need proportionally more cross-sectional area to support their own weight, or their own mass under the same accelerations.

Electrolytic welding

By electrodepositing a thin layer of metal on a metal object made by metal sheet lamination as described above, we can get several important advantages:

- The layers are connected together by the deposited metal. Although

it won't *penetrate* the layers, under some circumstances it can have sufficient adhesion to them to form a solid object. This depends crucially on their surface condition, which is more controllable in this situation (sheets freshly cut out, produced by a controlled process) than under some other circumstances.
• The alternative to connecting the lasers is to run slots or holes through many layers and put a sliding fastener through them; such fastener-based or sliding-joint construction can also be fixed "permanently" by such electrodeposition. (If the glue metal being deposited can be anodically dissolved at a more moderate voltage than the base metal, the glue metal can be selectively removed later by electrolysis, permitting disassembly.)
• The electrodeposited metal can smooth out layer lines which could otherwise interfere with appearances, fluid flow, optical performance, smooth sliding, human comfort, etc.
• The electrodeposited metal (or composite) can be a material that can't be processed by the original sheet-cutting process, or with more difficulty. For example, copper and nickel cannot be cut with oxy-acetylene torches, but you can very rapidly cut out a form from cheap mild steel on a CNC oxy cutting table, then electrodeposit them on its surface. This is especially true of nanolaminates, whose properties can be tuned to the application.
• If the originally deposited sheets can be anodically dissolved at a more moderate voltage than the newly electrodeposited metal, contrary to the anodic ungluing process described above, they can be selectively removed after the electroforming process is complete.

Subtractive 3-D printing with ECM

Electrodeposition suffers from a positive feedback process of dendrite growth, in which protuberances on the surface are exposed to greater electric fields; as a secondary, much weaker, effect, they physically obstruct ion transport ("mass transport control") to nearby parts of the surface. Consequently small protuberances grow into larger protuberances, potentially bridging all the way to the cathode while most of the material is only thinly plated. This is exacerbated by the anisotropic nature of crystal growth; as I understand it, many "brighteners" used in electroplating work by introducing grain boundaries to prevent the creation of large crystals. Others work by reducing ionic flow so that deposition rate is limited by ionic concentration rather than electric field.

Electrolytic cutting or electropolishing suffers no such effect (on the workpiece); instead of causing small irregularities in the surface to *grow* faster, it causes them to *shrink* faster, making the feedback *negative*. This permits the usage of much larger currents and correspondingly larger material removal rate.

So you can get faster free-form fabrication by alternating electroless plating with anodic removal of the unwanted part of the deposited layer, for example using a movable array of separately controlled cathode electrodes, each removing a controlled amount of material in a particular area of the workpiece. By limiting the degree to which the workpiece is dipped into the electroless plating bath, you can prevent material from depositing elsewhere on the workpiece than the current layer, enabling a layer-by-layer printing process. With

electroless codeposition you can even print in a metal-matrix composite, and with the right choice of baths you can switch between two or more different baths to produce a nanolaminated material.

This procedure, as described, requires moving the workpiece back and forth between an electroless plating bath and a second bath where it is electrolytically cut by cathodes in precise positions relative to it (which may be moved in the process). You probably cannot use a single bath because the electroless plating solution will probably all plate out on your cathodes; you might be able to find a cathode material that doesn't do this, but it may be difficult. An alternative using a single bath is to alternate between selective electrodeposition (at higher current densities than are normal for electroplating) and selective electropolishing (to smooth out any irregularities in the surface that are unwanted). This permits much finer layers and eliminates the dead time and material loss between the two baths.

Unfortunately, both the negative feedback in electropolishing and the positive feedback in electrodeposition increase with the resistivity of the electrolyte.

Topics

- Digital fabrication (p. 1149) (31 notes)
- Electrolysis (p. 1158) (18 notes)
- Welding (p. 1181) (9 notes)
- ECM (p. 1186) (9 notes)
- 2-D cutting (p. 1201) (7 notes)

MOSFET body diodes as Geiger counter avalanche detectors?

Kragen Javier Sitaker, 02021-12-17 (updated 02021-12-30) (1 minute)

MOSFET body diodes are PIN diodes, according to Characterization of body diodes in the-state-of-the-art SiC FETs -Are they good enough as freewheeling diodes?. The paper is about carborundum FETs, but presumably this is true of silicon FETs too. I suspect this means that they would make usable ionizing particle detectors, perhaps even in reverse-biased avalanche mode; power MOSFETs are commonly very robustly constructed with very low capacitance between the source and drain, increasing the chance that they could survive such treatment.

Silicon MOSFETs would probably be better for this than GaN HEMTs or carborundum FETs, because carborundum's higher critical breakdown field strength is 7x higher than silicon's, permitting the carborundum device to be much smaller for a given maximum voltage rating. This, in turn, means a smaller area over which to capture particles.

This may be an appealing alternative to purpose-built PIN diodes for detecting ionizing particles, especially in places or times with supply-chain weaknesses and breakdowns, because power MOSFETs are very widely available, both as discrete parts easily salvaged from broken equipment (with sufficiently powerful soldering irons) and, because they are often the first part of the equipment to fail, as replacement parts.

Topics

- Electronics (p. 1145) (39 notes)
- Physics (p. 1157) (18 notes)
- Sensors (p. 1191) (8 notes)
- X rays (p. 1310) (2 notes)

The user interface potentialities of a barcoded paper notebook

Kragen Javier Sitaker, 02021-12-18 (updated 02021-12-30)
(6 minutes)

(Posted originally on the orange website.)

I wonder what value you could add to this [service for building websites by writing on paper] with a digitally referenced notebook? By printing an unobtrusive sort of barcode on each page, you could determine which part of which page of which notebook each scanned pixel came from, and what lighting conditions it was photographed under. What could you do with that?

Well, the simplest and most greyface application is forms; you can define particular areas of each page as being particular form fields. If you're blogging, you might have a field for a "slug" that appears in the URL, for example, or a field for tags, or checkboxes for some tags (plus a special page to declare the meanings of the checkboxes). Or, if you're tracking expenses, you could have a checkbox for each expense category and columns for the date and the amount.

For me, the special feature of paper notebooks that cellphones and other computers suck at is drawing. If I want to draw a diagram or illustration, it just works much better on paper: my pencil point occludes much less drawing area than my finger does, there's no tracking error where the ink appears 2 mm to the side of the point, it has much lower latency, and I can draw finer lines. But scanning those drawings into a computer is a pain, because I have to illuminate them evenly and hold them flat while I photograph them, which still probably involves some perspective distortion. Barcodes on the paper, together with reference lines and reference color swatches, could solve that problem, as well as providing information about which parts of the paper are occluded, if any.

For a few special applications like numismatics and entomology, the paper could provide a precise physical measurement reference for specimens.

Combining drawing with filling out forms, you can make a font from your handwriting; this is enormously easier if you can correct the various distortions. In http://canonical.org/~kragen/oilpencil/ I spent about 24 hours fiddling with various graphics programs, but there was a website I found somewhere where you can print out a form, draw the font on it, upload the scan, and download your TrueType font. This kind of thing might help with training OCR, too, especially if you don't have access to GPT-3. (Or if OpenAI decides to peremptorily destroy everything you've built because one of your users uses your service to write about their dead fiancee: https://towardsdatascience.com/openai-opens-gpt-3-for-everyone-f0b7fed309f6)

Other ways to combine drawing and filling out forms include sketching orthographic projections to build 3-D models; coloring a coloring book; drawing maps for Minetest and similar grid-cell

games (especially 2-D ones); drawing heightfields; and sketching different keyframes of an animation to automatically morph between. You could even draw a 2-D continuum of keyframes, thus providing an animation character that's continuously variable along two different axes; you might put time on the theta axis and some sort of emotion along the radius axis.

(You can also apply these ideas with drawings that are input via other media, such as touchscreens, Wacom tablets, and mice, not only scanning paper. When you're scanning paper it's hard to get feedback as you're drawing, although you could maybe glance at your cellphone screen periodically, or use a projector like DynamicLand, or have a continuously updated monitor using a webcam feed. It could even use the occlusion information from the barcode to patch in remembered images wherever your hands were occluding the paper.)

What should the barcodes look like?

In 02001 Anoto announced their "Digital Paper" approach: https://www.wired.com/2001/04/anoto/. As explained in https://en.wikipedia.org/wiki/Digital_paper this uses an unobtrusive 2-D barcode scanned by a camera in a "digital pen" (later called the "Fly Pen", 02005) to locate the pen in an enormous global "virtual desktop"; I think the NeoLAB "Neo smartpen" works the same way. This was all before cameraphones went mainstream and high resolution. They got 300 patents but fortunately everything they filed in 02001 expires this year. Anoto's barcodes use a grid of slightly displaced grid dots.

The Fly Pen provided a sort of graphical user interface on the paper, using audio for output. It was sort of aimed at kids doing schoolwork and playing games. It failed in 02009. The founder started a new company called Livescribe focusing on notetaking; the Livescribe smartpen allows you to spatially organize and annotate a continuous audio recording. It has been more commercially successful: https://en.wikipedia.org/wiki/Livescribe

Tiny unobtrusive dots might not reproduce reliably on a cellphone camera, though having been published in WIRED in 02001 means the technique is in the public domain (or will be next year). A better idea might be to use thin horizontal and vertical grid lines whose thickness varies slightly, perhaps in a pastel subtractive primary like cyan, magenta, or yellow; then you can optionally remove them in software after scanning. Scanning a whole page at a time, instead of a tiny area around a pen point like the "digital pens" described above, gives you a great deal more space for redundant page ID data in the barcode; probably 48 bits or so is sufficient.

Topics

- Human-computer interaction (p. 1156) (22 notes)
- Barcodes (p. 1385) (2 notes)

Aluminum refining

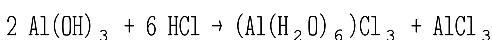
Kragen Javier Sitaker, 02021-12-20 (updated 02021-12-30)
(3 minutes)

Methylsulfonylmethane, a nontoxic useless dietary supplement the FDA deems “generally recognized as safe”, melts at 109° and can reportedly dissolve aluminum trichloride, from which a new paper reports that aluminum metal can be electrolytically deposited under an anhydrous argon atmosphere in a glove box (I think the paper said at 140°). (Aluminum trichloride alone reportedly sublimates at 182.7°, so some kind of solvent is needed.)

Aluminum trichloride is itself relatively safe; PubChem says it’s in antiperspirants at up to 15% concentration, though it “may cause severe irritation to eyes” and is “toxic by ingestion” (hexahydrate LD_{50}: 3311 mg/kg rat (oral); anhydrous 380 mg/kg). A tricky thing is that it’s maybe best to consider the anhydrous form as a separate compound from the hexahydrate — if you heat the hexahydrate ($(Al(H_2O)_6)Cl_3$), it doesn’t dehydrate to the anhydrous form, but rather releases muriatic acid and forms alumina at 100°! The Merck Index says, “Fumes in air, strong odor of HCl; when heated in small quantities volatilizes without melting. Combines with water with explosive violence and liberation of much heat.”

It’s normally produced from aluminum metal, as explained in US patent 3,343,911, though US patent 3,760,066 documents a process for producing it from chlorine, carbon, and alumina at 800°–1000°, part of the patent office’s classification “C01F7/60 - Preparation of anhydrous aluminium chloride from oxygen-containing aluminium compounds”.

At the risk of making a fool of myself by trivializing a patent classification with 16 patents in it (all expired: 01919, 01923, 01925, 01926, 01927, 01928, 01932, 01964, 01971, 01973, 01975, 01977, 01978, 01982, 01990, and 02004), this reaction seems like it would be pretty straightforward:


$$2\ Al(OH)_3 + 6\ HCl \rightarrow (Al(H_2O)_6)Cl_3 + AlCl_3$$

That is, if you react dry aluminum hydroxide with anhydrous muriatic acid, you should get a 1:1 molar ratio of the hexahydrate and the anhydrous, which you then must separate. Without using water! Worse, you probably don’t really have this neat separation of water-complexed aluminum ions from non-complexed ones; I think you’ll get polymeric aluminum chlorohydrate.

But I bet that if you heat up the mixture to drive off the HCl you’ll end up with a mix of anhydrous aluminum chloride and alumina, and I bet the alumina is insoluble in methylsulfonylmethane.

This is particularly interesting because purified $Al(OH)_3$ is the result of the Bayer process by which aluminum is electrowon today — but by dissolving alumina in cryolite, rather than aluminum chloride in methylsulfonylmethane. Crudely we should expect this

process to be cheaper because it runs at such a low temperature. Could this be a new economic route to aluminum refining?

It would be even more exciting if it turned out to be possible to do the same trick with, for example, magnesium chlorate, magnesium formate, magnesium perchlorate, or titanium tetrachloride.

(Thanks to Mina for help with the calculations!)

Topics

- Materials (p. 1138) (59 notes)
- Electrolysis (p. 1158) (18 notes)
- Aluminum (p. 1180) (10 notes)
- Refining (p. 1335) (2 notes)

Regenerative muffle kiln

Kragen Javier Sitaker, 02021-12-21 (updated 02021-12-30)
(19 minutes)

If they are not to be huge, pottery kilns necessarily use a largish amount of power, because during the peak temperatures of the firing, they need to maintain their contents at 1000° or so above ambient temperature. Under conservative assumptions about the kiln walls (≈0.1 m², ≈0.1 m, ≈0.5 W/m/K) this requires hundreds or thousands of watts (500 W in this case; cutting it to 100 W would involve thickening the effective wall thickness to 0.5 m — over a cubic meter for an internal volume of just 3 ℓ = spherevol(sqrt(.1m^2/4pi))) or using refractory insulating vacuum panels which, as far as I know, don't exist. A kiln the size of a domestic microwave oven requires more like 5 kW and thus cannot run on a regular 120 VAC circuit; it would need close to 42 amps.

This can be a problem for residential power consumption and wiring installations. 5 kW is a lot for a single circuit even in 240VAC countries — though not unheard of, it typically requires a purpose-installed circuit, which may not be an option for residential renters.

Firing with fire

Traditionally pottery was fired with actual fire. CO_2 has $\Delta_f H^{\ominus}{}_{298}$ = -393.5 kJ/mol and 12 grams of carbon per mole, so burning 12 grams of carbon gives you 393.5 kJ, 32.8 kJ/g. So 5 kW of carbon burning is only 152 mg per second, about two hours per kilogram of carbon. Unlike 42-amp electric circuits, 5-kilogram bags of carbon are readily available at convenience stores around the world, and refining pure carbon from biomass for fuel has been common practice since long before recorded history.

The combustion of small amounts of fuel can be readily controlled with much smaller amounts of electrical power, on the order of 1–10 W.

There are three main disadvantages to firing with fire:

- The pottery (or other thing being treated with heat) is exposed to the gases from the fire; typically these are reducing gases with almost no oxygen and a significant proportion of CO, and there may also be ashes, nitrogen oxides, sulfur dioxide, etc.
- It's difficult to control the heating from the fire.
- The gases have to be released *somewhere*, and they are usually toxic.

These three problems can be solved in a variety of ways; here is one way.

The main combustion chamber

First, to control the amount of heat produced from the fire, you can restrict the entry of air into the fire. If the fire is in a well-insulated chamber, it can remain hot enough to stay alive for many minutes without any air input at all; even a small air input will

suffice. The necessary oxygen for 5000 W and thus 152 mg/s is 32/12 of that, or 405 mg/s; oxygen is about 21% of air, so that's about 1.9 g of air per second, which is about 1.6 ℓ/s, or 3.4 cfm in the quaint units used for fans. But less than that should be fed into the main combustion chamber itself, since the fire there will not produce entirely CO_2.

(Of course this flow should be under the control of a microcontroller, like everything else described in this note.)

The afterburner

Running the fire always in the lean conditions described above ensures that the exhaust gas is maximally reducing and thus maximally toxic, though probably devoid of nitrogen oxides. To diminish the toxicity of the exhaust gas, it should have more air injected into it in an "afterburner" to complete the combustion process; this will produce more heat but, I think, in general reduce the output temperature, approaching, I think, the adiabatic flame temperature of carbon, 2180°. By this means the reducing exhaust gas can be converted into an oxidizing exhaust gas whose only important remaining toxic component is CO_2; this process also burns off other unwanted materials like PCAHs and other components of tar, though it introduces nitrogen oxides. Further air injection after complete combustion is probably usually desirable to lower the exhaust temperature to more convenient levels.

The regenerator

As I understand it, the standard way to transfer this heat to the interior of a (non-electric) muffle kiln is to run it through pipes that heat the kiln. But the pipe walls are necessarily of fairly limited surface area, so most of the heat will not be thus transferred. A recuperator-type heat exchanger, heating gas to be fed into the kiln or recirculated through it, is another alternative, and in file `capillary-heat-exchanger` in Dercuano I explained how to build one that's orders of magnitude denser than existing microchannel heat exchangers. But what I want to explore here is regenerative heat exchangers, because those don't require exotic fabrication technology.

So you run this hot oxidizing exhaust through a regenerator chamber packed full of refractory balls of, for example, glucina, quartz, forsterite, mullite, silicon, philosopher's wool, thoria, sapphire, chromia, quicklime, fluorspar, zirconia, yttria, carborundum, or aluminum phosphate. I don't know of any other fluorides that are refractory enough, and I think almost anything that's not an oxide, fluoride, or phosphate would be oxidized under these conditions — even chromium, which would also suffer nitrogen attack. Carborundum is a special case because it forms a refractory protective oxide layer. Silicon and fluorspar are low-melting, 1414° and 1418° respectively, but the others mentioned have adequate temperatures.

Smaller balls present more surface area to absorb the heat but also leave smaller spaces for gas flow, with the result that they consume more head. High-conductivity materials like glucina (330 W/m/K at

room temperature), carborundum (360–490), or silicon (150) ease this tradeoff compared to things like sapphire (30), philosopher's wool (21), quartz (10?), forsterite (7?), quicklime (6?), mullite (4?), thoria (4?), or zirconia (2?). Unless you want to tangle with glucina, carborundum would seem to be the obvious choice (though sapphire is the traditional one); its specific heat is 0.67 J/g/K (3.2 g/cc), while sapphire's is 0.88 (4.0 g/cc) and glucina's is about 1.05 (2.9 g/cc). If we pretend that these numbers don't vary with temperature, 100 g of carborundum with $\Delta T = 1000°$ could hold 88 kJ, so mere heat capacity will not require us to use a large regenerator; each regenerator chamber might contain 10 g of balls.

Periodically you switch the exhaust to the next cold regenerator chamber; then you flush the remaining exhaust from the hot chamber with air, which is also oxidizing, and then switch the direction of airflow and its source destination: now you are flowing air backwards through the regenerator chamber, from its cold side to its hot side, from which you direct it into the kiln chamber to heat it up. The source of the air can be either the kiln itself or the outside air; in the latter case, hot air is displaced from the kiln and thus must be exhausted; this is probably the best source for air for the fire, since it is already preheated, unless it has unwanted contaminants from the things in the kiln.

The duration of heating any one regenerator chamber needs to be short compared to its heat capacity, and the balls need to be small enough that the biggest temperature gradient in the chamber is from one end to the other and not from the surface to the center of each ball.

There are many refractory metals with higher thermal conductivity and high density, including some cheap ones, but I think you would have to prevent them from ever contacting hot oxidizing atmospheres. Though it would be straightforward to ensure that the exhaust gases were always reducing, the job of the regenerator is to heat up clean air, so parts of it will inevitably be exposed to hot air. So I think it is best to simply lean out the mix in the afterburner so that the regenerator balls are not subjected to alternating oxidation and reduction.

If a higher temperature is desired, despite the higher nitrogen oxide emissions that will result, some of the heated clean air from the regenerator can be used as the afterburner air.

Exhaust remediation

The exhaust and purge air that come out of the regenerator are cool, but they still contain a lot of carbon dioxide, and probably nitrogen oxides as well. Sometimes you can just send it up a chimney or out a vent, as people do with hot-water heaters and stove hoods. In cases where this is problematic, a carbon dioxide scrubber can be used, though it may need to be rather large: running at 5000 W = 557 mg/s CO_2 for 12 hours produces 24 kg of CO_2, which would need to be stored in the scrubber. A simple soda-lime scrubber contains 56 g/mol of CaO and can fix 44 g/mol of CO_2, so you'd need 31 kg of soda-lime, which you would then have to discard. Other kinds of scrubbers might be smaller, since they can recycle the

absorbent, but they still need somewhere to store the CO_2.

On the plus side, I think soda lime will also slurp up nitrogen oxides and sulfur dioxide. With other kinds of scrubbers, diesel-engine-style selective catalytic reduction can remove nitrogen oxides.

Regenerator cleaning

If the fuel is coal or charcoal, periodically the regenerators will get clogged with ash, so they need to be cooled and washed periodically with acid; vinegar should be perfectly adequate. If you used quicklime for the regenerator balls, you'd have to just empty and refill the regenerator, since quicklime is a significant component of ash, and anything that would remove it chemically would also remove the balls.

Alternatively, the kiln could be operated on a liquid fuel such as gasoline, diesel fuel, or vegetable oil, or on combustible gas, which would not produce any significant amount of ash. Then, the only possible regenerator contamination would be partially burned fuel, which would burn off fairly quickly, though possibly damaging protective oxide layers in the regenerator in the process.

Afterburner or no? Run the main combustion chamber lean?

If the kiln thus runs on liquid fuels, you probably don't want all the fuel to be in the main combustion chamber — first, because having several kilograms of hydrocarbons above their flashpoint in your kitchen is a terrible idea; second, because it would make the kiln take quite a while to start, because you must heat them above their flashpoint first; and, third, because this allows you to run its combustion chamber lean (with excess air) and thus dispense with the separate afterburner. This approach may be worthwhile even with solid fuel, for example passing ground charcoal through a paddle wheel that prevents reflux of combustion into the fuel supply; this would also enable the main combustion chamber to be smaller, speeding up startup and shutdown.

One possible benefit of a separate afterburner is that it, like a steam-engine injector, it can provide suction. Its output is hotter and more voluminous than its input, so if the output channel is wider than the input channel it might be able to fluidically drive the other fluid flows in the kiln with suction, permitting the control system to work merely by constricting valves. Injecting water into the main combustion chamber would produce shift gas, shifting a potentially large amount of the combustion into the afterburner, which may be useful for this purpose.

Materials

The main combustion chamber and the afterburner are potentially exposed to temperatures of 2000° or more, and at least the temperature of the interior of the kiln (normally 1000°–1500°); moreover, they are exposed to both oxidizing and reducing conditions. These are the most challenging conditions, and in the best

of conditions, they probably require regular replacement of the refractory surfaces in question.

Forsterite sand (melting point 1890°) is commonly used in foundry practice as a refractory, and even cristobalite (quartz's ghost) doesn't melt until 1713°. Mullite remains solid up to 1840°, sapphire to 2072°, larnite (the calcium analogue of forsterite) to 2130°, quicklime to 2613°. Sands of these minerals (all dirt cheap except for maybe mullite) would work as replaceable refractory floors for these chambers, maybe introducing clean air through them. This will protect the floor from ground-up or lump coal, but for the ceiling and walls you still need something more solid.

The cheapest solution would be wall and ceiling tiles of quicklime, or if the chamber is small enough, just a single block of quicklime. It might be harmless for the quicklime to be produced in situ from calcium carbonate, but in that case it probably needs some kind of refractory fiber running through it to keep it from crumbling as it calcines. Larnite (dicalcium silicate, Mohs 6, 2130°) would probably work well for this. I think it can be conveniently prepared in a beaker from calcium chloride and sodium silicate, but I don't know how you'd bond the resulting grains together afterwards.

Except in case of malfunction, the rest of the kiln is only exposed to temperatures a little higher than the maximum payload temperature, and exclusively oxidizing atmospheres, so it can be made from almost any oxygen-resistant refractory material, including those mentioned earlier for the regenerator balls.

(The high-temperature zone is another argument in favor of running the main combustion chamber lean and dispensing with the afterburner: run it lean enough and you only need to deal with those same lower temperatures.)

Even this zone of moderate heat is too aggressive for organics or most metals, since it needs to operate at 1000° in oxidizing atmospheres, and the regenerators are exposed to constant thermal cycling. Some sort of ceramic is unavoidable; insulating firebrick might be the easiest choice, and probably one of the least damaged by thermal cycling.

The only necessarily hot mechanical parts are the valves on the hot side of the regenerator chambers, which probably operate pretty often. But they probably don't need to be super great valves; a little leakage would be tolerable. Incoming air can be driven with muffin fans, maintaining the whole apparatus at a slight positive gauge pressure to prevent any need to ever run hot air through any sort of blower or pump.

Control system

You need a microcontroller to control everything, which needs, minimally, an oxygen sensor on the output of the afterburner, a temperature sensor inside the kiln (perhaps just a quartz-halogen bulb used as a resistance temperature detector, or a hardware-store thermocouple of the type used to shut off pilot lights), control of at least three on-off valves per regenerator chamber or one three-way mux valve (hot side can connect to burner, purge air, payload, or nothing; cold side can connect to incoming air or to exhaust output,

but that could be just two check valves), proportional control over the incoming air fan, and, if you want to run the main combustion chamber lean, bang-bang control over the fuel injection. If you don't, you also need some kind of proportional control over the extra air injected into the afterburner chamber. So that's an oxygen sensor, a temperature sensor, nine relays or similar for the valves, and two fan speed control lines. Probably it would also be a bad idea to omit thermal overload switches like those in microwave ovens on the cold side of the regenerators.

You also need an igniter to start up the main combustion chamber, which can be done in lots of ways. If your fuel is gasoline or gas, maybe you could just use a sparkplug, but in a lot of other cases you probably want something more like a butane torch.

But that's just enough to barely work, maybe. I'd want temperature *sensors* for the input air, the output of the main combustion chamber, the output of the afterburner (if separate — another incentive to delete it), and the input and output of each regenerator chamber, and a backup temperature sensor inside the kiln chamber to do an emergency shutdown in case of sensor failure: ten more temperature channels for a total of 11. Thermocouples would probably respond faster than RTDs improvised from quartz-halogen bulbs, despite lower precision, and apparently commercial RTDs only go up to 500°, which is freezing cold.

With this collection of temperature sensors the microcontroller can slightly randomly vary the afterburner injection, the fuel flow, and the airflow, and infer immediately the derivatives of the resulting temperatures, and indeed the local linearization of the entire output transfer function — a trick that is useful for all kinds of control systems, not just this one. If there are important hidden variables, like the temperature of the sand, the humidity of the air, or the amount of payload in the kiln, you ought to be able to infer their existence and rough dimensionality from a principal components analysis or something, and then correct for them.

Topics

- Contrivances (p. 1143) (45 notes)
- Energy (p. 1170) (12 notes)
- Clay (p. 1179) (10 notes)
- Ceramic (p. 1193) (8 notes)
- Thermodynamics (p. 1219) (5 notes)
- Heating (p. 1253) (4 notes)
- Insulation (p. 1290) (3 notes)
- Regenerators (p. 1334) (2 notes)

Is liberal democracy's stability conditioned on historical conditions that no longer obtain?

Kragen Javier Sitaker, 02021-12-22 (updated 02021-12-30) (16 minutes)

As R.J. Rummel has exhaustively documented (blog), liberal democracy has substantially reduced the frequency of warfare, the fatality of wars that do occur, and the frequency and fatality of domestic mass killings such as genocides, which he terms "democides". And it seems to do this without damaging other desirable goals, such as healthcare access, technological progress, social justice, and material prosperity, at least compared to the other forms of government we've been able to observe in practice so far. Indeed, there are plausible arguments that liberal democracy is the best existing system at providing these other goods, though of course many socialists disagree, and there are clear failures, like the material prosperity of India and healthcare access in the US.

(Unlike Rummel, I think the US genocide of its Native American population, the US's current system of mass incarceration, and the mass murder in the French Congo demonstrate that democracy, even liberal democracy, is not a complete solution to the mass murder problem, just a palliative, solving most of it.)

When liberal democracy is unstable

So, if or when we can't abolish government entirely, liberal democracy is vastly preferable to the other forms of government we have so far been subjected to. But human affairs are not inevitably guided toward what is most preferable, and the relative stability of liberal democracy is contingent on two historical facts which may not obtain.

First, it can only survive where liberalism is popular. Where liberalism is unpopular, you can have liberal undemocratic rule (as in Singapore) or authoritarian democracy (as in Indira Gandhi's India), but the advent of democracy will extinguish whatever liberalism exists, and quite possibly democracy as well. Morsi's Egypt and post-Revolutionary Iran are examples of this: given the opportunity to vote, the people voted in authoritarian leaders on the strength of promises to crush freedoms; the soundbite version of the history is, "One person, one vote, one time." In the US, the same thing could plausibly happen with Donald Trump, Mike Pence, Kamala Harris, or Alexandra Ocasio-Cortez, none of whom favor liberalism.

Second, liberal or not, democracy is unstable where small elite armies can defeat large popular armies. States are institutionalized violence, so a more effective manifestation of institutionalized violence can overthrow any particular state. This can manifest externally, as conquest through invasion, or internally, as a military coup. Historically we can see that, in times and places when victory

in warfare was mostly determined by numbers of soldiers, some form of republicanism puts the population under arms, and democracy is stable if it is popular — Classical Greece with its spears and the 20th century with its assault rifles, for example. But, in times and places where a military elite of well-equipped warriors can defeat a much larger group of volunteers, frequently aristocracy or colonialism is established, and the worst abuses follow — for example, late 19th-century and early 20th-century colonialism, the post-Columbian Spanish conquests, and medieval European feudalism.

We can explain this by putting ourselves in the position of a proponent of a late-20th-century warrior faction which has just lost a democratic election, involving (as they always do) values you consider to be sacred, which are now destined to be trampled under the feet of the new government. Overturning the results of the election by violence in order to uphold those sacred values involves great personal risk (you will likely die) and, because the voters for the other side are more numerous, it probably will not succeed; they will probably beat you on the battlefield just as they did at the polls. In most cases your best option is to suffer the indignities of enemy rule and hope for a better showing in the next election.

A brief history of politics and warfare outside Asia

By contrast, consider the position of a member of the British colonial administration in the 01890s, faced with the knowledge that your policies are unpopular ("with the natives", they might say.) As Hilaire Belloc wrote satirically in 01898 in *The Modern Traveller*, p. 41:

I never shall forget the way
That [Captain] Blood upon this awful day
Preserved us all from death.
He stood upon a little mound,
Cast his lethargic eyes around,
And said beneath his breath:
"Whatever happens we have got
The Maxim Gun, and they have not."
He marked them in their rude advance,
He hushed their rebel cheers;
With one extremely vulgar glance
He broke the Mutineers.
(I have a picture in my book
Of how he quelled them with a look.)
We shot and hanged a few, and then
The rest became devoted men.

Belloc is writing about a caravan, but the same principle applies to the violent subjugation of whole countries. (The book's illustration of the "look" shows an angry white man with his arms crossed standing next to a wheeled machine-gun; Belloc's satire extended to skewering the viciousness of his countrymen, but he only barely ridicules the racism which made it so fatal, a racism he seems to have wholeheartedly embraced.)

If you have the Maxim Gun and your political opponents are Zulus armed with spears (or, as in Belloc's scathingly cynical verse, natives of Liberia, perhaps Mande or Kru), what incentive do you have to

permit them a vote, knowing that they may vote to confiscate your property, disestablish Anglicanism, persecute Christians and the Khoi-San (perhaps merely by refusing recognition to their marriages, or perhaps in more severe ways), lay heavy tariffs on your exports, slash the military budget that pays your salary, and purge the universities of professors who oppose their cause? They may not do all of these things, of course, if they are committed to liberalism; but any political faction that accedes to democracy is implicitly acceding to the possibility of being governed by their enemies, who may treat them very badly indeed.

Of course, what the British colonial powers did with this unlimited power throughout Africa was to abuse it thoroughly, even if to a smaller extreme than the Congolese colonies; the African-American colonists of Liberia did the same to the Liberian natives, for that matter. Power corrupts, and absolute power corrupts absolutely, so no government is free of corruption, and unassailable governments are enormously corrupt.

Quoting Belloc is of course epistemological malpractice: we cannot deduce anything about what really happened from his intentional fiction. But as Wikipedia says:

> The Maxim gun was first used extensively in an African conflict during the First Matabele War in Rhodesia. During the Battle of the Shangani, 700 soldiers fought off 5,000 Matabele warriors with just five Maxim guns. It played an important role in the "Scramble for Africa" in the late 19th century. The extreme lethality was employed to devastating effect against obsolete charging tactics, when African opponents could be lured into pitched battles in open terrain.

Similarly, the open-field battles we know about in the Bronze Age were carried out between chariot-mounted archers, whose horses could easily outrun unarmored massed infantry, dodging their arrows, while the archer picked off the immobile soldiers one by one. Building chariots and maintaining teams of horses involved major expenses beyond what average citizens could muster; consequently they belonged to palaces whose rulers could tax many subjects, and there were no citizens. Iron Age warfare, by contrast, was dominated by mass formations like the phalanx, and maneuvers of infantry like the Greek hoplites, following the “hoplite revolution” around 00700 BCE. This is the era during which (illiberal) democracy flourished in Athens and many other Greek city-states, despite frequent interruptions and setbacks, and (illiberal) republicanism would rule Rome for centuries before the Empire; Livy and Plutarch trace this development to Greek influence. Rousseau argued that it was the “professionalization” of the Roman legions (previously a conscript force) that ended the Republic; he may have been right.

The European Middle Ages were dominated by a knightly class whose power over the serfs was nearly absolute; various “communes” governed by the common people arose from time to time but, except for Switzerland and some Italian republics, were overthrown within a century or three by ambitious kings and knights, notably the Habsburgs. This was possible because the technological basis of warfare at the time enabled a few well-trained warriors with horses and armor to defeat a much larger number of peasants, so democracy, liberal or not, could not withstand contact with warriors — whether external or internal.

Guns, especially smoothbore guns like the arquebus and musket, require much less training to use effectively, enabling large groups of peasants to defeat capital-intensive, highly-trained knights, for example in the Hussite revolt in 01419. In the Early Modern Period, gunpowder thus gave the military advantage to whichever faction was more popular, and Renaissance republicanism took form, with Machiavelli strongly advocating universal conscription. The French National Assembly put Machiavelli's advice into practice with the *levée en masse*, allowing Napoleon to defeat the better-trained Prussian army (accustomed to the cabinet wars that started in 01648) by outnumbering them ten to one — and of course the result of this new form of power was bloodthirsty conquest.

This military dynamic was disrupted by the late-19th-century emergence of the crew-served machine-gun, exemplified by the Maxim type mentioned above, then to some extent restored by the emergence of Kalashnikov's AK-47. Peasant forces armed with AK-47s and similar weapons were able to resist invasions by vastly better equipped armies in some cases in the 20th and early 21st century, notably the US's defeat in Vietnam, the USSR's defeat in Afghanistan, and the US's defeat in Afghanistan in 02021 after 20 years of fighting.

Where do we stand today, and where are we going?

Still, it's hard to credit that a military élite would be unable to defeat a much larger untrained force in a world full of nuclear weapons, aircraft carriers, near-universal gun control laws (extending even to knives in a few cases, such as the UK since 01953), fighter jets, and pervasive surveillance. Nowadays surveillance enlists cellphone networks, social networking sites, license-plate cameras, RFID toll-road payment schemes, DNA databases, credit cards, and sometimes more extreme measures; to assassinate Osama bin Laden, the US subverted a vaccination program to scan DNA, and in putting down the rebellion in occupied Iraq in 02007, the US used high-resolution telescopes like ARGUS-IS to record all public movement in an area, so as to be able to trace back roadside bombs to the houses of rebels after they exploded.

The US's moderately successful invasion of Iraq in 01991 demonstrated the effectiveness of precision-guided munitions, which have been pivotal weapons in every war since then; in 02014, Putin gave a speech at Sochi (new location), saying they were as important as nuclear weapons:

> Many states do not see any other ways of ensuring their sovereignty but to obtain their own bombs. This is extremely dangerous. We insist on continuing talks; we are not only in favor of talks, but insist on continuing talks to reduce nuclear arsenals. The less nuclear weapons we have in the world, the better. And we are ready for the most serious, concrete discussions on nuclear disarmament – but only serious discussions without any double standards.
>
> What do I mean? Today, many types of high-precision weaponry are already close to mass-destruction weapons in terms of their capabilities, and in the event of full renunciation of nuclear weapons or radical reduction of nuclear potential, nations that are leaders in creating and producing high-precision systems will have a clear military advantage. Strategic parity will be disrupted, and this is likely to bring destabilization. The use of a so-called first global pre-emptive strike may

become tempting. In short, the risks do not decrease, but intensify.

Precision-guided munitions can be immensely more efficient than conventional munitions; bombing Dresden into a firestorm took 7100 tonnes of bombs, and the atomic bomb that devastated Hiroshima weighed 4.4 tonnes, but it only takes 100 milligrams of explosives, 200 nanograms of botox, 50 nanograms of polonium, or 100 millimeters of knife blade to kill a general or president on the opposing side, or — perhaps more effectively — his son or daughter, if he doesn't cooperate. There are two tricky parts to this sort of assassination: knowing where the target is, and delivering the weapon to the target. A precision-guided munition solves the second part of the problem, but presently the first part remains the province of centralized spy agencies like the КГБ where Putin grew up.

The wide availability of PGMs, then, which Raskin argued in 02001 was inevitable, would seem to deliver a strong military advantage to whichever group is best at spying on the weak points of its rivals — and concealing their own. The organizations that will be best at this will probably be criminal gangs like Mexico's Zetas.

There is of course no guarantee that such conditions will permit the continuance of civilization, much less democracy; civilization has collapsed on many occasions in the past (around the Mediterranean in the Bronze Age Collapse, in the Maya collapse, in Egypt under the Christian onslaught in the 5th century, in the collapse of the Anasazi, to a lesser extent the post-USSR age, and so on) and it may do so again. Technological conditions that permit small military élites to militarily defeat much larger popular movements may doom democracy, since it deprives those élites of reasons to accede to electoral defeat; conditions that make attacking easy and defending impossible would seem to doom civilization entirely.

In March we saw ransomware shut down the main oil pipeline on the East Coast of the US, and on December 7 there were major outages of much of the World-Wide Web when the biggest data center of Amazon Web Services went down; many customers were revealed to have no disaster plan. There is no evidence that this incident was due to any kind of hostile action, just that AWS is a central point of vulnerability a malicious actor could attack, possibly commanding a high ransom.

Topics

• History (p. 1153) (24 notes)
• Politics (p. 1279) (3 notes)

Xerogel compacting

Kragen Javier Sitaker, 02021-12-22 (updated 02021-12-30)
(12 minutes)

As explained in Material observations (p. 633), Elmer's Glitter Glue changes in an interesting way when it dries: as deposited, the glitter flakes are dispersed almost isotropically, without any preferred orientation, but drying makes the glitter flakes mostly parallel to the surface.

Basic high-laminar-filler composites

This suggests that by this means we can fabricate composite materials with high loadings of fibrous or especially laminar reinforcing filler (e.g., talc, clays, carbon fibers, carborundum whiskers, graphene, MXenes; see Electrolytic 2-D cutting and related electrolytic digital fabrication processes (p. 1085) for some fuller lists) without having to assemble them layer by layer. Specifically, you bulk up the polymer with a lot of solvent, mix in your functional filler, deposit it in an X–Y sheet of uniform thickness on some substrate, evaporate off the solvent resulting in a great loss of volume, and then break the sheet free from the substrate.

Adherence to the substrate prevents the sheet from shrinking in the X–Y plane as it dries, forcing all of the volume loss to be in the Z direction; this reorients the filler particles nearly parallel to the X–Y plane, allowing them to pack much more tightly than laminar reinforcing particles normally could, potentially resulting in a final product that consists almost entirely of the filler, bound together with a small amount of matrix. If the solution is sufficiently viscous or dense, or if surface charges maintain the filler particles deflocculated, the filler will not settle out or float out before being held in place during the drying process. Thus the filler will be evenly dispersed through the resulting matrix.

Laminar reinforcing fillers are extremely desirable as reinforcement because, while each particle of a fibrous filler adds strength in one dimension, laminar fillers can add the same strength in two dimensions. So, in theory, where most of the composite's strength comes from the filler and not the matrix, laminar reinforcement should be able to make materials that are twice as strong. Normally, though, they can only be added as very low percentages, because they either run into one another at weird angles so that more filler can't enter, or they clump up in stacks, so their large surface area doesn't touch the surrounding matrix, so their strength doesn't get transferred.

Sheets, whiskers, and other fibers below a critical dimension are "flaw-insensitive": they're small enough that most of their length or width lacks crystalline defects, so there is no flaw at which to concentrate stress. This commonly increases their strength by an order of magnitude or more. This effect does not come into play with craft glitter.

Waterglass-matrix composites

Waterglass is another interesting possible candidate here, since it forms a soluble mostly-silica xerogel when it dries, although it may be brittle enough at room temperature that it won't achieve super strength. Mixing waterglass with clay to repair things has a long tradition in pottery, and it deflocculates the clay due to its alkali-metal ions.

(It is possible to further harden waterglass after it dries by exchanging its alkali ions with polyvalent cations, as is done in KEIM paint.)

Non-evaporated composites

Alternatively, rather than evaporating the solvent, you could try removing it by some other means. For example, you could press the mass against a semiporous membrane so as to reverse-osmotically force the solvent through the membrane, but not the dissolved matrix or the filler, similar to using frit compression to produce buckypaper (which typically has no binding matrix) or slipcasting pottery. I'm pretty sure this will work in a closely analogous way; the situation is very closely analogous to the solvent case.

Maybe it would even work to squeeze a molten matrix material through a porous material (such as a sintered frit, unglazed fired clay, or dirt) in this way, leaving only the oriented filler and a small remnant of binder, which would remain when the material was cooled; in some cases, as with slipcasting, the capillary action in the porous material would itself be enough to suck away the excess binder.

The things I'm not sure about is ① whether the currents of molten binder will tumble the filler particles as they pass (I think not; I think they'll just press the filler particles up against the porous wall, and at any rate you can do the squeezing more slowly to get slower currents) and ② whether you can separate the composite from the frit afterwards (but in the worst case you can cut it off parallel to the frit surface while it's still almost molten).

Metal evaporation

As I understand it, the drying process normally works by first gelling the viscous solution into a hydrogel, then contracting it into a xerogel under the influence of surface tension in the nanopores of the gel. Thus, carrying out the above process with metals using mercury as a solvent may or may not work, because solid amalgams are not gels, and the shrinking process may be different from xerogel collapse. It may work anyway, though; mercury can dissolve all of zinc, copper, tin, lead, silver, and gold to an appreciable extent, and at least in the case of gold it is commonplace to recover fully dense solid gold by heating, which is how mercury gilding works.

The IUPAC solubility series volume 25 (*Metals in Mercury*) has the following solubilities for some selected metals in mercury at a couple of temperatures:

Metal	Room temperature	300°
Magnesium	2.52%	26%
Aluminum	0.014%	5.6%
Tin	1.05%	>84% (tin melts at 231°; miscible?)

Lead 1.47% 93% (lead melts at 327°)

Titanium 0.000017% 0.0035%

Chromium ???? too low to measure

Iron ???? <0.00004%

Cobalt ???? <0.00007%

Nickel ???? 0.007%

Copper 0.0092% 0.6%

Silver 0.065% 5.1%

Gold 0.13% 14%

Zinc 6.32% 70%

Magnesium is the most tempting entry here, but I'm guessing that if you were going to dissolve magnesium in mercury and then evaporate off the mercury, you'd have to do it in a way well protected from oxygen. Aluminum amalgams aggressively extrude fibers of aluminum oxide over the course of hours when in contact with air.

Aluminum, zinc, and tin are also soluble to a useful extent; you could dissolve a significant amount of zamak 3 (96% zinc, 4% aluminum) in hot mercury.

Rather than using unfashionable and costly mercury, it might be better to try to dissolve other metals in affordable and nontoxic magnesium or zinc, and then use an elevated temperature to vaporize the magnesium (boiling point: 1091°) or zinc (boiling point: 907°) from the alloy.

Water's vapor pressure at 25° is about 3.2 kPa, 24 mmHg, at which rate it evaporates fast enough to be useful. Zinc melts at 419.5°, and its vapor pressure is well approximated by $\log_{10} P_{mm} = -7198/T + 9.664$ (McKinley & Vance 01954), where T is in K, so it reaches that pressure at 869 K = 596°. As metal-fabrication processes go, that's a pretty moderate temperature, which is why zinc fumes pose such a risk of metal fume fever. You might want to evaporate off the zinc in vacuum or under argon or nitrogen. (At 600° you have to use ammonia to get zinc nitride, so just nitrogen is adequately inert for this.)

Unfortunately, there's no IUPAC solubility series volume on the solubility of various metals in molten zinc, but there are lots of phase diagrams for zinc alloys. In particular, molten zinc can dissolve about 10% Cu at 596°, and eutectics and near-eutectics used in soldering include $Sn_{91}Zn_9$ (KappAloy9) 199°, $Zn_{95}Al_5$ 382°, and $Cd_{82.5}Zn_{17.5}$ 265°, so molten zinc is evidently capable of dissolving substantial amounts of aluminum even at much lower temperatures. Below 596° no other structurally useful metals melt, but metals that melt at lower temperatures than copper include gold, silver, and of course lead, tin, and magnesium. So we might reasonably guess that, like copper, substantial amounts of those metals can dissolve in molten zinc at 596°; a paper suggests it can handle 20 mol% of silver. And in particular you ought to be able to dissolve *bronze* in zinc at that temperature, then evaporate off the zinc, or most of it.

More excitingly, a calculated phase diagram suggests that zinc should be able to dissolve about 3 mol% nickel at 600° and about 25 mol% at 873°, and another suggests 2 mol% iron at 600°.

Nanolaminating to get flaw-insensitive laminar fillers

Typically the critical dimension for flaw-insensitivity is a few tens of nanometers, which is an entirely practical thickness at which to electroplate. It occurs to me that if you want a lot of high-aspect-ratio sheets, you could make them out of a metal in the following way. You start by plating a nanolaminate consisting of alternating layers of your desired metal and some other metal or material that is easily etched later, using an etchant that will spare your desired metal; you might deposit, for example, 20 nm of each metal. Then you pulverize the nanolaminate (perhaps easiest if you initially plated it onto a metal where it had terrible adhesion, or onto a layer of graphite), for example by ball milling, into particles of, say, 1 μm. Then you etch these particles with the etchant and separate the resulting metal sheets, which are 1 μm × 1 μm × 20 nm in the example I've given.

If the adhesion between the layers of the nanolaminate were sufficiently poor, maybe you wouldn't even need the etching step.

These high-aspect-ratio flaw-insensitive metal particles are suitable for use as a functional filler to make an ultrastrong composite material, whether the binder is an organic polymer, a geopolymer, waterglass, another metal, or something else.

Some pairs of metals cannot be plated from the same bath; in that case you have to move the forming nanolaminate back and forth between two baths, rinsing it in between. In other cases, you can make a bath which will plate only one metal at one voltage and a mixture of two metals at a different voltage. In other cases (chromium and titanium being notable here) you can grow an anodic oxide layer by reversing the voltage; this may be sufficiently thick to etch later but sufficiently thin to permit plating metal on top of it.

An alternative to moving back and forth between baths is to consume all the platable metal in one bath, leaving only, say, alkali metals; then you can inject the new metal directly into the bath. Indeed, you may be able to "inject" the new metal simply by turning off the inert cathode and switching to a cathode that will dissolve, or increasing the voltage on the cathode. By using a thin electrolyte (say, 1 mm) and cathodes even more closely intercalated (say, 0.1 mm, perhaps foils of two metals stacked alternatingly with dielectric sheets between them, like a multilayer capacitor) you may be able to switch back and forth more rapidly between metals than with a rinse tank.

Another possible alternative separator is to deposit not an anodic oxide film but the insoluble hydroxide of a metal in solution, such as magnesium, which will deposit on the cathode, just as metals do (see Fast electrolytic mineral accretion (seacrete) for digital fabrication? (p. 779)). Magnesium hydroxide in particular is easy to remove with many acids.

Topics

• Materials (p. 1138) (59 notes)

- Filled systems (p. 1161) (16 notes)
- Strength of materials (p. 1164) (13 notes)
- Waterglass (p. 1189) (8 notes)
- Anisotropic fillers (p. 1218) (6 notes)
- Solubility (p. 1273) (3 notes)

Photoemissive power

Kragen Javier Sitaker, 02021-12-23 (updated 02021-12-28)
(15 minutes)

On Earth we make our photovoltaic panels out of semiconductors, separating the positive and negative charge collection nets with the depletion region of a reverse-biased pn semiconductor junction, but in space we could use photoemission across a vacuum gap; this will probably give less power per unit area but more power per unit mass than silicon solar cells, but will be thoroughly dominated by thin-film cells.

I got this idea from a discussion with Luke Parrish, who suggested that for space-based PV panels you could just use vacuum, and contributed several other key ideas to what follows.

Basic design

We were discussing aerographite, which has recently been mooted as a possible solar sail material with 1 kPa UTS and 180 g/m³. As it happens, aerographite is fairly conductive (0.2 S/m at that density).

You could make a large photoemissive solar panel, like old vacuum-tube electric eyes but backward-biased. WP claims that cesium on a silver oxide support gives photoemissivity down into the infrared, so you could plate such a mixture on the sunward side of the aerographite support to make a photoemissive cathode, potentially a gigantic one; photon energy beyond what is needed to overcome the work function becomes electron kinetic energy, which can push the electron uphill against a potential difference to an electron collector grid anode, which needs to be porous, to let light through, and spaced far enough away from the cathode with insulating supports to prevent field emission from stealing the electrons back to the cathode. The spacing can be any distance that is small relative to the mean free path in the vacuum medium.

The anode grid

The spaces in the electron collector grid through which light comes will also permit the loss of some photoelectrons, perhaps the majority. Assuming no charge transfer to the solar wind, the lost electrons will eventually fall back to the positively-charged PV panel, some striking the cathode and others the anode. If it's desired to maximize efficiency per area rather than efficiency per mass, you can extend the grid sunward into a honeycomb which lets almost all of the light through, while capturing all of the electrons, except for those emitted at a very small angle to the incoming light. However, extending the grid "vertically" in this way runs into diminishing returns very quickly; to maximize the electrons captured per unit mass of panel, the thickness should be only a little thicker than the width of the "wires" in the grid in the "horizontal" direction.

This means that the mesh of the grid should be as fine as possible, but its holes still need to be large relative to the wavelength of light and relative to the thickness of its "wires". I suspect that hole

diameter on the order of 1–10 μm will be optimal, with "wires" on the order of 0.1–1 μm "horizontally" and 0.1–3 μm "vertically". An omnitriangulated mesh would be optimal for rigidity; a hexagonal mesh would be optimal for compliance and wire-to-hole ratio; a square mesh is in between these extremes.

This works out to be on the order of 1 g/m² for the mesh if it is made of something like aluminum (2 · 0.3 μm · 1 μm · 3 μm / (3 μm)² · (3 g/cc) = 0.6 g/m²), corresponding to the areal density of a uniform sheet of about 100 nm. These dimensions are too small to make use of the lower density of aerographite itself, because those result from heterogeneity at larger scales than that.

The cathode structure

If the cathode has 50 nm of low-work-function photoemissive material plated onto the front of it, which I think is realistic, backed by the low-density aerographite mentioned above with an areal density of ½ g/m², it would be about 2.8 mm in average thickness. You would of course want to give both this cathode and the anode thicker and thinner parts, like the veins of a dicotyledon leaf or the threads of ripstop nylon, to reduce their electrical resistance and mechanical compliance.

It may be important to keep the electrodes cool to avoid loss of the electrodes from the anode mesh. Using a high-work-function surface for the anode and the non-sunward side of the cathode may be helpful to reduce such losses. Also, if you're using volatile metals like cesium, you need to keep the electrode cool or it will evaporate off into space.

Areal density: 2 g/m²

This adds up to an areal density for our orbiting solar panels on the order of 2 g/m² or 2 tonnes per km², roughly a hundred times lighter than conventional silicon solar panels at 100 μm thickness; typically in space multijunction cells with efficiency around 30% are used.

Calculating efficiency

A km² of sunlight is about 1400 megawatts at Earth's orbital distance, or much more if you're closer to the sun, but how much of that can we really gather?

This depends on the quantum efficiency of photoemission from the cathode (what fraction of photons eject an electron) and the reverse bias voltage we demand the electrons fight against. Photons whose energy is precisely the work function plus the bias voltage are converted with 100% efficiency; photons at any lower energy are entirely wasted; any excess photon energy over that minimum is wasted. (We could imagine multiple cleverly shaped anodes whose electric fields guide most electrons to the highest-energy anode they'd be able to reach, but let's assume we don't do that.)

Bias voltage limits spectral efficiency to 33%

So, too high a bias voltage will produce zero current, but lowering the bias voltage will eventually produce insufficient additional current to make up for the energy loss per electron; there's some voltage at

which we see the maximum power. (This closely corresponds to spectrum losses in conventional PV panels.) This MPPT bias voltage will be a little lower than the energy of the average photon, which is probably about 700 nm; $hc/700\text{ nm} = 1.2398\text{ eV}/0.7 \approx 1.8\text{ eV}$, so probably the right bias voltage is on the order of 1.8 V, which is a conveniently tractable voltage; the Shockley-Queisser limit is an efficiency of 33% at a semiconductor bandgap of 1.34 eV, which I think corresponds to a bias voltage of 1.34 V in this photoemissive panel.

(Note: the above is incorrect, and energy efficiency calculations hereafter erroneously assume that the bias voltage between the electrodes is 1.8 V, which is wrong. 1.34 V or 1.8 V is the amount of energy per electron lost in overcoming the work function of the photocathode material; the energy remaining to be harvested at the anode is whatever the photon energy is, *minus* that work function. So the right bias voltage might be 0.5 V or 1 V or something. I should fix this but I don't have time this year. It means that the main efficiency conclusions below are too high by some unknown factor probably between 1 and 4.)

I think the recombination losses found in semiconductor PV cells do not have much of an analogue in this device; the space charge is entirely negative, and the only way electrons can "recombine" after leaving the cathode is to fall back onto it, either because they lacked the energy to reach the anode or because they went through holes in the anode twice. Presumably there is at least some probability that they will be "emitted" into the cathode material, though, where they will immediately "recombine".

Quantum efficiency can be around 15%

So, what about the quantum efficiency? Evidently in silicon PV it's around 0.8, but these photoemissive panels might be much worse. If their quantum efficiency were, say, 10^{-6}, they would produce less electrical energy per mass than conventional silicon cells, rendering them useless. Wikipedia says that phototubes typically produce microamperes, and they typically have a cathode area around 10 cm² and are typically illuminated by an infrared beam that can't be much more than 10 W/m² (or we'd feel it on our skin and possibly damage our eyes), which puts a lower bound on their QE of about $10\text{ cm}^2 \cdot 10\text{ W/m}^2 / 1.7\text{ eV} / (\mu\text{A}/e) \approx 1/6000$. At this QE we would expect $33\% \cdot 1400\text{ W/m}^2 / 6000 \approx 77\text{ mW/m}^2$, which is high enough to be useful but not high enough to compete with conventional solar cells; dividing by the estimate above of 2 g/m², we get 39 mW/g, which is much lower than the areal efficiency of conventional multijunction silicon solar cells, $30\% \cdot 1400\text{ W} / \text{m}^2 / (230\text{ g/m}^2) \approx 1800\text{ mW/g}$, 46 times higher.

So this approach can be mass-competitive with multijunction silicon solar cells if the photoemissive cathode quantum efficiency is more than about 1/130, i.e., 0.8%.

In fact the cesium-antimony photocathodes used in the first commercially successful photomultiplier tubes have a quantum efficiency of 12% at 400 nm, though the quantum efficiency of earlier silver-oxide-cesium photocathodes peaked at 0.4% at 800 nm. This information seems to come from p. 4 of the Photomultiplier

Handbook; on p. 11 it says, "on the best sensitized commercial photosurfaces, the maximum yield reported is as high as one electron for three light quanta," which would work out to 33% QE. This would give an overall solar cell efficiency of $33\% \cdot 33\% = 11\%$, but that's probably for a single wavelength; a few of the QEs of different materials plotted on p. 15 are above 10% at 555 nm, and some, like Na_2KSb, are above 20% at 450 nm, so maybe $33\% \cdot 15\% \approx 5\%$ is more realistic. In Table I on p. 16, Na_2KSb's responsivity to tungsten light at 2856 K is given as 43 μA/lumen, while K_2CsSb (nominally 33% QE) is given as 90 μA/lumen. Nominally lower QE materials with longer-wavelength peaks are even higher: GaAs:Cs-O is said to have 720 μA/lumen despite only a 12% QE due to an 800-nm response peak, and semitransparent Na_2KSb:Cs on a reflecting substrate is 300 μA/lumen with 16% QE with a 530-nm response peak, which matches sunlight better than it does a tungsten lightbulb. Presumably these are all in a forward-biased condition, as they are used in PMTs, not back-biased, but hopefully the correction is small.

Rechecking the calculation from a different angle, 1000 W/m² is about 128000 lux, so the above-the-atmosphere 1400 W/m² should be about 180 klux = 180 klm/m², which at 300 μA/lm would be 54 A/m²; at 1.8 V that would be 97 W/m², which is 6.9% efficiency, close to the 6% I estimated above.

So it seems likely that, using new ultralight electrode materials like aerographite, coated with modern (semiconducting?) multialkali photocathode materials, this photoemissive generator can probably beat silicon PV in power per unit mass by a factor of, say, 20 or so (50 W/g instead of 1.8 W/g), but it will be five times worse in power per unit area (6% efficiency rather than 30%).

Thin-film semiconductor PV cells like CIGS can probably beat it in power per unit mass, too.

Moreover, the Photomultiplier Handbook says, "Semiconductors, therefore, are superior to metals in all three steps of the photoemissive process: they absorb a much higher fraction of the incident light, photoelectrons can escape from a greater distance from the vacuum interface, and the threshold wavelengths can be made longer than those of a metal. Thus, it is not surprising that all photoemitters of practical importance are semiconducting materials." So in a sense this gadget *is* a semiconductor thin film solar cell.

10.1088/1361-648X/aa79bd "Super low work function of alkali-metal-adsorbed transition metal dichalcogenides" claims work functions as low as 0.7 V with a potassium film on a strained tungsten telluride backing.

Interestingly, the "semitransparent" photocathode materials are "deposited on a transparent medium," with typical film thicknesses around 30 nm, so as to emit electrons in the opposite direction from the incident light. That suggests the possibility of reversing the positions of the cathode and anode and making the anode opaque, so there is no question of electrons escaping through holes in it. Conceivably supporting the photocathode thin film in a vacuum on a sparse grid like the anode grid described earlier, covering what would be holes in the grid, would get photoelectrons coming out both sides, so that by placing anodes on both sides you could increase the

quantum efficiency, perhaps doubling it. That might boost you to 14% efficiency or so, but still not enough to compete with existing CIGS and similar solid-state thin-film PV cells.

Cathode meshes

Most of the mass of the cathode in the above setup comes from the thin-film cathode (and then I just calculated on the assumption that the anode mesh would have comparable mass). An interesting way to reduce the mass further is to use a photocathode *mesh* or foam rather than a solid layer. A mesh with holes significantly smaller than the wavelength of light can be essentially opaque to the light if it's sufficiently conductive, so you could use a photocathode mesh with 100-nm-wide pores separated by 1-nm-wide "wires", thus reducing the necessary areal density of the cathode by 98%.

Existing systems

Parrish commented that existing systems are about an order of magnitude heavier than the number I was using above as a silicon-solar-cell comparison:

> The ISS uses 8 solar array wings massing about 1 ton each that get 84–120kW average or up to 240 in direct sunlight. So about 30W/kg in direct sunlight. We're talking 3 orders of magnitude improvement.

Apparently photoelectric solar power is a thing, and I should read about how well it works, but I don't have time this year.

Topics

- Physics (p. 1157) (18 notes)
- Energy (p. 1170) (12 notes)
- Solar (p. 1203) (6 notes)
- The future (p. 1220) (5 notes)
- Space (p. 1323) (2 notes)
- Photoemission (p. 1341) (2 notes)

Toggling eccentrics for removing preload from spring clamps

Kragen Javier Sitaker, 02021-12-28 (updated 02021-12-31)
(22 minutes)

Watched an Abom79 video I'd seen before tonight which featured a parting blade with replaceable inserts. A parting blade doesn't have a lot of space for holding down an insert; you can't put a big screw in there, for example. So the insert was wedged into a sort of two-tine fork, which flexes elastically to admit it.

The parting-blade setup

This means the force clamping the insert into the blade, perpendicular to the friction surface, is the same force you need to apply to force the tines apart to insert or remove the insert. To supply this force, there are round holes in the two tines, and an opening tool is supplied, consisting of a spacer between two parallel dowel pins that fit into the holes. One of the dowel pins is machined on an eccentric with a handle to rotate it relative to the spacer, moving it nearer to and farther from the other dowel pin, thus forcing the tines open. It's sort of similar to circlip pliers in how it engages with the fork, and in the fact that it consists of three revolute joints on parallel axes (one between the two links of the tool, the other two connecting the tool to the workpiece), but it has much greater mechanical advantage.

Why this is awesome

This is a really appealing concept for a couple of reasons.

First, if we disregard friction and the compliance of the tool, the mechanical advantage is unlimited; the ratio between the length of the handle and the center-to-center distance of the eccentric provides the mechanical advantage, and the center-to-center distance (eccentric axis offset distance) can be any value down to zero. With an ordinary lever, to get a very short lever arm, you also have to make the lever arm thin, which limits the load it can take; no such limit exists with this eccentric lever, because the eccentric pin can be as large as it needs to be to resist the shear load (and, in the case where the hole is deep, bending loads).

Friction is still an issue: the surface of the dowel pin rotating half a turn in the hole creates a frictional moment opposing the action of the tool, whose lever arm is the radius of the pin, which must grow according to the square root of the fork-opening force in order for the pin not to shear off. So you have an opposing moment of $F^{3/2}$, which in the usual case will be about as big as the work you're actually doing, since the radius of the pin is typically a bit larger than the eccentric axis offset distance, and the coefficient of friction is typically a few times smaller than 1. Friction inside the tool may or may not be reduced with ball bearings or similar, but if not, it's an additional comparable loss. Such losses may be desirable in this context to prevent back-drivability, but they limit its applicability.

Second, when the fork opens and closes, assuming parallel planar clamping surfaces, it doesn't have any tendency to screw with the position of the insert it's clamping down on, except in the direction of clamping and the two directions of rotation whose axes aren't parallel to the direction of clamping. So it restrains the insert in six degrees of freedom, but purely with friction in three of those degrees, permitting any position in those three degrees. This is a nice improvement over things like jam nuts, which have a tendency to put slightly askew the position you were intending to hold steady.

Third, the mechanical advantage becomes infinite as the eccentric rotates to the position where the dowel pins are farthest apart, in the usual toggle-mechanism way, because the lever arm becomes zero.

We can ask, what is the maximum clamping force we can apply with this mechanism? There's no inherent limit coming from the applied force (we can make the lever you rotate arbitrarily long, so given a fixed point, you can move the Earth with an arbitrarily small force) but there might be a limit coming from the energy, once friction is taken into account.

Also, the compliance of the thing you are clamping may itself provide a minimal energy cost: if under 10kN it elastically compresses 0.1 mm, you need to open the jaws by more than 0.1 mm in order to insert it in its uncompressed state, applying something more than 10kN in the process, probably only a little bit more if the spring clamp (which you are expanding) is more compliant than the thing being clamped. So that would require about a joule, plus frictional losses.

(Many cheap digital fabrication processes have imprecision on the order of 0.1 mm: laser-cutting MDF, laser-cutting acrylic, laser-cutting steel, CNC plasma tables, CNC oxy cutting tables, RepRap FDM, etc. You need to be able to flex the spring by more than the fabrication error or in some cases you'll get no clamping and in other cases you won't be able to insert the thing to be clamped.)

The compliance of the elastic part of the setup (the workpiece, in this case the clamp) can be reduced almost arbitrarily, as long as the compliance of the tool is smaller or at least not too much greater. Given a nominal Young's modulus of 200 GPa for steel, and considering a 10 mm distance between the points where force is being applied, we can get a compliance of 5 microns per newton stretching a 100-micron-square steel wire, or 50 nm/N stretching (or compressing) a 1-mm-square steel rod, or 500 pm/N stretching (or compressing) something that averages out to a 10 mm block of steel. At this last compliance, 10kN would result in a 5 micron elongation, which is still plenty to switch between contact and non-contact regimes for things like electrical conductivity and so might be enough to clamp and unclamp something. As with brakes and clutches, the smaller the compliance, the smaller the energy that is needed to reach a given clamping force and thus a given stiction force. In this example, reaching 10kN of clamping force would only require 50 mJ plus frictional losses.

(Of course in the geometry described earlier the tension path between the prongs was not straight, increasing compliance, but there are rivet-like clamping geometries where the tension path *is* straight.)

Larger compliance may be desirable for resistance to shock loads: if

the energy barrier to unclamping is 100 joules, shocks are much less likely to result in slippage than if it is 0.05 joules.

In cases where the elastic piece doesn't have to be planar, the tool can be simplified to a single rigid body consisting of a handle perpendicular to a shaft consisting of two cylindrical sections with parallel axes. The shaft is inserted through a slot in the elastic piece into a round hole in another part of the elastic piece, and rotating the shaft with the handle then moves the round hole perpendicular to the length of the slot and to the shaft's axis of rotation. This could reasonably be used for things like fasteners, though in the case of one tool to operate many fasteners, it might be cheaper to make the fasteners as simple as possible (like circlips) and put any extra complexity in the tool.

To reduce frictional losses, the actual holes can be mounted in rotating flexures, or the eccentric pin in the tool can be held in a pair of bearings to reduce friction. These bearings are mounted eccentrically inside a larger shaft mounted on its own bearings which is rotated by the handle. It's possible but maybe not practical to avoid the use of large bearings in this case: the larger shaft can neck down to fit into two small bearings at the non-workpiece end, which are spaced far enough apart to handle the resulting moment.

Another way to reduce frictional losses is to provide some leverage within the flexing workpiece itself. In the case of a fork, you might clamp the workpiece half as far from the bifurcation as the distance from the bifurcation to the holes for the tool. Then the tool only has to apply half as much force as is applied to the thing being clamped, and the tool rotation experiences about half as much friction.

How big *are* those frictional losses?

Suppose that we are applying, again, 10kN of force with the tool, over a whole half turn, using a handle which cannot be more than 1 m long (the example in the video was about 150 mm long) and to which we can only apply 200 N of force. Perhaps the eccentric pin is made of a bearing bronze such as SAE 660 tin bronze. Its yield tensile strength is given as 125 MPa; its shear strength might be 0.6 of that, 75 MPa. To fail under 10kN of load, then, it needs to be 13 mm in diameter, giving a cross-sectional area of 132 mm^2; 20 mm diameter would give a good safety factor. The eccentric axis distance could be as large as 20 mm, or actually even greater since when the lever arm is at its largest, the clip isn't fully extended yet, so the force is not at its largest. If it's 20 mm, we can use it to deform the clip by 40 mm, for a total useful work of 200 J.

That bronze is rated as having a frictional coefficient of 0.10, presumably on steel (though in 02001 Purcek et al. measured 0.68 when dry, so maybe 0.10 is with an oil film), so the pin rotating in the workpiece hole ramps up to 1 kN of friction. Half a turn is 62 mm, so we have 31 J of frictional losses rotating in the workpiece, and probably another 31 J of frictional losses where the shaft rotates inside the tool, for a total of 62 J losses, 76% efficiency. Ball bearings or similar could reduce these losses by about an order of magnitude, so they are more like 2% instead of 24%.

Note that this means you have to apply more force to the handle

than the 200 N I was calculating with. More like 262 N. Except that the force is going up as the lever arm goes down; at 45 degrees you have 70.7% of the force, 7 kN, and also 70.7% of the lever arm, 14 mm, so only half that 200 N, 100 N, plus 70.7% of the frictional force, another 44 N at the handle, for a total of 144 N. A little lower, at 30 degrees, you have half the force, 5 kN, and 87% of the lever arm, so 43% of the 200 N plus 50% of the frictional force. I should plot this I guess.

This bronze is also rated as having 315 MPa compressive strength, so the hole in which that pin is turning would need to be at least 1.6 mm deep to avoid damaging the surface, maybe more like 3 mm deep.

This is a fairly terrifying tool configuration, though, sort of like garage-door-spring winding but at lower energy and much higher force. If you lose your grip on the handle, it is going to acquire 138 J of kinetic energy.

Suppose we are stretching a less compliant workpiece, so we only need to deform it 4 mm. If we use the same tool, we get about the same efficiency (a little better, actually, since much of the frictional loss is in parts of the circle that do very little real work) but we could alternatively redesign the tool to have an eccentric axis offset of only, say, 2 mm. This would reduce the moment from the workpiece from 200 N m to only 20 N m, but we still have 20 N m of frictional loads because the pin is still 20 mm in diameter. This enables us to reduce the handle length to 200 mm, to which we apply the same 200 N, but now at only 50% efficiency. This tool is no longer backdrivable by the workpiece's elasticity, since the moment from the workpiece is equal to the moment from friction. (Even the larger configuration wasn't backdrivable when fully toggled, because the frictional forces are at maximum and the lever arm is zero.)

We could maybe reduce the frictional losses further by using a mostly hardened steel pin with just a surface of bronze on it, enabling us to reduce the pin diameter by a factor of 2 or more without losing shear strength. Tool steels normally have (tensile yield) strengths of 1 GPa or higher.

In the case where we start almost "toggled" — in the sense that the eccentric pin is nearly at its furthest distance from the other pin when you insert the tool into the holes — the mechanical advantage is very much greater, being limited only by the compliance of the tool.

Variations

A couple of slight variations on the tool configuration are worth mentioning. Shear and flexural loading on the pins can be eliminated if they are only *half* cylinders, with the pivoting bushing inside of the eccentric pin, which can be longer than the workpiece; this makes the loading on the pins entirely compressive, but requires the "holes" in the workpiece to be two circular notches facing each other, between which the tool is inserted. This permits a much smaller hole radius and thus dramatically reduces frictional moments and thus losses. In this configuration it is advisable for the handle to be roughly perpendicular to the line between the two notches in the workpiece, and to collide with the other pin when rotated just past the toggled

position in order to lock the pins ε less than their furthest distance apart.

Such a tool can be cut out of a thick sheet in a single piece with 2-D cutting processes; it consists of two quasi-rigid parts (the handle and eccentric pin being one part, the other being either the Minkowski sum of a circle and a line, or a rhombus with rounded corners) which meet in a cylindrical sliding contact, held in roughly the right position during insertion by a compliant spring that runs along the handle, but which exerts forces that are insignificant compared to the forces encountered during use.

The sheet needs to be thick to prevent it from twisting out of plane.

This pivoting bushing can be split in two to allow the workpiece (or what it is clamping) to protrude past the notches, reintroducing half the shear loading but not the flexural load.

If such a tool is used to push apart jaws on one side of a flexural pivot, those jaws can pivot to come closer together on the other side, with potentially some additional mechanical advantage; when the tool is removed, they will spring back apart, at which point they can bear on the inside of one or more holes, forming a fastener. (However, you need some way to hold the fastener in place as you're applying the tool, or it will just rotate along with the tool instead of flexing; see below.) Such a fastener can also be fabricated by 2-D cutting, in which case the hole or holes can be just a slot. If the hole is tapered to widen away from the surface, withdrawing the fastener from the hole will require adding energy to the flexural pivot, so vibration will tend to seat the fastener deeper in the hole, similar to flexural clips and very much contrary to the situation with screw fasteners.

A fastener containing a double flexural pivot can be used in the same way to convert the opening of jaws, on the side where the tool is inserted, into compression in the middle of the piece, into the opening of another set of jaws on the opposite side, with potentially another layer of mechanical advantage, permitting clamping with truly enormous forces.

(And, of course, the full range of clip-connector techniques is available for these jaws: they can be smooth, serrated, or hooked, potentially mating with matching features on the part they grasp.)

These pivoting-flexure connectors allow the same tool to be used for a variety of sizes of fastener, because the jaw spacing on the tool side of the fastener need not be the same as the jaw spacing on the clamping or expanding side.

In cases where both locating and friction are desired, because the fastener does not have to rotate, it can have a second tab that slips into a second hole or slot in the workpieces to locate them relative to one another, thus unifying in a single part functions similar to those of a screw and a dowel pin. If this second tab is longer than the flexural parts, it can be inserted before applying force to the tool, thus holding the fastener in place while force is being applied.

If the eccentric pin on the tool is replaced by a concave partial cylindrical bearing surface, in which a convex cylindrical part of the

flexible workpiece can slide to form a revolute joint (or, really, a cylindrical joint) then the tool can be used to compress the workpiece rather than to expand it. The effective lever arm is still the distance between the center of this cylindrical surface and that of the cylindrical bearing surface on which the other part of the tool pivots.

If the fastener includes a parallel-movement flexure, a single tool action can engage many hooks, inserted into many slots, in a single motion. This is probably not useful for clamping as such (the clamping load would be distributed among the slots, and in an unpredictable way unless the fabrication tolerances are much smaller than the parts' compliance), but with hook fasteners it allows you to "stitch" two or more parts together along a whole line in a single action. Such a long fastener allows you the leverage to prevent the fastener from rotating along with the tool just by holding it in your other hand.

Contrast with bolts

You need to rotate the bolt through the nut through some 6 turns against the thread friction, which ought to be negligible but usually is on the order of a tenth of the tightening load. Then you crank down on the bolt to preload it in tension; for a 10-mm-head bolt that torque might be 25 foot-pounds or 30 newton meters, applied over maybe a third of a turn, or about 30 J (would be 60 J but the torque goes up almost linearly as you snug it up). Maybe ⅓ of that energy goes into the elastic clamping energy, 10 J; the other ⅔ is lost in friction, 20 J, on top of the ≈3 N m times six turns you lost in just getting the bolt into the nut, which is another 110 J. So you had to spend 140 J and six turns of the wrench to get 10 J of elastic clamping energy, which might be the same 10 kN we were calculating with above. Six turns of the wrench is a real PITA in a confined space, because you have to slip the wrench on and off of the head between 12 times and 72 times.

And then only stiction stabilizes the connection; enough vibration will loosen it unless you apply loctite or lockwire or something. And manufacturing the threads requires a lathe, taps and dies, or a thread-rolling machine, rather than a simple digital 2-D cutting setup.

Topics

- Contrivances (p. 1143) (45 notes)
- Digital fabrication (p. 1149) (31 notes)
- Physics (p. 1157) (18 notes)
- Mechanical (p. 1159) (17 notes)
- Hand tools (p. 1197) (7 notes)
- 2-D cutting (p. 1201) (7 notes)
- Flexures (p. 1232) (5 notes)
- Spreadtools (p. 1321) (2 notes)

Safe decentralized cloud storage

Kragen Javier Sitaker, 02021-12-30 (10 minutes)

How could you safely store files "in the cloud"? How can a "cloud storage" system work? The idea is that you ("the speaker") have a large file and you would like storage servers ("providers") to be incentivized to store it for you so you can retrieve it in the future. Assume for the time being that the file is private, not public.

(This is basically the problem that MojoNation, MNet, Filecoin, etc., attempt to solve, and it's possible that their solutions are better than what I propose here. IIRC Tahoe didn't try to solve the incentive problem, punting it to social "friendnet" considerations, which ultimately failed in almost all cases.)

Of course, you could just send a hosting service some Mastercard payments or bitcoin every month, as long as the file remains accessible. This has a few limitations:

- A malicious hosting service can see the contents of your private file.
- Or change them, which might be worse.
- Or just delete the file, which might even happen by accident.
- Setting it up is hard to automate.
- It stops working when you die.

You can solve problem #1 by encrypting the file and storing the keys locally, and problem #2 by computing a secure hash of the file which you store locally. If you want to be able to retrieve small pieces of the file, instead of just storing a single secure hash, you need to store the hash of a Merkle tree of the file, as some modern hashes like BLAKE3 do implicitly.

A particularly simple way to build such a Merkle tree is by dividing the file into pieces of, say, 64KiB, and concatenating the hashes of the pieces to form a piece table file. Files up to 64KiB will have a single-entry table (32 bytes with SHA-256), which can be stored directly; larger files would require a piece table to be stored somewhere, such as in the cloud storage system itself. With the given example parameters, a single-piece piece table would handle files up to 128 MiB, while you'd also need to store a piece table for your piece table for files up to 256 GiB, while three levels are needed for files up to 512 TiB, etc.

Solving problem #3 is a little more difficult; it requires redundancy among different hosting providers who don't know about each other, which is potentially expensive because you need to store M copies of the file, which costs M times as much. N-of-M Shamir secret sharing gives you a less expensive way to do this and also solves problem #1, again, as long as there's no way for an attacker to find out which hosting providers they are. Then you need to store a Merkle tree of each of the M shares so they can be verified independently. And you need to periodically test some randomly chosen pieces of each share to ensure it hasn't been lost ("auditing").

So far, as I understand it, this is more or less how modern cloud backup systems like Borg, Restic, kup, rdedup, and perkeep work.

If a malicious provider wants to get paid but not perform the service, they can cheat by the following approach: when asked for piece #42 of a file, retrieve piece #42 from N other providers, compute piece #42 of the original file, and then recompute piece #42 of the share they were asked to store. This is not a problem if they have no way to find out about the other providers, but that would impede extending the scheme to public files, because if a file is public then anyone can find out what its pieces are and retrieve them. A different defense against this attack is to encrypt each share before sending it to the provider; then they cannot recompute their piece without the encryption key. By using public-key cryptography, even clients who can decrypt the piece using the public key will be unable to recompute it, which would require the private key.

Extending the system to public files, however, makes it easier to censor: a would-be censor can repeatedly retrieve a public file while observing the network in various ways to find out who the providers of the file are, then attack those providers, perhaps presenting them with ultimatums to delete the file to end the attack. In a simple example where the providers are directly contacted over IP, they can observe which IP addresses they are talking to and traceroute those addresses, an attack which can be stymied by putting the providers on Tor onion services. But even passive traffic analysis (in the time domain, the frequency domain, a wavelet domain, etc.) is probably sufficient to unmask Tor onion services, particularly in the presence of observable network outages, which will tend to increase the latency of particular providers or even knock them offline. Active traffic analysis, in which streams of artificial traffic are used to artificially alter network latency and loss, is even more powerful. High-latency networks like uucp, Fidonet, and the old cypherpunks remailers, particularly with randomized rendezvous times, are one defense that reduces the information leakage to traffic-analysis attackers; constant-bandwidth mixnets like the old ISDN mix proposal are another.

Sybil problems are also important for #3: if all your providers are in Amazon's us-east-1 data center, that's really only a single provider with uncountable faces, because when that data center goes offline, all the providers disappear at once. I don't know how to really solve that problem; the best attack on it so far seems to be Satoshi's proof of work, and that's turned out not to be as watertight as Satoshi had hoped. (Centralized identity systems like government IDs are often suggested to solve Sybil problems, but those don't help with the problem of many distinct real persons hosting their storage server in us-east-1.) Spreading across *many* providers will tend to diminish it, which requires automation, bringing us to problem #4.

Problem #4 is partly a political problem rather than a technical one: censorship policies, especially policies for liability for distributing forbidden information, tend to disincentivize automated data preservation systems. Similarly, repudiable payment systems like Paypal and Mastercard impose counterparty risk on hosting providers: a customer can revoke payment for service they've already received, a problem solved by Bitcoin and similar cryptocurrencies.

Problem #5 is the trickiest: it's an incentive-design problem. You need to delegate the auditing function to some sort of third party that

will probably survive your death. Such systems are rife with principal–agent problems:

- If the auditor simply has custody of money they disburse, they have an incentive to just take the money and run, particularly if they suspect the principal has died and so no further money will be forthcoming.
- If the auditor can only disburse money to predetermined providers, they have an incentive to collude with one of the providers to falsely report that that provider is okay, but all the others are returning no data or wrong data. Then the auditor and the fraudulent provider can split their winnings.
- If the auditor gets paid for producing auditing reports, they have an incentive to produce auditing reports without doing any auditing, since doing the auditing is, if not very expensive, at least not cost-free.

- If the auditor is itself being cross-checked against other auditors, the various auditors have an incentive to become non-anonymous to each other and collude to behave as a single auditor.
- If the providers can somehow predict or influence the auditor's choice of pieces to audit, perhaps without the auditor's knowledge, they can retain only the pieces that will be audited.

If the auditor can be trusted by the provider as well as the speaker, a system becomes possible where the provider posts bonds that are forfeit if it fails auditing. This way, speakers can choose to use providers who stand to lose more if they fail an audit, perhaps long after the speaker's death.

Two ways of cross-checking auditors are to have multiple auditors performing the same audit, who ought to get the same result, or to include "dummy providers" who will intentionally return no data or bad data for certain nonexistent files. If the auditor cannot determine whether a provider is a real provider or a dummy provider, false auditing results can be detected.

Who audits the auditors? Szabo's "property club" approach suggests using a quorum of auditors who can vote dishonest auditors off the island. Alternatively, for public databases (which are the case we most care about after the speaker's death), micropayments can flow from readers of a database to the auditors and the providers; if a database stops working reliably, the auditors can anticipate that readers will stop paying.

Topics

- Programming (p. 1141) (49 notes)
- Systems architecture (p. 1205) (6 notes)
- Protocols (p. 1206) (6 notes)
- Security (p. 1224) (5 notes)
- Incentives (p. 1230) (5 notes)
- Politics (p. 1279) (3 notes)
- Decentralization (p. 1374) (2 notes)
- Censorship

Notes concerning “Materials”

• Duplicating Durham’s Rock-Hard Putty (p. 79) 02021-01-22 (updated 02021-01-27) (1 minute)
• The use of silver in solar cells (p. 112) 02021-02-02 (updated 02021-09-11) (8 minutes)
• A boiler for submillisecond steam pulses (p. 361) 02021-04-28 (updated 02021-12-30) (10 minutes)
• Thixotropic electrodeposition (p. 372) 02021-05-04 (updated 02021-12-31) (2 minutes)
• Precisely measuring out particulates with a trickler (p. 384) 02021-05-09 (updated 02021-12-30) (17 minutes)
• 3-D printing in carbohydrates (p. 393) 02021-05-16 (updated 02021-12-30) (10 minutes)
• Clay-filled PLA filament for firing to ceramic (p. 396) 02021-05-17 (updated 02021-12-30) (1 minute)
• Multicolor filament (p. 397) 02021-05-17 (updated 02021-12-30) (5 minutes)
• Acicular low binder pastes (p. 399) 02021-05-19 (updated 02021-12-30) (1 minute)
• Selectively curing one-component silicone by injecting water (p. 408) 02021-05-19 (updated 02021-12-30) (2 minutes)
• Metal welding fuel (p. 411) 02021-05-23 (updated 02021-12-30) (6 minutes)
• Ghetto electrical discharge machining (EDM) (p. 423) 02021-05-31 (updated 02021-12-30) (5 minutes)
• Verstickulite (p. 457) 02021-06-23 (updated 02021-07-27) (3 minutes)
• More cements (p. 466) 02021-06-26 (updated 02021-08-15) (5 minutes)
• Can you use stabilized cubic zirconia as an ECM cathode in molten salt? (p. 475) 02021-06-27 (updated 02021-12-30) (3 minutes)
• Electrolytic glass machining (p. 477) 02021-06-28 (updated 02021-12-30) (6 minutes)
• Super ghetto digital fabrication (p. 479) 02021-06-28 (updated 02021-12-30) (33 minutes)
• Glass powder-bed 3-D printing (p. 490) 02021-06-29 (updated 02021-12-30) (20 minutes)
• Sulfur jet metal cutting (p. 504) 02021-06-30 (updated 02021-12-30) (6 minutes)
• Stochastically generated self-amalgamating tape variations for composite fabrication (p. 510) 02021-07-02 (updated 02021-12-30) (26 minutes)
• Spin-coating clay-filled plastics to make composites with high anisotropic filler loadings (p. 521) 02021-07-02 (updated 02021-12-30) (4 minutes)
• ECM for machining nonmetals? (p. 523) 02021-07-05 (updated 02021-07-27) (11 minutes)
• Notes on Richards et al.’s nascent catalytic ROS water treatment process (p. 534) 02021-07-07 (updated 02021-07-27) (14 minutes)
• Electrolytic berlinite (p. 561) 02021-07-12 (updated 02021-12-30)

(7 minutes)
• Making mirabilite and calcite from drywall (p. 564) 02021-07-12 (updated 02021-12-30) (4 minutes)
• Potential local sources and prices of refractory materials (p. 566) 02021-07-14 (updated 02021-09-11) (9 minutes)
• Firing talc (p. 576) 02021-07-14 (updated 02021-12-30) (17 minutes)
• Fiberglass CMCs? (p. 588) 02021-07-15 (updated 02021-07-27) (8 minutes)
• Can you 3-D print Sorel cement by inhibiting setting with X-rays? (p. 592) 02021-07-16 (updated 02021-07-27) (1 minute)
• Tetrahedral expanded metal (p. 593) 02021-07-16 (updated 02021-07-27) (3 minutes)
• Glass foam (p. 595) 02021-07-16 (updated 02021-08-15) (17 minutes)
• Aluminum fuel (p. 603) 02021-07-17 (updated 02021-12-30) (2 minutes)
• Boosters for self-propagating high-temperature synthesis (SHS) (p. 604) 02021-07-17 (updated 02021-12-30) (4 minutes)
• SHS of magnesium phosphate (p. 608) 02021-07-22 (updated 02021-07-27) (3 minutes)
• Back-drivable differential windlass (p. 610) 02021-07-23 (updated 02021-07-27) (15 minutes)
• Synthesizing reactive magnesia? (p. 615) 02021-07-25 (updated 02021-08-15) (4 minutes)
• Material observations (p. 633) 02021-07-29 (updated 02021-12-30) (194 minutes)
• Dipropylene glycol (p. 687) 02021-08-01 (updated 02021-08-15) (2 minutes)
• The ayurvedic “fire mud” of Bhudeb Mookerji and modern castable refractories (p. 688) 02021-08-05 (updated 02021-08-15) (22 minutes)

• Cola flavor (p. 707) 02021-08-10 (updated 02021-08-15) (2 minutes)
• Methane bag (p. 710) 02021-08-10 (updated 02021-08-15) (8 minutes)
• Iodine patterning (p. 713) 02021-08-11 (updated 02021-08-15) (1 minute)
• Heating a shower tank with portable TCES? (p. 714) 02021-08-11 (updated 02021-08-15) (6 minutes)
• Maximizing phosphate density from aqueous reaction (p. 757) 02021-08-21 (updated 02021-12-30) (8 minutes)
• A construction set using SHS (p. 765) 02021-08-24 (updated 02021-09-11) (5 minutes)
• Fast electrolytic mineral accretion (seacrete) for digital fabrication? (p. 779) 02021-09-01 (updated 02021-12-30) (52 minutes)
• Patterning metal surfaces by coating decomposition with lasers or plasma? (p. 795) 02021-09-03 (updated 02021-12-30) (7 minutes)
• Blowing agents (p. 847) 02021-09-29 (updated 02021-12-30) (4 minutes)
• Liquid dielectrics for hand-rolled self-healing capacitors (p. 853) 02021-09-30 (updated 02021-12-30) (3 minutes)
• The sol-gel transition and selective gelling for 3-D printing (p. 858) 02021-10-03 (updated 02021-12-30) (6 minutes)

• An aluminum pencil for marking iron? (p. 1001) 02021-11-06 (updated 02021-12-30) (2 minutes)
• At small scales, electrowinning may be cheaper than smelting (p. 1029) 02021-11-21 (updated 02021-12-30) (25 minutes)
• Some notes on Bhattacharyya's ECM book (p. 1043) 02021-11-25 (updated 02021-12-30) (11 minutes)
• Exotic steel analogues in other metals (p. 1050) 02021-12-01 (updated 02021-12-30) (8 minutes)
• Solid rock on a gossamer skeleton through exponential deposition (p. 1076) 02021-12-15 (updated 02021-12-30) (11 minutes)
• 3-D printing in poly(vinyl alcohol) (p. 1080) 02021-12-15 (updated 02021-12-30) (2 minutes)
• Ghetto electrochromic displays for ultra-low-power computing? (p. 1082) 02021-12-16 (updated 02021-12-30) (9 minutes)
• Aluminum refining (p. 1106) 02021-12-20 (updated 02021-12-30) (3 minutes)
• Xerogel compacting (p. 1119) 02021-12-22 (updated 02021-12-30) (12 minutes)

Notes concerning “Programming”

• Principled APL redux (p. 22) 02021-01-03 (updated 02021-12-31) (12 minutes)
• First class locations (p. 27) 02021-01-04 (3 minutes)
• Layout typescript (p. 29) 02021-01-04 (5 minutes)
• Relayout with heaps (p. 32) 02021-01-10 (updated 02021-01-15) (6 minutes)
• Transactional editor (p. 35) 02021-01-14 (updated 02021-01-15) (73 minutes)
• Trie PEGs (p. 53) 02021-01-15 (4 minutes)
• Chat over a content-centric network (p. 55) 02021-01-15 (updated 02021-01-16) (3 minutes)
• Can transactions solve the N+1 performance problem on web pages? (p. 67) 02021-01-16 (8 minutes)
• Compiling machine-code loops to pipelined dataflow graphs (p. 81) 02021-01-23 (updated 02021-01-27) (2 minutes)
• Trying and failing to design an efficient index for folksonomy data based on BDDs (p. 108) 02021-01-26 (updated 02021-01-27) (7 minutes)
• ASCII art, but in Unicode, with Braille and other alternatives (p. 128) 02021-02-10 (updated 02021-02-24) (9 minutes)
• How do you fit a high-level language into a microcontroller? Let’s look at BBN Lisp (p. 160) 02021-02-23 (updated 02021-08-18) (76 minutes)
• Some notes on IPL-VI, Lisp’s 01958 precursor (p. 196) 02021-03-02 (4 minutes)
• A survey of imperative programming operations’ prevalence (p. 201) 02021-03-02 (updated 02021-09-11) (61 minutes)
• Vaughan Pratt and Henry Baker’s COMFY control-flow combinators (p. 234) 02021-03-04 (updated 02021-03-20) (8 minutes)
• Generating novel unique pronounceable identifiers with letter frequency data (p. 239) 02021-03-10 (updated 02021-03-22) (11 minutes)
• Garbage-collected allocation performance on current computers (p. 245) 02021-03-13 (updated 02021-04-08) (4 minutes)
• Brute force speech (p. 262) 02021-03-21 (updated 02021-03-22) (7 minutes)
• Minor improvements to pattern matching (p. 306) 02021-03-24 (updated 02021-04-08) (10 minutes)
• Safe FORTH with the FORTRAN memory model? (p. 351) 02021-04-21 (updated 02021-06-12) (2 minutes)
• Diskstrings: Bernstein’s netstrings for single-pass streaming output (p. 356) 02021-04-21 (updated 02021-07-27) (4 minutes)
• Greek operating systems (p. 430) 02021-06-04 (updated 02021-06-12) (4 minutes)
• The algebra of N-ary relations (p. 432) 02021-06-14 (updated 02021-07-27) (4 minutes)
• PEG-like flexibility for parsing right-to-left? (p. 437) 02021-06-16 (updated 02021-07-27) (2 minutes)

• How little code can a filesystem be? (p. 438) 02021-06-16 (updated 02021-07-27) (1 minute)
• Notes on what would be needed to drive a PS/2 keyboard from an Arduino (p. 447) 02021-06-20 (updated 02021-12-30) (12 minutes)
• Self hosting kernel (p. 452) 02021-06-21 (updated 02021-12-30) (1 minute)
• Stack syntax (p. 453) 02021-06-22 (updated 02021-07-27) (4 minutes)
• Bead hypertext (p. 455) 02021-06-22 (updated 02021-12-30) (1 minute)
• Simple linear-time linear-space nested delimiter parsing (p. 459) 02021-06-24 (updated 02021-12-30) (1 minute)
• Smolsay: the Ur-Lisp, but with dicts instead of conses (p. 552) 02021-07-12 (updated 02021-07-27) (22 minutes)
• Improvements on C for low-level programming such as block arguments (p. 584) 02021-07-14 (updated 02021-12-30) (8 minutes)
• Ropes with constant-time concatenation and equality comparisons with monoidal hash consing (p. 619) 02021-07-27 (15 minutes)
• Compilation of block arguments to high-performance code (p. 624) 02021-07-29 (updated 02021-12-30) (19 minutes)
• Lazy heapsort (p. 761) 02021-08-22 (updated 02021-09-11) (6 minutes)
• Deriving binary search (p. 855) 02021-10-01 (updated 02021-12-30) (5 minutes)
• Some notes on perusing the Udanax Green codebase (p. 860) 02021-10-05 (updated 02021-10-08) (12 minutes)
• Fung's "I can't believe it can sort" algorithm and others (p. 864) 02021-10-05 (updated 02021-12-30) (5 minutes)
• Some notes on learning Rust (p. 874) 02021-10-06 (updated 02021-10-10) (39 minutes)
• PBKDF content addressing with keyphrase hashcash: a non-blockchain attack on Zooko's Triangle (p. 896) 02021-10-08 (24 minutes)
• Wordlists for maximum drama (p. 904) 02021-10-08 (updated 02021-12-30) (16 minutes)
• Pipelined piece chain painting (p. 926) 02021-10-10 (updated 02021-12-30) (23 minutes)
• An algebra of partial functions for interactively composing programs (p. 933) 02021-10-10 (updated 02021-12-30) (3 minutes)
• Balanced ropes (p. 948) 02021-10-16 (updated 02021-12-30) (7 minutes)
• The astounding UI responsivity of PDP-10 DDT on ITS (p. 972) 02021-10-22 (updated 02021-10-23) (28 minutes)
• Example based regexp (p. 984) 02021-10-24 (updated 02021-12-30) (5 minutes)
• Adversarial control (p. 987) 02021-10-25 (updated 02021-12-30) (13 minutes)
• Constant weight dithering (p. 991) 02021-10-28 (updated 02021-12-30) (5 minutes)
• Safe decentralized cloud storage (p. 1135) 02021-12-30 (10 minutes)

Notes concerning “Contrivances”

• Fan noise would be less annoying if intermittent (p. 21) 02021-01-03 (updated 02021-01-04) (1 minute)
• iPhone replacement cameras as 6-μs streak cameras (p. 80) 02021-01-22 (updated 02021-12-30) (2 minutes)
• Energy autonomous computing (p. 143) 02021-02-18 (updated 02021-12-30) (58 minutes)
• Panelization in PCB manufacturing (p. 193) 02021-02-25 (updated 02021-02-26) (7 minutes)
• Refreshing Flash memory periodically for archival (p. 198) 02021-03-02 (1 minute)
• Geneva wheel stopwork (p. 321) 02021-04-07 (updated 02021-04-08) (6 minutes)
• A bargain-basement Holter monitor with a BOM under US$2.50 (p. 323) 02021-04-07 (updated 02021-07-27) (33 minutes)
• Locking telescope (p. 333) 02021-04-07 (updated 02021-12-30) (2 minutes)
• Logarithmic low-power SERDES (p. 334) 02021-04-08 (4 minutes)

• Forming steel with copper instead of vice versa (p. 344) 02021-04-16 (updated 02021-06-12) (2 minutes)
• Phased-array imaging sonar from a mesh network of self-localizing sensor nodes (p. 358) 02021-04-27 (updated 02021-12-30) (8 minutes)

• A boiler for submillisecond steam pulses (p. 361) 02021-04-28 (updated 02021-12-30) (10 minutes)
• Planetary roller screw worm drive (p. 375) 02021-05-07 (updated 02021-12-30) (4 minutes)
• Weighing an eyelash on an improvised Kibble balance (p. 382) 02021-05-08 (updated 02021-12-30) (3 minutes)
• A four-dimensional keyboard matrix made of linear voltage differential transformers (LVDTs) to get 30 or 180 keys on five pins (p. 390) 02021-05-12 (updated 02021-12-30) (4 minutes)
• Planetary screw potentiometer (p. 392) 02021-05-12 (updated 02021-12-30) (1 minute)
• Omnidirectional wheels (p. 422) 02021-05-30 (updated 02021-12-30) (1 minute)
• Broken hard disks are the cheapest source of ultraprecision components (p. 425) 02021-06-02 (updated 02021-06-12) (3 minutes)

• Micro impact driver (p. 427) 02021-06-02 (updated 02021-06-12) (2 minutes)
• Flux-gate downconversion in a loopstick antenna? (p. 436) 02021-06-15 (updated 02021-07-27) (2 minutes)
• Base 3 gage blocks (p. 468) 02021-06-27 (updated 02021-12-30) (5 minutes)
• Multiple counter-rotating milling cutters to eliminate side loading (p. 470) 02021-06-27 (updated 02021-12-30) (7 minutes)
• Sonic screwdriver resonance (p. 527) 02021-07-06 (updated 02021-12-30) (11 minutes)

• Subnanosecond thermochromic light modulation for real-time holography and displays (p. 531) 02021-07-06 (updated 02021-12-30) (8 minutes)
• Spreadtool (p. 543) 02021-07-10 (updated 02021-12-30) (30 minutes)
• The ayurvedic “fire mud” of Bhudeb Mookerji and modern castable refractories (p. 688) 02021-08-05 (updated 02021-08-15) (22 minutes)

• Arc maker (p. 695) 02021-08-07 (updated 02021-12-30) (11 minutes)
• Pocket kiln (p. 704) 02021-08-09 (updated 02021-08-15) (7 minutes)
• Recursive bearings (p. 764) 02021-08-23 (updated 02021-12-30) (1 minute)
• Sorption vacuum pumps really can’t operate continuously (p. 767) 02021-08-24 (updated 02021-09-11) (5 minutes)
• Better screw head designs? (p. 770) 02021-08-25 (updated 02021-09-11) (4 minutes)
• Weighing balance design (p. 799) 02021-09-06 (updated 02021-12-30) (9 minutes)
• Fast-slicing ECM (p. 802) 02021-09-08 (updated 02021-12-30) (3 minutes)
• Compliance spectroscopy (p. 849) 02021-09-29 (updated 02021-12-30) (4 minutes)
• Liquid dielectrics for hand-rolled self-healing capacitors (p. 853) 02021-09-30 (updated 02021-12-30) (3 minutes)
• Triggering a spark gap with low jitter using ultraviolet LEDs? (p. 954) 02021-10-20 (updated 02021-10-23) (8 minutes)
• Thread rolling roller screw (p. 999) 02021-11-04 (updated 02021-12-30) (1 minute)
• An aluminum pencil for marking iron? (p. 1001) 02021-11-06 (updated 02021-12-30) (2 minutes)
• Wire brush microscope (p. 1006) 02021-11-06 (updated 02021-12-30) (1 minute)
• Ivan Miranda’s snap-pin fasteners and similar snaps (p. 1009) 02021-11-11 (updated 02021-12-30) (3 minutes)
• Aqueous scanning probe microscopy (p. 1013) 02021-11-12 (updated 02021-12-30) (7 minutes)
• Some notes on reading parts of Reuleaux’s engineering handbook (p. 1019) 02021-11-17 (updated 02021-12-30) (7 minutes)
• Vernier indicator (p. 1040) 02021-11-22 (updated 02021-12-30) (6 minutes)
• Regenerative muffle kiln (p. 1108) 02021-12-21 (updated 02021-12-30) (19 minutes)
• Toggling eccentrics for removing preload from spring clamps (p. 1129) 02021-12-28 (updated 02021-12-31) (22 minutes)

Notes concerning “Electronics”

• Notes on simulating a ZVS converter (Baxandall converter) (p. 70) 02021-01-16 (6 minutes)
• A ghetto linear voltage regulator from discrete components (p. 73) 02021-01-21 (updated 02021-01-27) (10 minutes)
• Trying to design a simple switchmode power supply using Schmitt-trigger relaxation oscillators (p. 92) 02021-01-26 (updated 02021-01-27) (32 minutes)
• Snap logic, revisited, and four-phase logic (p. 115) 02021-02-08 (9 minutes)
• Can you do direct digital synthesis (DDS) at over a gigahertz? (p. 119) 02021-02-08 (updated 02021-02-24) (30 minutes)
• Energy autonomous computing (p. 143) 02021-02-18 (updated 02021-12-30) (58 minutes)
• Panelization in PCB manufacturing (p. 193) 02021-02-25 (updated 02021-02-26) (7 minutes)
• Refreshing Flash memory periodically for archival (p. 198) 02021-03-02 (1 minute)
• Bench supply (p. 250) 02021-03-19 (updated 02021-12-30) (25 minutes)
• Failing to stabilize the amplitude of an opamp phase-delay oscillator (p. 298) 02021-03-23 (updated 02021-03-24) (10 minutes)
• A bargain-basement Holter monitor with a BOM under US$2.50 (p. 323) 02021-04-07 (updated 02021-07-27) (33 minutes)
• Logarithmic low-power SERDES (p. 334) 02021-04-08 (4 minutes)

• Notes on pricing of locally available oscilloscopes (p. 346) 02021-04-16 (updated 02021-07-27) (2 minutes)
• Can you get JLCPCB to fabricate a CPU for you affordably from “basic” parts? (p. 347) 02021-04-17 (updated 02021-12-30) (9 minutes)
• A boiler for submillisecond steam pulses (p. 361) 02021-04-28 (updated 02021-12-30) (10 minutes)
• Three phase logic (p. 364) 02021-04-30 (updated 02021-07-27) (9 minutes)
• A four-dimensional keyboard matrix made of linear voltage differential transformers (LVDTs) to get 30 or 180 keys on five pins (p. 390) 02021-05-12 (updated 02021-12-30) (4 minutes)
• Planetary screw potentiometer (p. 392) 02021-05-12 (updated 02021-12-30) (1 minute)
• Broken hard disks are the cheapest source of ultraprecision components (p. 425) 02021-06-02 (updated 02021-06-12) (3 minutes)

• Micro impact driver (p. 427) 02021-06-02 (updated 02021-06-12) (2 minutes)
• Flux-gate downconversion in a loopstick antenna? (p. 436) 02021-06-15 (updated 02021-07-27) (2 minutes)
• Notes on what would be needed to drive a PS/2 keyboard from an Arduino (p. 447) 02021-06-20 (updated 02021-12-30) (12 minutes)
• Arc maker (p. 695) 02021-08-07 (updated 02021-12-30)

(11 minutes)
• Power transistors (p. 700) 02021-08-07 (updated 02021-12-30) (12 minutes)
• Constant current buck (p. 708) 02021-08-10 (updated 02021-08-15) (4 minutes)
• Weighing balance design (p. 799) 02021-09-06 (updated 02021-12-30) (9 minutes)
• Switching kiloamps in microseconds (p. 804) 02021-09-09 (updated 02021-12-30) (1 minute)
• Three phase differential data (p. 843) 02021-09-22 (updated 02021-12-30) (4 minutes)
• Liquid dielectrics for hand-rolled self-healing capacitors (p. 853) 02021-09-30 (updated 02021-12-30) (3 minutes)
• Ranking MOSFETs for, say, rapid localized electrolysis to make optics (p. 938) 02021-10-11 (updated 02021-12-30) (8 minutes)
• An even simpler offline power supply than a capacitive dropper, with a 7¢ BOM (p. 943) 02021-10-14 (updated 02021-12-30) (7 minutes)
• Triggering a spark gap with an exploding wire (p. 953) 02021-10-19 (updated 02021-12-30) (1 minute)
• Triggering a spark gap with low jitter using ultraviolet LEDs? (p. 954) 02021-10-20 (updated 02021-10-23) (8 minutes)
• Implementation and applications of low-voltage Marx generators with solid-state avalanche breakdown? (p. 960) 02021-10-20 (updated 02021-12-31) (39 minutes)
• Viscoelastic probing (p. 1000) 02021-11-04 (updated 02021-12-30) (2 minutes)
• A simple 2-D programmable graphics pipeline to unify tiles and palettes (p. 1022) 02021-11-18 (updated 02021-12-30) (6 minutes)
• Simplest blinker (p. 1053) 02021-12-01 (updated 02021-12-30) (9 minutes)
• Capacitive linear encoder sensors (p. 1056) 02021-12-11 (updated 02021-12-30) (7 minutes)
• MOSFET body diodes as Geiger counter avalanche detectors? (p. 1103) 02021-12-17 (updated 02021-12-30) (1 minute)

Notes concerning "Pricing"

• Duplicating Durham's Rock-Hard Putty (p. 79) 02021-01-22 (updated 02021-01-27) (1 minute)
• The use of silver in solar cells (p. 112) 02021-02-02 (updated 02021-09-11) (8 minutes)
• Can you do direct digital synthesis (DDS) at over a gigahertz? (p. 119) 02021-02-08 (updated 02021-02-24) (30 minutes)
• Energy autonomous computing (p. 143) 02021-02-18 (updated 02021-12-30) (58 minutes)
• Panelization in PCB manufacturing (p. 193) 02021-02-25 (updated 02021-02-26) (7 minutes)
• A bargain-basement Holter monitor with a BOM under US$2.50 (p. 323) 02021-04-07 (updated 02021-07-27) (33 minutes)
• Notes on pricing of locally available oscilloscopes (p. 346) 02021-04-16 (updated 02021-07-27) (2 minutes)
• Can you get JLCPCB to fabricate a CPU for you affordably from "basic" parts? (p. 347) 02021-04-17 (updated 02021-12-30) (9 minutes)
• Three phase logic (p. 364) 02021-04-30 (updated 02021-07-27) (9 minutes)
• Aluminum foil (p. 413) 02021-05-24 (updated 02021-09-11) (14 minutes)
• Ghetto electrical discharge machining (EDM) (p. 423) 02021-05-31 (updated 02021-12-30) (5 minutes)
• Broken hard disks are the cheapest source of ultraprecision components (p. 425) 02021-06-02 (updated 02021-06-12) (3 minutes)

• Notes on Richards et al.'s nascent catalytic ROS water treatment process (p. 534) 02021-07-07 (updated 02021-07-27) (14 minutes)
• Spreadtool (p. 543) 02021-07-10 (updated 02021-12-30) (30 minutes)
• Potential local sources and prices of refractory materials (p. 566) 02021-07-14 (updated 02021-09-11) (9 minutes)
• Fiberglass CMCs? (p. 588) 02021-07-15 (updated 02021-07-27) (8 minutes)
• Leaf vein roof (p. 600) 02021-07-16 (updated 02021-09-11) (9 minutes)
• SHS of magnesium phosphate (p. 608) 02021-07-22 (updated 02021-07-27) (3 minutes)
• Material observations (p. 633) 02021-07-29 (updated 02021-12-30) (194 minutes)
• Dipropylene glycol (p. 687) 02021-08-01 (updated 02021-08-15) (2 minutes)
• The ayurvedic "fire mud" of Bhudeb Mookerji and modern castable refractories (p. 688) 02021-08-05 (updated 02021-08-15) (22 minutes)

• Arc maker (p. 695) 02021-08-07 (updated 02021-12-30) (11 minutes)
• Argentine pricing of PEX pipe and alternatives for phase-change fluids (p. 699) 02021-08-07 (updated 02021-12-30) (2 minutes)

• Power transistors (p. 700) 02021-08-07 (updated 02021-12-30) (12 minutes)
• Methane bag (p. 710) 02021-08-10 (updated 02021-08-15) (8 minutes)
• Recursive bearings (p. 764) 02021-08-23 (updated 02021-12-30) (1 minute)
• Dense fillers (p. 772) 02021-08-25 (updated 02021-12-30) (7 minutes)
• Switching kiloamps in microseconds (p. 804) 02021-09-09 (updated 02021-12-30) (1 minute)
• Spot welding (p. 805) 02021-09-09 (updated 02021-12-30) (8 minutes)
• PBKDF content addressing with keyphrase hashcash: a non-blockchain attack on Zooko's Triangle (p. 896) 02021-10-08 (24 minutes)
• Beyond overstrike (p. 922) 02021-10-10 (updated 02021-12-30) (13 minutes)
• Ranking MOSFETs for, say, rapid localized electrolysis to make optics (p. 938) 02021-10-11 (updated 02021-12-30) (8 minutes)
• An even simpler offline power supply than a capacitive dropper, with a 7¢ BOM (p. 943) 02021-10-14 (updated 02021-12-30) (7 minutes)
• Triggering a spark gap with low jitter using ultraviolet LEDs? (p. 954) 02021-10-20 (updated 02021-10-23) (8 minutes)
• Implementation and applications of low-voltage Marx generators with solid-state avalanche breakdown? (p. 960) 02021-10-20 (updated 02021-12-31) (39 minutes)

Notes concerning “Digital fabrication”

• Can you get JLCPCB to fabricate a CPU for you affordably from “basic” parts? (p. 347) 02021-04-17 (updated 02021-12-30) (9 minutes)
• How fast do von Neumann probes need to reproduce to colonize space in our lifetimes? (p. 368) 02021-05-04 (updated 02021-06-12) (5 minutes)
• Thixotropic electrodeposition (p. 372) 02021-05-04 (updated 02021-12-31) (2 minutes)
• Cheap cutting jig (p. 373) 02021-05-06 (updated 02021-12-30) (1 minute)
• Fresnel mirror electropolishing (p. 377) 02021-05-08 (updated 02021-12-30) (6 minutes)
• Precisely measuring out particulates with a trickler (p. 384) 02021-05-09 (updated 02021-12-30) (17 minutes)
• 3-D printing in carbohydrates (p. 393) 02021-05-16 (updated 02021-12-30) (10 minutes)
• Clay-filled PLA filament for firing to ceramic (p. 396) 02021-05-17 (updated 02021-12-30) (1 minute)
• Multicolor filament (p. 397) 02021-05-17 (updated 02021-12-30) (5 minutes)
• Acicular low binder pastes (p. 399) 02021-05-19 (updated 02021-12-30) (1 minute)
• Cutting clay (p. 400) 02021-05-19 (updated 02021-12-30) (10 minutes)
• Selectively curing one-component silicone by injecting water (p. 408) 02021-05-19 (updated 02021-12-30) (2 minutes)
• Clay wire cutter (p. 409) 02021-05-21 (updated 02021-12-30) (2 minutes)
• Aluminum foil (p. 413) 02021-05-24 (updated 02021-09-11) (14 minutes)
• Glass powder-bed 3-D printing (p. 490) 02021-06-29 (updated 02021-12-30) (20 minutes)
• Spreadtool (p. 543) 02021-07-10 (updated 02021-12-30) (30 minutes)
• Electrolytic berlinite (p. 561) 02021-07-12 (updated 02021-12-30) (7 minutes)
• Can you 3-D print Sorel cement by inhibiting setting with X-rays? (p. 592) 02021-07-16 (updated 02021-07-27) (1 minute)
• Iodine patterning (p. 713) 02021-08-11 (updated 02021-08-15) (1 minute)
• A construction set using SHS (p. 765) 02021-08-24 (updated 02021-09-11) (5 minutes)
• Negative feedback control to prevent runaway positive feedback in 3-D MIG welding printing (p. 777) 02021-08-30 (updated 02021-12-30) (3 minutes)
• Patterning metal surfaces by coating decomposition with lasers or plasma? (p. 795) 02021-09-03 (updated 02021-12-30) (7 minutes)

- The sol-gel transition and selective gelling for 3-D printing (p. 858) 02021-10-03 (updated 02021-12-30) (6 minutes)
- Ranking MOSFETs for, say, rapid localized electrolysis to make optics (p. 938) 02021-10-11 (updated 02021-12-30) (8 minutes)
- Trying to quantify relative speeds of different digital fabrication processes with “matter bandwidth” (p. 946) 02021-10-15 (updated 02021-12-30) (5 minutes)
- Adversarial control (p. 987) 02021-10-25 (updated 02021-12-30) (13 minutes)
- Solid rock on a gossamer skeleton through exponential deposition (p. 1076) 02021-12-15 (updated 02021-12-30) (11 minutes)
- 3-D printing in poly(vinyl alcohol) (p. 1080) 02021-12-15 (updated 02021-12-30) (2 minutes)
- Electrolytic 2-D cutting and related electrolytic digital fabrication processes (p. 1085) 02021-12-16 (updated 02021-12-30) (48 minutes)
- Layers plus electroforming (p. 1100) 02021-12-16 (updated 02021-12-30) (7 minutes)
- Toggling eccentrics for removing preload from spring clamps (p. 1129) 02021-12-28 (updated 02021-12-31) (22 minutes)

Notes concerning “Manufacturing”

• The use of silver in solar cells (p. 112) 02021-02-02 (updated 02021-09-11) (8 minutes)
• Panelization in PCB manufacturing (p. 193) 02021-02-25 (updated 02021-02-26) (7 minutes)
• A bargain-basement Holter monitor with a BOM under US$2.50 (p. 323) 02021-04-07 (updated 02021-07-27) (33 minutes)
• Forming steel with copper instead of vice versa (p. 344) 02021-04-16 (updated 02021-06-12) (2 minutes)
• Can you get JLCPCB to fabricate a CPU for you affordably from “basic” parts? (p. 347) 02021-04-17 (updated 02021-12-30) (9 minutes)
• Three phase logic (p. 364) 02021-04-30 (updated 02021-07-27) (9 minutes)
• Electroforming rivets (p. 410) 02021-05-22 (updated 02021-12-30) (2 minutes)
• Metal welding fuel (p. 411) 02021-05-23 (updated 02021-12-30) (6 minutes)
• Ghetto electrical discharge machining (EDM) (p. 423) 02021-05-31 (updated 02021-12-30) (5 minutes)
• Base 3 gage blocks (p. 468) 02021-06-27 (updated 02021-12-30) (5 minutes)
• Multiple counter-rotating milling cutters to eliminate side loading (p. 470) 02021-06-27 (updated 02021-12-30) (7 minutes)
• Layered ECM (p. 473) 02021-06-27 (updated 02021-12-30) (2 minutes)
• Can you use stabilized cubic zirconia as an ECM cathode in molten salt? (p. 475) 02021-06-27 (updated 02021-12-30) (3 minutes)
• Electrolytic glass machining (p. 477) 02021-06-28 (updated 02021-12-30) (6 minutes)
• Super ghetto digital fabrication (p. 479) 02021-06-28 (updated 02021-12-30) (33 minutes)
• Sulfur jet metal cutting (p. 504) 02021-06-30 (updated 02021-12-30) (6 minutes)
• Stochastically generated self-amalgamating tape variations for composite fabrication (p. 510) 02021-07-02 (updated 02021-12-30) (26 minutes)
• Spin-coating clay-filled plastics to make composites with high anisotropic filler loadings (p. 521) 02021-07-02 (updated 02021-12-30) (4 minutes)
• ECM for machining nonmetals? (p. 523) 02021-07-05 (updated 02021-07-27) (11 minutes)
• Spreadtool (p. 543) 02021-07-10 (updated 02021-12-30) (30 minutes)
• Tetrahedral expanded metal (p. 593) 02021-07-16 (updated 02021-07-27) (3 minutes)
• Glass foam (p. 595) 02021-07-16 (updated 02021-08-15) (17 minutes)

- Electrodeposition welding (p. 769) 02021-08-25 (updated 02021-09-11) (2 minutes)
- Dense fillers (p. 772) 02021-08-25 (updated 02021-12-30) (7 minutes)
- Fast-slicing ECM (p. 802) 02021-09-08 (updated 02021-12-30) (3 minutes)
- Spot welding (p. 805) 02021-09-09 (updated 02021-12-30) (8 minutes)
- Waterglass “Loctite”? (p. 845) 02021-09-22 (updated 02021-12-30) (1 minute)
- Blowing agents (p. 847) 02021-09-29 (updated 02021-12-30) (4 minutes)
- Adversarial control (p. 987) 02021-10-25 (updated 02021-12-30) (13 minutes)

Notes concerning "History"

• Layout typescript (p. 29) 02021-01-04 (5 minutes)
• Skew tilesets (p. 135) 02021-02-14 (updated 02021-02-24) (7 minutes)
• Threechet (p. 140) 02021-02-16 (updated 02021-02-24) (4 minutes)

• How do you fit a high-level language into a microcontroller? Let's look at BBN Lisp (p. 160) 02021-02-23 (updated 02021-08-18) (76 minutes)
• Some notes on IPL-VI, Lisp's 01958 precursor (p. 196) 02021-03-02 (4 minutes)
• Vaughan Pratt and Henry Baker's COMFY control-flow combinators (p. 234) 02021-03-04 (updated 02021-03-20) (8 minutes)
• Veskeno is a "fantasy platform" like TIC-80 (p. 267) 02021-03-21 (updated 02021-03-22) (3 minutes)
• Why Bitcoin is puzzling to people in rich countries (p. 312) 02021-03-31 (updated 02021-07-27) (10 minutes)
• Manually writing code in static single assignment (SSA) form, inspired by Kemeny's DOPE, isn't worth it (p. 353) 02021-04-21 (updated 02021-06-12) (3 minutes)
• Scaling laws (p. 404) 02021-05-19 (updated 02021-12-30) (8 minutes)
• The nature of mathematical discourse (p. 418) 02021-05-27 (updated 02021-12-30) (5 minutes)
• Economic history (p. 460) 02021-06-25 (updated 02021-07-27) (17 minutes)
• Can you use stabilized cubic zirconia as an ECM cathode in molten salt? (p. 475) 02021-06-27 (updated 02021-12-30) (3 minutes)
• The ayurvedic "fire mud" of Bhudeb Mookerji and modern castable refractories (p. 688) 02021-08-05 (updated 02021-08-15) (22 minutes)

• A compact bytecode sketch that should average about 3 bytes per line of C (p. 720) 02021-08-17 (updated 02021-09-13) (66 minutes)
• A short list of the most useful Unix CLI tools (p. 841) 02021-09-15 (updated 02021-09-16) (2 minutes)
• Some notes on perusing the Udanax Green codebase (p. 860) 02021-10-05 (updated 02021-10-08) (12 minutes)
• Beyond overstrike (p. 922) 02021-10-10 (updated 02021-12-30) (13 minutes)
• Pipelined piece chain painting (p. 926) 02021-10-10 (updated 02021-12-30) (23 minutes)
• The astounding UI responsivity of PDP-10 DDT on ITS (p. 972) 02021-10-22 (updated 02021-10-23) (28 minutes)
• My Heathkit H8 (p. 996) 02021-11-03 (updated 02021-12-30) (2 minutes)
• Some notes on reading parts of Reuleaux's engineering handbook (p. 1019) 02021-11-17 (updated 02021-12-30) (7 minutes)
• Interesting works that entered the public domain in 02021, in the US and elsewhere (p. 1024) 02021-11-20 (updated 02021-12-30)

(15 minutes)

- Is liberal democracy's stability conditioned on historical conditions that no longer obtain? (p. 1114) 02021-12-22 (updated 02021-12-30) (16 minutes)

Notes concerning “Performance”

• Principled APL redux (p. 22) 02021-01-03 (updated 02021-12-31) (12 minutes)
• Relayout with heaps (p. 32) 02021-01-10 (updated 02021-01-15) (6 minutes)
• Trie PEGs (p. 53) 02021-01-15 (4 minutes)
• Can transactions solve the N+1 performance problem on web pages? (p. 67) 02021-01-16 (8 minutes)
• Compiling machine-code loops to pipelined dataflow graphs (p. 81) 02021-01-23 (updated 02021-01-27) (2 minutes)
• Trying and failing to design an efficient index for folksonomy data based on BDDs (p. 108) 02021-01-26 (updated 02021-01-27) (7 minutes)
• How do you fit a high-level language into a microcontroller? Let’s look at BBN Lisp (p. 160) 02021-02-23 (updated 02021-08-18) (76 minutes)
• Variable length unaligned bytecode (p. 199) 02021-03-02 (updated 02021-03-03) (4 minutes)
• A survey of imperative programming operations’ prevalence (p. 201) 02021-03-02 (updated 02021-09-11) (61 minutes)
• Garbage-collected allocation performance on current computers (p. 245) 02021-03-13 (updated 02021-04-08) (4 minutes)
• Some notes on reading Chris Seaton’s TruffleRuby dissertation (p. 269) 02021-03-21 (updated 02021-03-22) (16 minutes)
• Open coded primitives (p. 283) 02021-03-22 (26 minutes)
• Minor improvements to pattern matching (p. 306) 02021-03-24 (updated 02021-04-08) (10 minutes)
• Faygoo: a yantra-smashing ersatz version of Piumarta and Warth’s COLA (p. 570) 02021-07-14 (updated 02021-12-30) (17 minutes)
• Ropes with constant-time concatenation and equality comparisons with monoidal hash consing (p. 619) 02021-07-27 (15 minutes)
• Compilation of block arguments to high-performance code (p. 624) 02021-07-29 (updated 02021-12-30) (19 minutes)
• A compact bytecode sketch that should average about 3 bytes per line of C (p. 720) 02021-08-17 (updated 02021-09-13) (66 minutes)
• Lazy heapsort (p. 761) 02021-08-22 (updated 02021-09-11) (6 minutes)
• Qfitzah: a minimal term-rewriting language (p. 809) 02021-09-10 (updated 02021-12-31) (62 minutes)
• Fung’s “I can’t believe it can sort” algorithm and others (p. 864) 02021-10-05 (updated 02021-12-30) (5 minutes)
• Pipelined piece chain painting (p. 926) 02021-10-10 (updated 02021-12-30) (23 minutes)
• Hashing text with recursive shingling to find duplication efficiently (p. 993) 02021-10-30 (updated 02021-12-30) (6 minutes)

Notes concerning “Human-computer interaction”

• First class locations (p. 27) 02021-01-04 (3 minutes)
• Layout typescript (p. 29) 02021-01-04 (5 minutes)
• Relayout with heaps (p. 32) 02021-01-10 (updated 02021-01-15) (6 minutes)
• Transactional editor (p. 35) 02021-01-14 (updated 02021-01-15) (73 minutes)
• Running scripts once per frame for guaranteed GUI responsivity (p. 303) 02021-03-23 (updated 02021-10-12) (7 minutes)
• List of random GUI ideas (p. 370) 02021-05-04 (updated 02021-07-27) (6 minutes)
• Leaf hypertext (p. 380) 02021-05-08 (updated 02021-12-30) (3 minutes)
• A four-dimensional keyboard matrix made of linear voltage differential transformers (LVDTs) to get 30 or 180 keys on five pins (p. 390) 02021-05-12 (updated 02021-12-30) (4 minutes)
• Bead hypertext (p. 455) 02021-06-22 (updated 02021-12-30) (1 minute)
• A kernel you can type commands to (p. 474) 02021-06-27 (updated 02021-12-30) (1 minute)
• Rator-port GUIs (p. 496) 02021-06-29 (updated 02021-12-30) (26 minutes)
• Memory view (p. 539) 02021-07-09 (updated 02021-12-30) (6 minutes)
• Wiki models (p. 751) 02021-08-19 (updated 02021-12-30) (1 minute)
• Beyond overstrike (p. 922) 02021-10-10 (updated 02021-12-30) (13 minutes)
• Inverse perspective (p. 937) 02021-10-11 (updated 02021-12-30) (1 minute)
• The astounding UI responsivity of PDP-10 DDT on ITS (p. 972) 02021-10-22 (updated 02021-10-23) (28 minutes)
• Example based regexp (p. 984) 02021-10-24 (updated 02021-12-30) (5 minutes)
• Embedding runnable code in text paragraphs for numerical modeling (p. 1002) 02021-11-06 (updated 02021-12-30) (6 minutes)
• DSLs for calculations on dates (p. 1018) 02021-11-14 (updated 02021-12-30) (1 minute)
• Chording commands (p. 1047) 02021-11-26 (updated 02021-12-30) (7 minutes)
• Two finger multitouch (p. 1059) 02021-12-11 (updated 02021-12-30) (3 minutes)
• The user interface potentialities of a barcoded paper notebook (p. 1104) 02021-12-18 (updated 02021-12-30) (6 minutes)

Notes concerning “Physics”

• When is it better to compute by moving atoms rather than electrons? (p. 265) 02021-03-21 (updated 02021-03-22) (5 minutes)
• Forming steel with copper instead of vice versa (p. 344) 02021-04-16 (updated 02021-06-12) (2 minutes)
• Phased-array imaging sonar from a mesh network of self-localizing sensor nodes (p. 358) 02021-04-27 (updated 02021-12-30) (8 minutes)

• A boiler for submillisecond steam pulses (p. 361) 02021-04-28 (updated 02021-12-30) (10 minutes)
• Scaling laws (p. 404) 02021-05-19 (updated 02021-12-30) (8 minutes)
• Metal welding fuel (p. 411) 02021-05-23 (updated 02021-12-30) (6 minutes)
• Spreadtool (p. 543) 02021-07-10 (updated 02021-12-30) (30 minutes)
• Back-drivable differential windlass (p. 610) 02021-07-23 (updated 02021-07-27) (15 minutes)
• Methane bag (p. 710) 02021-08-10 (updated 02021-08-15) (8 minutes)
• Sorption vacuum pumps really can’t operate continuously (p. 767) 02021-08-24 (updated 02021-09-11) (5 minutes)
• Selective laser sintering of copper (p. 775) 02021-08-30 (updated 02021-12-30) (6 minutes)
• Spot welding (p. 805) 02021-09-09 (updated 02021-12-30) (8 minutes)
• Liquid dielectrics for hand-rolled self-healing capacitors (p. 853) 02021-09-30 (updated 02021-12-30) (3 minutes)
• Aqueous scanning probe microscopy (p. 1013) 02021-11-12 (updated 02021-12-30) (7 minutes)
• Micro ramjet (p. 1038) 02021-11-22 (updated 02021-12-30) (3 minutes)
• MOSFET body diodes as Geiger counter avalanche detectors? (p. 1103) 02021-12-17 (updated 02021-12-30) (1 minute)
• Photoemissive power (p. 1124) 02021-12-23 (updated 02021-12-28) (15 minutes)
• Toggling eccentrics for removing preload from spring clamps (p. 1129) 02021-12-28 (updated 02021-12-31) (22 minutes)

Notes concerning “Electrolysis”

• Fresnel mirror electropolishing (p. 377) 02021-05-08 (updated 02021-12-30) (6 minutes)
• Aluminum foil (p. 413) 02021-05-24 (updated 02021-09-11) (14 minutes)
• Layered ECM (p. 473) 02021-06-27 (updated 02021-12-30) (2 minutes)
• Can you use stabilized cubic zirconia as an ECM cathode in molten salt? (p. 475) 02021-06-27 (updated 02021-12-30) (3 minutes)
• Electrolytic glass machining (p. 477) 02021-06-28 (updated 02021-12-30) (6 minutes)
• ECM for machining nonmetals? (p. 523) 02021-07-05 (updated 02021-07-27) (11 minutes)
• Electrolytic berlinite (p. 561) 02021-07-12 (updated 02021-12-30) (7 minutes)
• Electrodeposition welding (p. 769) 02021-08-25 (updated 02021-09-11) (2 minutes)
• Fast electrolytic mineral accretion (seacrete) for digital fabrication? (p. 779) 02021-09-01 (updated 02021-12-30) (52 minutes)
• Fast-slicing ECM (p. 802) 02021-09-08 (updated 02021-12-30) (3 minutes)
• The sol-gel transition and selective gelling for 3-D printing (p. 858) 02021-10-03 (updated 02021-12-30) (6 minutes)
• Ranking MOSFETs for, say, rapid localized electrolysis to make optics (p. 938) 02021-10-11 (updated 02021-12-30) (8 minutes)
• At small scales, electrowinning may be cheaper than smelting (p. 1029) 02021-11-21 (updated 02021-12-30) (25 minutes)
• Some notes on Bhattacharyya’s ECM book (p. 1043) 02021-11-25 (updated 02021-12-30) (11 minutes)
• Ghetto electrochromic displays for ultra-low-power computing? (p. 1082) 02021-12-16 (updated 02021-12-30) (9 minutes)
• Electrolytic 2-D cutting and related electrolytic digital fabrication processes (p. 1085) 02021-12-16 (updated 02021-12-30) (48 minutes)
• Layers plus electroforming (p. 1100) 02021-12-16 (updated 02021-12-30) (7 minutes)
• Aluminum refining (p. 1106) 02021-12-20 (updated 02021-12-30) (3 minutes)

Notes concerning “Mechanical”

• When is it better to compute by moving atoms rather than electrons? (p. 265) 02021-03-21 (updated 02021-03-22) (5 minutes)
• Geneva wheel stopwork (p. 321) 02021-04-07 (updated 02021-04-08) (6 minutes)
• Locking telescope (p. 333) 02021-04-07 (updated 02021-12-30) (2 minutes)
• A boiler for submillisecond steam pulses (p. 361) 02021-04-28 (updated 02021-12-30) (10 minutes)
• Planetary roller screw worm drive (p. 375) 02021-05-07 (updated 02021-12-30) (4 minutes)
• Planetary screw potentiometer (p. 392) 02021-05-12 (updated 02021-12-30) (1 minute)
• Omnidirectional wheels (p. 422) 02021-05-30 (updated 02021-12-30) (1 minute)
• Micro impact driver (p. 427) 02021-06-02 (updated 02021-06-12) (2 minutes)
• Spreadtool (p. 543) 02021-07-10 (updated 02021-12-30) (30 minutes)
• Back-drivable differential windlass (p. 610) 02021-07-23 (updated 02021-07-27) (15 minutes)
• Recursive bearings (p. 764) 02021-08-23 (updated 02021-12-30) (1 minute)
• Better screw head designs? (p. 770) 02021-08-25 (updated 02021-09-11) (4 minutes)
• Weighing balance design (p. 799) 02021-09-06 (updated 02021-12-30) (9 minutes)
• Flexural mounts for self-aligning bushings (p. 952) 02021-10-18 (updated 02021-12-30) (3 minutes)
• Thread rolling roller screw (p. 999) 02021-11-04 (updated 02021-12-30) (1 minute)
• Some notes on reading parts of Reuleaux’s engineering handbook (p. 1019) 02021-11-17 (updated 02021-12-30) (7 minutes)
• Toggling eccentrics for removing preload from spring clamps (p. 1129) 02021-12-28 (updated 02021-12-31) (22 minutes)

Notes concerning “3-D printing”

• Precisely measuring out particulates with a trickler (p. 384) 02021-05-09 (updated 02021-12-30) (17 minutes)
• 3-D printing in carbohydrates (p. 393) 02021-05-16 (updated 02021-12-30) (10 minutes)
• Clay-filled PLA filament for firing to ceramic (p. 396) 02021-05-17 (updated 02021-12-30) (1 minute)
• Multicolor filament (p. 397) 02021-05-17 (updated 02021-12-30) (5 minutes)
• Acicular low binder pastes (p. 399) 02021-05-19 (updated 02021-12-30) (1 minute)
• Selectively curing one-component silicone by injecting water (p. 408) 02021-05-19 (updated 02021-12-30) (2 minutes)
• Glass powder-bed 3-D printing (p. 490) 02021-06-29 (updated 02021-12-30) (20 minutes)
• Powder-bed 3-D printing with a sacrificial binder (p. 506) 02021-06-30 (updated 02021-12-30) (12 minutes)
• Electrolytic berlinite (p. 561) 02021-07-12 (updated 02021-12-30) (7 minutes)
• Can you 3-D print Sorel cement by inhibiting setting with X-rays? (p. 592) 02021-07-16 (updated 02021-07-27) (1 minute)
• A construction set using SHS (p. 765) 02021-08-24 (updated 02021-09-11) (5 minutes)
• Selective laser sintering of copper (p. 775) 02021-08-30 (updated 02021-12-30) (6 minutes)
• Negative feedback control to prevent runaway positive feedback in 3-D MIG welding printing (p. 777) 02021-08-30 (updated 02021-12-30) (3 minutes)
• Fast electrolytic mineral accretion (seacrete) for digital fabrication? (p. 779) 02021-09-01 (updated 02021-12-30) (52 minutes)
• The sol-gel transition and selective gelling for 3-D printing (p. 858) 02021-10-03 (updated 02021-12-30) (6 minutes)
• Flexural mounts for self-aligning bushings (p. 952) 02021-10-18 (updated 02021-12-30) (3 minutes)
• 3-D printing in poly(vinyl alcohol) (p. 1080) 02021-12-15 (updated 02021-12-30) (2 minutes)

Notes concerning “Filled systems”

• Clay-filled PLA filament for firing to ceramic (p. 396) 02021-05-17 (updated 02021-12-30) (1 minute)
• Multicolor filament (p. 397) 02021-05-17 (updated 02021-12-30) (5 minutes)
• Acicular low binder pastes (p. 399) 02021-05-19 (updated 02021-12-30) (1 minute)
• Cutting clay (p. 400) 02021-05-19 (updated 02021-12-30) (10 minutes)
• Stochastically generated self-amalgamating tape variations for composite fabrication (p. 510) 02021-07-02 (updated 02021-12-30) (26 minutes)
• Spin-coating clay-filled plastics to make composites with high anisotropic filler loadings (p. 521) 02021-07-02 (updated 02021-12-30) (4 minutes)
• Electrolytic berlinite (p. 561) 02021-07-12 (updated 02021-12-30) (7 minutes)
• The ayurvedic “fire mud” of Bhudeb Mookerji and modern castable refractories (p. 688) 02021-08-05 (updated 02021-08-15) (22 minutes)

• Sandwich panel optimization (p. 754) 02021-08-21 (updated 02021-09-11) (3 minutes)
• Glass wood (p. 755) 02021-08-21 (updated 02021-12-30) (4 minutes)
• Maximizing phosphate density from aqueous reaction (p. 757) 02021-08-21 (updated 02021-12-30) (8 minutes)
• Dense fillers (p. 772) 02021-08-25 (updated 02021-12-30) (7 minutes)
• Fast electrolytic mineral accretion (seacrete) for digital fabrication? (p. 779) 02021-09-01 (updated 02021-12-30) (52 minutes)
• Rock-wool-filled composites (p. 798) 02021-09-03 (updated 02021-12-30) (2 minutes)
• Electrolytic 2-D cutting and related electrolytic digital fabrication processes (p. 1085) 02021-12-16 (updated 02021-12-30) (48 minutes)
• Xerogel compacting (p. 1119) 02021-12-22 (updated 02021-12-30) (12 minutes)

Notes concerning “Experiment report”

• Transactional editor (p. 35) 02021-01-14 (updated 02021-01-15) (73 minutes)
• A survey of imperative programming operations’ prevalence (p. 201) 02021-03-02 (updated 02021-09-11) (61 minutes)
• Generating novel unique pronounceable identifiers with letter frequency data (p. 239) 02021-03-10 (updated 02021-03-22) (11 minutes)
• Brute force speech (p. 262) 02021-03-21 (updated 02021-03-22) (7 minutes)
• Open coded primitives (p. 283) 02021-03-22 (26 minutes)
• Aluminum foil (p. 413) 02021-05-24 (updated 02021-09-11) (14 minutes)
• Notes on what would be needed to drive a PS/2 keyboard from an Arduino (p. 447) 02021-06-20 (updated 02021-12-30) (12 minutes)
• Glass powder-bed 3-D printing (p. 490) 02021-06-29 (updated 02021-12-30) (20 minutes)
• Firing talc (p. 576) 02021-07-14 (updated 02021-12-30) (17 minutes)
• Glass foam (p. 595) 02021-07-16 (updated 02021-08-15) (17 minutes)
• Synthesizing amorphous magnesium silicate (p. 617) 02021-07-25 (updated 02021-08-15) (6 minutes)
• Material observations (p. 633) 02021-07-29 (updated 02021-12-30) (194 minutes)
• Methane bag (p. 710) 02021-08-10 (updated 02021-08-15) (8 minutes)
• Spanish phonology (p. 867) 02021-10-05 (updated 02021-12-31) (15 minutes)

Notes concerning “Algorithms”

• Relayout with heaps (p. 32) 02021-01-10 (updated 02021-01-15) (6 minutes)
• Trie PEGs (p. 53) 02021-01-15 (4 minutes)
• Trying and failing to design an efficient index for folksonomy data based on BDDs (p. 108) 02021-01-26 (updated 02021-01-27) (7 minutes)
• Recursive residue number systems? (p. 259) 02021-03-20 (updated 02021-03-22) (8 minutes)
• Differential filming (p. 374) 02021-05-07 (updated 02021-12-30) (1 minute)
• PEG-like flexibility for parsing right-to-left? (p. 437) 02021-06-16 (updated 02021-07-27) (2 minutes)
• Simple linear-time linear-space nested delimiter parsing (p. 459) 02021-06-24 (updated 02021-12-30) (1 minute)
• Ropes with constant-time concatenation and equality comparisons with monoidal hash consing (p. 619) 02021-07-27 (15 minutes)
• Residual stream windowing (p. 752) 02021-08-21 (updated 02021-09-11) (5 minutes)
• Lazy heapsort (p. 761) 02021-08-22 (updated 02021-09-11) (6 minutes)
• Deriving binary search (p. 855) 02021-10-01 (updated 02021-12-30) (5 minutes)
• Fung’s “I can’t believe it can sort” algorithm and others (p. 864) 02021-10-05 (updated 02021-12-30) (5 minutes)
• Balanced ropes (p. 948) 02021-10-16 (updated 02021-12-30) (7 minutes)
• Hashing text with recursive shingling to find duplication efficiently (p. 993) 02021-10-30 (updated 02021-12-30) (6 minutes)

Notes concerning “Strength of materials”

• Forming steel with copper instead of vice versa (p. 344) 02021-04-16 (updated 02021-06-12) (2 minutes)
• Aluminum foil (p. 413) 02021-05-24 (updated 02021-09-11) (14 minutes)
• Spreadtool (p. 543) 02021-07-10 (updated 02021-12-30) (30 minutes)
• Fiberglass CMCs? (p. 588) 02021-07-15 (updated 02021-07-27) (8 minutes)
• Glass foam (p. 595) 02021-07-16 (updated 02021-08-15) (17 minutes)
• Leaf vein roof (p. 600) 02021-07-16 (updated 02021-09-11) (9 minutes)
• Back-drivable differential windlass (p. 610) 02021-07-23 (updated 02021-07-27) (15 minutes)
• The ayurvedic “fire mud” of Bhudeb Mookerji and modern castable refractories (p. 688) 02021-08-05 (updated 02021-08-15) (22 minutes)
• Sandwich panel optimization (p. 754) 02021-08-21 (updated 02021-09-11) (3 minutes)
• Glass wood (p. 755) 02021-08-21 (updated 02021-12-30) (4 minutes)
• Rock-wool-filled composites (p. 798) 02021-09-03 (updated 02021-12-30) (2 minutes)
• Exotic steel analogues in other metals (p. 1050) 02021-12-01 (updated 02021-12-30) (8 minutes)
• Xerogel compacting (p. 1119) 02021-12-22 (updated 02021-12-30) (12 minutes)

Notes concerning “Machining”

• Forming steel with copper instead of vice versa (p. 344) 02021-04-16 (updated 02021-06-12) (2 minutes)
• Fresnel mirror electropolishing (p. 377) 02021-05-08 (updated 02021-12-30) (6 minutes)
• Aluminum foil (p. 413) 02021-05-24 (updated 02021-09-11) (14 minutes)
• Base 3 gage blocks (p. 468) 02021-06-27 (updated 02021-12-30) (5 minutes)
• Multiple counter-rotating milling cutters to eliminate side loading (p. 470) 02021-06-27 (updated 02021-12-30) (7 minutes)
• Layered ECM (p. 473) 02021-06-27 (updated 02021-12-30) (2 minutes)
• Can you use stabilized cubic zirconia as an ECM cathode in molten salt? (p. 475) 02021-06-27 (updated 02021-12-30) (3 minutes)
• Electrolytic glass machining (p. 477) 02021-06-28 (updated 02021-12-30) (6 minutes)
• Super ghetto digital fabrication (p. 479) 02021-06-28 (updated 02021-12-30) (33 minutes)
• Sulfur jet metal cutting (p. 504) 02021-06-30 (updated 02021-12-30) (6 minutes)
• ECM for machining nonmetals? (p. 523) 02021-07-05 (updated 02021-07-27) (11 minutes)
• Spreadtool (p. 543) 02021-07-10 (updated 02021-12-30) (30 minutes)
• Fast-slicing ECM (p. 802) 02021-09-08 (updated 02021-12-30) (3 minutes)

Notes concerning “Python”

• Relayout with heaps (p. 32) 02021-01-10 (updated 02021-01-15) (6 minutes)
• ASCII art, but in Unicode, with Braille and other alternatives (p. 128) 02021-02-10 (updated 02021-02-24) (9 minutes)
• Skew tilesets (p. 135) 02021-02-14 (updated 02021-02-24) (7 minutes)
• A survey of imperative programming operations’ prevalence (p. 201) 02021-03-02 (updated 02021-09-11) (61 minutes)
• Generating novel unique pronounceable identifiers with letter frequency data (p. 239) 02021-03-10 (updated 02021-03-22) (11 minutes)
• Statistics on the present and future of energy in the People’s Republic of China (p. 316) 02021-04-01 (updated 02021-04-08) (10 minutes)
• Stochastically generated self-amalgamating tape variations for composite fabrication (p. 510) 02021-07-02 (updated 02021-12-30) (26 minutes)
• Memory view (p. 539) 02021-07-09 (updated 02021-12-30) (6 minutes)
• Sorption vacuum pumps really can’t operate continuously (p. 767) 02021-08-24 (updated 02021-09-11) (5 minutes)
• Deriving binary search (p. 855) 02021-10-01 (updated 02021-12-30) (5 minutes)
• Spanish phonology (p. 867) 02021-10-05 (updated 02021-12-31) (15 minutes)
• Balanced ropes (p. 948) 02021-10-16 (updated 02021-12-30) (7 minutes)

Notes concerning “Pulsed machinery”

• Forming steel with copper instead of vice versa (p. 344) 02021-04-16 (updated 02021-06-12) (2 minutes)
• A boiler for submillisecond steam pulses (p. 361) 02021-04-28 (updated 02021-12-30) (10 minutes)
• Micro impact driver (p. 427) 02021-06-02 (updated 02021-06-12) (2 minutes)
• Sonic screwdriver resonance (p. 527) 02021-07-06 (updated 02021-12-30) (11 minutes)
• Switching kiloamps in microseconds (p. 804) 02021-09-09 (updated 02021-12-30) (1 minute)
• Spot welding (p. 805) 02021-09-09 (updated 02021-12-30) (8 minutes)
• The spark-pen pointing device (p. 921) 02021-10-10 (updated 02021-10-12) (1 minute)
• Ranking MOSFETs for, say, rapid localized electrolysis to make optics (p. 938) 02021-10-11 (updated 02021-12-30) (8 minutes)
• Triggering a spark gap with an exploding wire (p. 953) 02021-10-19 (updated 02021-12-30) (1 minute)
• Triggering a spark gap with low jitter using ultraviolet LEDs? (p. 954) 02021-10-20 (updated 02021-10-23) (8 minutes)
• Implementation and applications of low-voltage Marx generators with solid-state avalanche breakdown? (p. 960) 02021-10-20 (updated 02021-12-31) (39 minutes)
• Simplest blinker (p. 1053) 02021-12-01 (updated 02021-12-30) (9 minutes)

Notes concerning "Frrickin' lasers!"

- Broken hard disks are the cheapest source of ultraprecision components (p. 425) 02021-06-02 (updated 02021-06-12) (3 minutes)

- Micro impact driver (p. 427) 02021-06-02 (updated 02021-06-12) (2 minutes)
- Subnanosecond thermochromic light modulation for real-time holography and displays (p. 531) 02021-07-06 (updated 02021-12-30) (8 minutes)
- Constant current buck (p. 708) 02021-08-10 (updated 02021-08-15) (4 minutes)
- Iodine patterning (p. 713) 02021-08-11 (updated 02021-08-15) (1 minute)
- Selective laser sintering of copper (p. 775) 02021-08-30 (updated 02021-12-30) (6 minutes)
- Patterning metal surfaces by coating decomposition with lasers or plasma? (p. 795) 02021-09-03 (updated 02021-12-30) (7 minutes)
- The sol-gel transition and selective gelling for 3-D printing (p. 858) 02021-10-03 (updated 02021-12-30) (6 minutes)
- Trying to quantify relative speeds of different digital fabrication processes with "matter bandwidth" (p. 946) 02021-10-15 (updated 02021-12-30) (5 minutes)
- Adversarial control (p. 987) 02021-10-25 (updated 02021-12-30) (13 minutes)
- At small scales, electrowinning may be cheaper than smelting (p. 1029) 02021-11-21 (updated 02021-12-30) (25 minutes)
- Electrolytic 2-D cutting and related electrolytic digital fabrication processes (p. 1085) 02021-12-16 (updated 02021-12-30) (48 minutes)

Notes concerning “Ghettobotics”

- Notes on simulating a ZVS converter (Baxandall converter) (p. 70) 02021-01-16 (6 minutes)
- A ghetto linear voltage regulator from discrete components (p. 73) 02021-01-21 (updated 02021-01-27) (10 minutes)
- Energy autonomous computing (p. 143) 02021-02-18 (updated 02021-12-30) (58 minutes)
- Bench supply (p. 250) 02021-03-19 (updated 02021-12-30) (25 minutes)
- Cheap cutting jig (p. 373) 02021-05-06 (updated 02021-12-30) (1 minute)
- Weighing an eyelash on an improvised Kibble balance (p. 382) 02021-05-08 (updated 02021-12-30) (3 minutes)
- Ghetto electrical discharge machining (EDM) (p. 423) 02021-05-31 (updated 02021-12-30) (5 minutes)
- Broken hard disks are the cheapest source of ultraprecision components (p. 425) 02021-06-02 (updated 02021-06-12) (3 minutes)
- Super ghetto digital fabrication (p. 479) 02021-06-28 (updated 02021-12-30) (33 minutes)
- Weighing balance design (p. 799) 02021-09-06 (updated 02021-12-30) (9 minutes)
- Capacitive linear encoder sensors (p. 1056) 02021-12-11 (updated 02021-12-30) (7 minutes)
- Ghetto electrochromic displays for ultra-low-power computing? (p. 1082) 02021-12-16 (updated 02021-12-30) (9 minutes)

Notes concerning "Energy"

• The use of silver in solar cells (p. 112) 02021-02-02 (updated 02021-09-11) (8 minutes)
• Energy autonomous computing (p. 143) 02021-02-18 (updated 02021-12-30) (58 minutes)
• Statistics on the present and future of energy in the People's Republic of China (p. 316) 02021-04-01 (updated 02021-04-08) (10 minutes)
• A bargain-basement Holter monitor with a BOM under US$2.50 (p. 323) 02021-04-07 (updated 02021-07-27) (33 minutes)
• Logarithmic low-power SERDES (p. 334) 02021-04-08 (4 minutes)

• Nuclear energy is the Amiga of energy sources (p. 434) 02021-06-14 (updated 02021-07-27) (3 minutes)
• Aluminum fuel (p. 603) 02021-07-17 (updated 02021-12-30) (2 minutes)
• Heating a shower tank with portable TCES? (p. 714) 02021-08-11 (updated 02021-08-15) (6 minutes)
• The relation between solar-panel efficiency for air conditioning and insulation thickness (p. 941) 02021-10-11 (updated 02021-12-30) (3 minutes)
• New nuclear power in the People's Republic of China (p. 1007) 02021-11-09 (updated 02021-12-30) (2 minutes)
• Regenerative muffle kiln (p. 1108) 02021-12-21 (updated 02021-12-30) (19 minutes)
• Photoemissive power (p. 1124) 02021-12-23 (updated 02021-12-28) (15 minutes)

Notes concerning “Bootstrapping”

- Can you get JLCPCB to fabricate a CPU for you affordably from “basic” parts? (p. 347) 02021-04-17 (updated 02021-12-30) (9 minutes)
- Weighing an eyelash on an improvised Kibble balance (p. 382) 02021-05-08 (updated 02021-12-30) (3 minutes)
- How little code can a filesystem be? (p. 438) 02021-06-16 (updated 02021-07-27) (1 minute)
- Self hosting kernel (p. 452) 02021-06-21 (updated 02021-12-30) (1 minute)
- Super ghetto digital fabrication (p. 479) 02021-06-28 (updated 02021-12-30) (33 minutes)
- Material observations (p. 633) 02021-07-29 (updated 02021-12-30) (194 minutes)
- Subset of C for the simplest self-compiling compiler (p. 717) 02021-08-12 (updated 02021-12-30) (6 minutes)
- Weighing balance design (p. 799) 02021-09-06 (updated 02021-12-30) (9 minutes)
- Qfitzah: a minimal term-rewriting language (p. 809) 02021-09-10 (updated 02021-12-31) (62 minutes)
- Triggering a spark gap with an exploding wire (p. 953) 02021-10-19 (updated 02021-12-30) (1 minute)
- Capacitive linear encoder sensors (p. 1056) 02021-12-11 (updated 02021-12-30) (7 minutes)
- Ghetto electrochromic displays for ultra-low-power computing? (p. 1082) 02021-12-16 (updated 02021-12-30) (9 minutes)

Notes concerning “Safe programming languages”

• Principled APL redux (p. 22) 02021-01-03 (updated 02021-12-31) (12 minutes)
• First class locations (p. 27) 02021-01-04 (3 minutes)
• Transactional editor (p. 35) 02021-01-14 (updated 02021-01-15) (73 minutes)
• How do you fit a high-level language into a microcontroller? Let’s look at BBN Lisp (p. 160) 02021-02-23 (updated 02021-08-18) (76 minutes)
• A survey of imperative programming operations’ prevalence (p. 201) 02021-03-02 (updated 02021-09-11) (61 minutes)
• Veskeno is a “fantasy platform” like TIC-80 (p. 267) 02021-03-21 (updated 02021-03-22) (3 minutes)
• Some notes on reading Chris Seaton’s TruffleRuby dissertation (p. 269) 02021-03-21 (updated 02021-03-22) (16 minutes)
• Running scripts once per frame for guaranteed GUI responsivity (p. 303) 02021-03-23 (updated 02021-10-12) (7 minutes)
• Safe FORTH with the FORTRAN memory model? (p. 351) 02021-04-21 (updated 02021-06-12) (2 minutes)
• Improvements on C for low-level programming such as block arguments (p. 584) 02021-07-14 (updated 02021-12-30) (8 minutes)
• Compilation of block arguments to high-performance code (p. 624) 02021-07-29 (updated 02021-12-30) (19 minutes)

Notes concerning “Math”

• Fibonacci scan (p. 31) 02021-01-10 (updated 02021-01-15) (1 minute)
• Some notes on compiling and notations for grammars, starting from the inspiring RPN example in Parson (p. 57) 02021-01-15 (updated 02021-12-31) (15 minutes)
• Threechet (p. 140) 02021-02-16 (updated 02021-02-24) (4 minutes)

• Recursive residue number systems? (p. 259) 02021-03-20 (updated 02021-03-22) (8 minutes)
• The nature of mathematical discourse (p. 418) 02021-05-27 (updated 02021-12-30) (5 minutes)
• Minkowski deconvolution (p. 428) 02021-06-02 (updated 02021-12-30) (6 minutes)
• The algebra of N-ary relations (p. 432) 02021-06-14 (updated 02021-07-27) (4 minutes)
• Base 3 gage blocks (p. 468) 02021-06-27 (updated 02021-12-30) (5 minutes)
• Binomial coefficients and the dimensionality of spaces of polynomials (p. 957) 02021-10-20 (updated 02021-12-30) (4 minutes)

• Finite element analysis with sparse approximations (p. 959) 02021-10-20 (updated 02021-12-30) (2 minutes)
• Orthogonal rational vectors (p. 997) 02021-11-04 (updated 02021-12-30) (4 minutes)

Notes concerning "Lisp"

• Layout typescript (p. 29) 02021-01-04 (5 minutes)
• Transactional editor (p. 35) 02021-01-14 (updated 02021-01-15) (73 minutes)
• How do you fit a high-level language into a microcontroller? Let's look at BBN Lisp (p. 160) 02021-02-23 (updated 02021-08-18) (76 minutes)
• Some notes on IPL-VI, Lisp's 01958 precursor (p. 196) 02021-03-02 (4 minutes)
• Vaughan Pratt and Henry Baker's COMFY control-flow combinators (p. 234) 02021-03-04 (updated 02021-03-20) (8 minutes)
• Garbage-collected allocation performance on current computers (p. 245) 02021-03-13 (updated 02021-04-08) (4 minutes)
• Open coded primitives (p. 283) 02021-03-22 (26 minutes)
• Minor improvements to pattern matching (p. 306) 02021-03-24 (updated 02021-04-08) (10 minutes)
• Smolsay: the Ur-Lisp, but with dicts instead of conses (p. 552) 02021-07-12 (updated 02021-07-27) (22 minutes)
• A compact bytecode sketch that should average about 3 bytes per line of C (p. 720) 02021-08-17 (updated 02021-09-13) (66 minutes)
• Qfitzah: a minimal term-rewriting language (p. 809) 02021-09-10 (updated 02021-12-31) (62 minutes)

Notes concerning “Assembly-language programming”

• Compiling machine-code loops to pipelined dataflow graphs (p. 81) 02021-01-23 (updated 02021-01-27) (2 minutes)
• Some preliminary notes on the amazing RISC-V architecture (p. 82) 02021-01-24 (updated 02021-07-27) (29 minutes)
• Variable length unaligned bytecode (p. 199) 02021-03-02 (updated 02021-03-03) (4 minutes)
• Open coded primitives (p. 283) 02021-03-22 (26 minutes)
• Manually writing code in static single assignment (SSA) form, inspired by Kemeny’s DOPE, isn’t worth it (p. 353) 02021-04-21 (updated 02021-06-12) (3 minutes)
• Faygoo: a yantra-smashing ersatz version of Piumarta and Warth’s COLA (p. 570) 02021-07-14 (updated 02021-12-30) (17 minutes)
• Compilation of block arguments to high-performance code (p. 624) 02021-07-29 (updated 02021-12-30) (19 minutes)
• A compact bytecode sketch that should average about 3 bytes per line of C (p. 720) 02021-08-17 (updated 02021-09-13) (66 minutes)
• Qfitzah: a minimal term-rewriting language (p. 809) 02021-09-10 (updated 02021-12-31) (62 minutes)
• Fung’s “I can’t believe it can sort” algorithm and others (p. 864) 02021-10-05 (updated 02021-12-30) (5 minutes)
• The astounding UI responsivity of PDP-10 DDT on ITS (p. 972) 02021-10-22 (updated 02021-10-23) (28 minutes)

Notes concerning “Power supplies”

• Notes on simulating a ZVS converter (Baxandall converter) (p. 70) 02021-01-16 (6 minutes)
• A ghetto linear voltage regulator from discrete components (p. 73) 02021-01-21 (updated 02021-01-27) (10 minutes)
• Trying to design a simple switchmode power supply using Schmitt-trigger relaxation oscillators (p. 92) 02021-01-26 (updated 02021-01-27) (32 minutes)
• Bench supply (p. 250) 02021-03-19 (updated 02021-12-30) (25 minutes)
• Arc maker (p. 695) 02021-08-07 (updated 02021-12-30) (11 minutes)
• Power transistors (p. 700) 02021-08-07 (updated 02021-12-30) (12 minutes)
• Constant current buck (p. 708) 02021-08-10 (updated 02021-08-15) (4 minutes)
• Switching kiloamps in microseconds (p. 804) 02021-09-09 (updated 02021-12-30) (1 minute)
• Ranking MOSFETs for, say, rapid localized electrolysis to make optics (p. 938) 02021-10-11 (updated 02021-12-30) (8 minutes)
• An even simpler offline power supply than a capacitive dropper, with a 7¢ BOM (p. 943) 02021-10-14 (updated 02021-12-30) (7 minutes)

Notes concerning "Graphics"

• ASCII art, but in Unicode, with Braille and other alternatives (p. 128) 02021-02-10 (updated 02021-02-24) (9 minutes)
• Skew tilesets (p. 135) 02021-02-14 (updated 02021-02-24) (7 minutes)
• Threechet (p. 140) 02021-02-16 (updated 02021-02-24) (4 minutes)

• Thumbnail views in a Unicode character-cell terminal with Braille (p. 142) 02021-02-17 (updated 02021-02-24) (1 minute)
• Beyond overstrike (p. 922) 02021-10-10 (updated 02021-12-30) (13 minutes)
• Pipelined piece chain painting (p. 926) 02021-10-10 (updated 02021-12-30) (23 minutes)
• Constant weight dithering (p. 991) 02021-10-28 (updated 02021-12-30) (5 minutes)
• Paeth prediction and vector quantization (p. 1005) 02021-11-06 (updated 02021-12-30) (1 minute)
• Rendering 3-D graphics with PINNs and GANs? (p. 1010) 02021-11-11 (updated 02021-12-30) (10 minutes)
• A simple 2-D programmable graphics pipeline to unify tiles and palettes (p. 1022) 02021-11-18 (updated 02021-12-30) (6 minutes)

Notes concerning "Compilers"

• Principled APL redux (p. 22) 02021-01-03 (updated 02021-12-31) (12 minutes)
• Trie PEGs (p. 53) 02021-01-15 (4 minutes)
• Some notes on compiling and notations for grammars, starting from the inspiring RPN example in Parson (p. 57) 02021-01-15 (updated 02021-12-31) (15 minutes)
• Vaughan Pratt and Henry Baker's COMFY control-flow combinators (p. 234) 02021-03-04 (updated 02021-03-20) (8 minutes)
• Some notes on reading Chris Seaton's TruffleRuby dissertation (p. 269) 02021-03-21 (updated 02021-03-22) (16 minutes)
• Open coded primitives (p. 283) 02021-03-22 (26 minutes)
• Faygoo: a yantra-smashing ersatz version of Piumarta and Warth's COLA (p. 570) 02021-07-14 (updated 02021-12-30) (17 minutes)
• Compilation of block arguments to high-performance code (p. 624) 02021-07-29 (updated 02021-12-30) (19 minutes)
• Subset of C for the simplest self-compiling compiler (p. 717) 02021-08-12 (updated 02021-12-30) (6 minutes)
• A compact bytecode sketch that should average about 3 bytes per line of C (p. 720) 02021-08-17 (updated 02021-09-13) (66 minutes)

Notes concerning “Clay”

- Thixotropic electrodeposition (p. 372) 02021-05-04 (updated 02021-12-31) (2 minutes)
- Cheap cutting jig (p. 373) 02021-05-06 (updated 02021-12-30) (1 minute)
- Clay-filled PLA filament for firing to ceramic (p. 396) 02021-05-17 (updated 02021-12-30) (1 minute)
- Cutting clay (p. 400) 02021-05-19 (updated 02021-12-30) (10 minutes)
- Clay wire cutter (p. 409) 02021-05-21 (updated 02021-12-30) (2 minutes)
- Super ghetto digital fabrication (p. 479) 02021-06-28 (updated 02021-12-30) (33 minutes)
- Spin-coating clay-filled plastics to make composites with high anisotropic filler loadings (p. 521) 02021-07-02 (updated 02021-12-30) (4 minutes)
- Leaf vein roof (p. 600) 02021-07-16 (updated 02021-09-11) (9 minutes)
- The ayurvedic “fire mud” of Bhudeb Mookerji and modern castable refractories (p. 688) 02021-08-05 (updated 02021-08-15) (22 minutes)
- Regenerative muffle kiln (p. 1108) 02021-12-21 (updated 02021-12-30) (19 minutes)

Notes concerning “Aluminum”

• Fresnel mirror electropolishing (p. 377) 02021-05-08 (updated 02021-12-30) (6 minutes)
• Metal welding fuel (p. 411) 02021-05-23 (updated 02021-12-30) (6 minutes)
• Aluminum foil (p. 413) 02021-05-24 (updated 02021-09-11) (14 minutes)
• Aluminum fuel (p. 603) 02021-07-17 (updated 02021-12-30) (2 minutes)
• Material observations (p. 633) 02021-07-29 (updated 02021-12-30) (194 minutes)
• Maximizing phosphate density from aqueous reaction (p. 757) 02021-08-21 (updated 02021-12-30) (8 minutes)
• An aluminum pencil for marking iron? (p. 1001) 02021-11-06 (updated 02021-12-30) (2 minutes)
• At small scales, electrowinning may be cheaper than smelting (p. 1029) 02021-11-21 (updated 02021-12-30) (25 minutes)
• Electrolytic 2-D cutting and related electrolytic digital fabrication processes (p. 1085) 02021-12-16 (updated 02021-12-30) (48 minutes)
• Aluminum refining (p. 1106) 02021-12-20 (updated 02021-12-30) (3 minutes)

Notes concerning “Welding”

• Electroforming rivets (p. 410) 02021-05-22 (updated 02021-12-30) (2 minutes)
• Metal welding fuel (p. 411) 02021-05-23 (updated 02021-12-30) (6 minutes)
• Super ghetto digital fabrication (p. 479) 02021-06-28 (updated 02021-12-30) (33 minutes)
• Tetrahedral expanded metal (p. 593) 02021-07-16 (updated 02021-07-27) (3 minutes)
• Arc maker (p. 695) 02021-08-07 (updated 02021-12-30) (11 minutes)
• Electrodeposition welding (p. 769) 02021-08-25 (updated 02021-09-11) (2 minutes)
• Negative feedback control to prevent runaway positive feedback in 3-D MIG welding printing (p. 777) 02021-08-30 (updated 02021-12-30) (3 minutes)
• Spot welding (p. 805) 02021-09-09 (updated 02021-12-30) (8 minutes)
• Layers plus electroforming (p. 1100) 02021-12-16 (updated 02021-12-30) (7 minutes)

Notes concerning “Virtual machines”

• How do you fit a high-level language into a microcontroller? Let’s look at BBN Lisp (p. 160) 02021-02-23 (updated 02021-08-18) (76 minutes)
• Variable length unaligned bytecode (p. 199) 02021-03-02 (updated 02021-03-03) (4 minutes)
• A survey of imperative programming operations’ prevalence (p. 201) 02021-03-02 (updated 02021-09-11) (61 minutes)
• Veskeno is a “fantasy platform” like TIC-80 (p. 267) 02021-03-21 (updated 02021-03-22) (3 minutes)
• Some notes on reading Chris Seaton’s TruffleRuby dissertation (p. 269) 02021-03-21 (updated 02021-03-22) (16 minutes)
• .xosm: experimental obvious stack machine (p. 274) 02021-03-21 (updated 02021-03-24) (20 minutes)
• Leaf hypertext (p. 380) 02021-05-08 (updated 02021-12-30) (3 minutes)
• Smolsay: the Ur-Lisp, but with dicts instead of conses (p. 552) 02021-07-12 (updated 02021-07-27) (22 minutes)
• A compact bytecode sketch that should average about 3 bytes per line of C (p. 720) 02021-08-17 (updated 02021-09-13) (66 minutes)

Notes concerning "Precision"

• Weighing an eyelash on an improvised Kibble balance (p. 382) 02021-05-08 (updated 02021-12-30) (3 minutes)
• Broken hard disks are the cheapest source of ultraprecision components (p. 425) 02021-06-02 (updated 02021-06-12) (3 minutes)

• Minkowski deconvolution (p. 428) 02021-06-02 (updated 02021-12-30) (6 minutes)
• Electrolytic glass machining (p. 477) 02021-06-28 (updated 02021-12-30) (6 minutes)
• Super ghetto digital fabrication (p. 479) 02021-06-28 (updated 02021-12-30) (33 minutes)
• Weighing balance design (p. 799) 02021-09-06 (updated 02021-12-30) (9 minutes)
• Wire brush microscope (p. 1006) 02021-11-06 (updated 02021-12-30) (1 minute)
• Vernier indicator (p. 1040) 02021-11-22 (updated 02021-12-30) (6 minutes)
• Capacitive linear encoder sensors (p. 1056) 02021-12-11 (updated 02021-12-30) (7 minutes)

Notes concerning “Phosphates”

• More cements (p. 466) 02021-06-26 (updated 02021-08-15) (5 minutes)
• Electrolytic berlinite (p. 561) 02021-07-12 (updated 02021-12-30) (7 minutes)
• SHS of magnesium phosphate (p. 608) 02021-07-22 (updated 02021-07-27) (3 minutes)
• Material observations (p. 633) 02021-07-29 (updated 02021-12-30) (194 minutes)
• The ayurvedic “fire mud” of Bhudeb Mookerji and modern castable refractories (p. 688) 02021-08-05 (updated 02021-08-15) (22 minutes)
• Cola flavor (p. 707) 02021-08-10 (updated 02021-08-15) (2 minutes)
• Maximizing phosphate density from aqueous reaction (p. 757) 02021-08-21 (updated 02021-12-30) (8 minutes)
• Fast electrolytic mineral accretion (seacrete) for digital fabrication? (p. 779) 02021-09-01 (updated 02021-12-30) (52 minutes)
• Solid rock on a gossamer skeleton through exponential deposition (p. 1076) 02021-12-15 (updated 02021-12-30) (11 minutes)

Notes concerning “Foam”

- Clay wire cutter (p. 409) 02021-05-21 (updated 02021-12-30) (2 minutes)
- More cements (p. 466) 02021-06-26 (updated 02021-08-15) (5 minutes)
- Super ghetto digital fabrication (p. 479) 02021-06-28 (updated 02021-12-30) (33 minutes)
- Glass foam (p. 595) 02021-07-16 (updated 02021-08-15) (17 minutes)
- Material observations (p. 633) 02021-07-29 (updated 02021-12-30) (194 minutes)
- Sandwich panel optimization (p. 754) 02021-08-21 (updated 02021-09-11) (3 minutes)
- Glass wood (p. 755) 02021-08-21 (updated 02021-12-30) (4 minutes)
- Blowing agents (p. 847) 02021-09-29 (updated 02021-12-30) (4 minutes)
- Solid rock on a gossamer skeleton through exponential deposition (p. 1076) 02021-12-15 (updated 02021-12-30) (11 minutes)

Notes concerning “ECM”

• Fresnel mirror electropolishing (p. 377) 02021-05-08 (updated 02021-12-30) (6 minutes)
• Aluminum foil (p. 413) 02021-05-24 (updated 02021-09-11) (14 minutes)
• Layered ECM (p. 473) 02021-06-27 (updated 02021-12-30) (2 minutes)
• Can you use stabilized cubic zirconia as an ECM cathode in molten salt? (p. 475) 02021-06-27 (updated 02021-12-30) (3 minutes)
• Electrolytic glass machining (p. 477) 02021-06-28 (updated 02021-12-30) (6 minutes)
• ECM for machining nonmetals? (p. 523) 02021-07-05 (updated 02021-07-27) (11 minutes)
• Fast-slicing ECM (p. 802) 02021-09-08 (updated 02021-12-30) (3 minutes)
• Some notes on Bhattacharyya’s ECM book (p. 1043) 02021-11-25 (updated 02021-12-30) (11 minutes)
• Layers plus electroforming (p. 1100) 02021-12-16 (updated 02021-12-30) (7 minutes)

Notes concerning “Composites”

• Stochastically generated self-amalgamating tape variations for composite fabrication (p. 510) 02021-07-02 (updated 02021-12-30) (26 minutes)
• Fiberglass CMCs? (p. 588) 02021-07-15 (updated 02021-07-27) (8 minutes)
• Leaf vein roof (p. 600) 02021-07-16 (updated 02021-09-11) (9 minutes)
• Sandwich panel optimization (p. 754) 02021-08-21 (updated 02021-09-11) (3 minutes)
• Glass wood (p. 755) 02021-08-21 (updated 02021-12-30) (4 minutes)
• Maximizing phosphate density from aqueous reaction (p. 757) 02021-08-21 (updated 02021-12-30) (8 minutes)
• Fast electrolytic mineral accretion (seacrete) for digital fabrication? (p. 779) 02021-09-01 (updated 02021-12-30) (52 minutes)
• Rock-wool-filled composites (p. 798) 02021-09-03 (updated 02021-12-30) (2 minutes)
• Electrolytic 2-D cutting and related electrolytic digital fabrication processes (p. 1085) 02021-12-16 (updated 02021-12-30) (48 minutes)

Notes concerning “Composability”

• Skew tilesets (p. 135) 02021-02-14 (updated 02021-02-24) (7 minutes)
• Threechet (p. 140) 02021-02-16 (updated 02021-02-24) (4 minutes)

• Panelization in PCB manufacturing (p. 193) 02021-02-25 (updated 02021-02-26) (7 minutes)
• Base 3 gage blocks (p. 468) 02021-06-27 (updated 02021-12-30) (5 minutes)
• Rator-port GUIs (p. 496) 02021-06-29 (updated 02021-12-30) (26 minutes)
• A construction set using SHS (p. 765) 02021-08-24 (updated 02021-09-11) (5 minutes)
• An algebra of partial functions for interactively composing programs (p. 933) 02021-10-10 (updated 02021-12-30) (3 minutes)
• Two finger multitouch (p. 1059) 02021-12-11 (updated 02021-12-30) (3 minutes)
• The Habitaculum: a modular dwelling machine (p. 1061) 02021-12-13 (updated 02021-12-31) (16 minutes)

Notes concerning “Waterglass”

• Verstickulite (p. 457) 02021-06-23 (updated 02021-07-27) (3 minutes)
• Powder-bed 3-D printing with a sacrificial binder (p. 506) 02021-06-30 (updated 02021-12-30) (12 minutes)
• Potential local sources and prices of refractory materials (p. 566) 02021-07-14 (updated 02021-09-11) (9 minutes)
• Glass foam (p. 595) 02021-07-16 (updated 02021-08-15) (17 minutes)
• Material observations (p. 633) 02021-07-29 (updated 02021-12-30) (194 minutes)
• Waterglass “Loctite”? (p. 845) 02021-09-22 (updated 02021-12-30) (1 minute)
• Solid rock on a gossamer skeleton through exponential deposition (p. 1076) 02021-12-15 (updated 02021-12-30) (11 minutes)
• Xerogel compacting (p. 1119) 02021-12-22 (updated 02021-12-30) (12 minutes)

Notes concerning “Small is beautiful”

- Chat over a content-centric network (p. 55) 02021-01-15 (updated 02021-01-16) (3 minutes)
- How little code can a filesystem be? (p. 438) 02021-06-16 (updated 02021-07-27) (1 minute)
- Self hosting kernel (p. 452) 02021-06-21 (updated 02021-12-30) (1 minute)
- Smolsay: the Ur-Lisp, but with dicts instead of conses (p. 552) 02021-07-12 (updated 02021-07-27) (22 minutes)
- Faygoo: a yantra-smashing ersatz version of Piumarta and Warth’s COLA (p. 570) 02021-07-14 (updated 02021-12-30) (17 minutes)
- Subset of C for the simplest self-compiling compiler (p. 717) 02021-08-12 (updated 02021-12-30) (6 minutes)
- Qfitzah: a minimal term-rewriting language (p. 809) 02021-09-10 (updated 02021-12-31) (62 minutes)
- Fung’s “I can’t believe it can sort” algorithm and others (p. 864) 02021-10-05 (updated 02021-12-30) (5 minutes)

Notes concerning "Sensors"

• Phased-array imaging sonar from a mesh network of self-localizing sensor nodes (p. 358) 02021-04-27 (updated 02021-12-30) (8 minutes)

• Differential filming (p. 374) 02021-05-07 (updated 02021-12-30) (1 minute)
• Minkowski deconvolution (p. 428) 02021-06-02 (updated 02021-12-30) (6 minutes)
• Compliance spectroscopy (p. 849) 02021-09-29 (updated 02021-12-30) (4 minutes)
• Viscoelastic probing (p. 1000) 02021-11-04 (updated 02021-12-30) (2 minutes)
• Wire brush microscope (p. 1006) 02021-11-06 (updated 02021-12-30) (1 minute)
• Aqueous scanning probe microscopy (p. 1013) 02021-11-12 (updated 02021-12-30) (7 minutes)
• MOSFET body diodes as Geiger counter avalanche detectors? (p. 1103) 02021-12-17 (updated 02021-12-30) (1 minute)

Notes concerning "Programming languages"

• Principled APL redux (p. 22) 02021-01-03 (updated 02021-12-31) (12 minutes)
• First class locations (p. 27) 02021-01-04 (3 minutes)
• Minor improvements to pattern matching (p. 306) 02021-03-24 (updated 02021-04-08) (10 minutes)
• :fqozl, a normal-order text macro language (p. 336) 02021-04-09 (updated 02021-07-27) (14 minutes)
• Safe FORTH with the FORTRAN memory model? (p. 351) 02021-04-21 (updated 02021-06-12) (2 minutes)
• Smolsay: the Ur-Lisp, but with dicts instead of conses (p. 552) 02021-07-12 (updated 02021-07-27) (22 minutes)
• Qfitzah: a minimal term-rewriting language (p. 809) 02021-09-10 (updated 02021-12-31) (62 minutes)
• An algebra of partial functions for interactively composing programs (p. 933) 02021-10-10 (updated 02021-12-30) (3 minutes)

Notes concerning “Ceramic”

• Thixotropic electrodeposition (p. 372) 02021-05-04 (updated 02021-12-31) (2 minutes)
• Cheap cutting jig (p. 373) 02021-05-06 (updated 02021-12-30) (1 minute)
• Clay-filled PLA filament for firing to ceramic (p. 396) 02021-05-17 (updated 02021-12-30) (1 minute)
• Super ghetto digital fabrication (p. 479) 02021-06-28 (updated 02021-12-30) (33 minutes)
• Firing talc (p. 576) 02021-07-14 (updated 02021-12-30) (17 minutes)
• Leaf vein roof (p. 600) 02021-07-16 (updated 02021-09-11) (9 minutes)
• The ayurvedic “fire mud” of Bhudeb Mookerji and modern castable refractories (p. 688) 02021-08-05 (updated 02021-08-15) (22 minutes)

• Regenerative muffle kiln (p. 1108) 02021-12-21 (updated 02021-12-30) (19 minutes)

Notes concerning “C”

• First class locations (p. 27) 02021-01-04 (3 minutes)
• Chat over a content-centric network (p. 55) 02021-01-15 (updated 02021-01-16) (3 minutes)
• A survey of imperative programming operations’ prevalence (p. 201) 02021-03-02 (updated 02021-09-11) (61 minutes)
• Improvements on C for low-level programming such as block arguments (p. 584) 02021-07-14 (updated 02021-12-30) (8 minutes)
• Subset of C for the simplest self-compiling compiler (p. 717) 02021-08-12 (updated 02021-12-30) (6 minutes)
• A compact bytecode sketch that should average about 3 bytes per line of C (p. 720) 02021-08-17 (updated 02021-09-13) (66 minutes)
• Some notes on perusing the Udanax Green codebase (p. 860) 02021-10-05 (updated 02021-10-08) (12 minutes)
• Fung’s “I can’t believe it can sort” algorithm and others (p. 864) 02021-10-05 (updated 02021-12-30) (5 minutes)

Notes concerning “Real time”

• Relayout with heaps (p. 32) 02021-01-10 (updated 02021-01-15) (6 minutes)
• Garbage-collected allocation performance on current computers (p. 245) 02021-03-13 (updated 02021-04-08) (4 minutes)
• Running scripts once per frame for guaranteed GUI responsivity (p. 303) 02021-03-23 (updated 02021-10-12) (7 minutes)
• Notes on what would be needed to drive a PS/2 keyboard from an Arduino (p. 447) 02021-06-20 (updated 02021-12-30) (12 minutes)
• Rator-port GUIs (p. 496) 02021-06-29 (updated 02021-12-30) (26 minutes)
• Ropes with constant-time concatenation and equality comparisons with monoidal hash consing (p. 619) 02021-07-27 (15 minutes)
• Lazy heapsort (p. 761) 02021-08-22 (updated 02021-09-11) (6 minutes)

Notes concerning “Higher order programming”

• Leaf hypertext (p. 380) 02021-05-08 (updated 02021-12-30) (3 minutes)
• Rator-port GUIs (p. 496) 02021-06-29 (updated 02021-12-30) (26 minutes)
• Smolsay: the Ur-Lisp, but with dicts instead of conses (p. 552) 02021-07-12 (updated 02021-07-27) (22 minutes)
• Faygoo: a yantra-smashing ersatz version of Piumarta and Warth’s COLA (p. 570) 02021-07-14 (updated 02021-12-30) (17 minutes)
• Improvements on C for low-level programming such as block arguments (p. 584) 02021-07-14 (updated 02021-12-30) (8 minutes)
• Compilation of block arguments to high-performance code (p. 624) 02021-07-29 (updated 02021-12-30) (19 minutes)
• Qfitzah: a minimal term-rewriting language (p. 809) 02021-09-10 (updated 02021-12-31) (62 minutes)

Notes concerning “Hand tools”

• Micro impact driver (p. 427) 02021-06-02 (updated 02021-06-12) (2 minutes)
• Sonic screwdriver resonance (p. 527) 02021-07-06 (updated 02021-12-30) (11 minutes)
• Spreadtool (p. 543) 02021-07-10 (updated 02021-12-30) (30 minutes)
• Thread rolling roller screw (p. 999) 02021-11-04 (updated 02021-12-30) (1 minute)
• An aluminum pencil for marking iron? (p. 1001) 02021-11-06 (updated 02021-12-30) (2 minutes)
• Vernier indicator (p. 1040) 02021-11-22 (updated 02021-12-30) (6 minutes)
• Toggling eccentrics for removing preload from spring clamps (p. 1129) 02021-12-28 (updated 02021-12-31) (22 minutes)

Notes concerning “Falstad’s circuit simulator”

• Notes on simulating a ZVS converter (Baxandall converter) (p. 70) 02021-01-16 (6 minutes)
• A ghetto linear voltage regulator from discrete components (p. 73) 02021-01-21 (updated 02021-01-27) (10 minutes)
• Trying to design a simple switchmode power supply using Schmitt-trigger relaxation oscillators (p. 92) 02021-01-26 (updated 02021-01-27) (32 minutes)
• Snap logic, revisited, and four-phase logic (p. 115) 02021-02-08 (9 minutes)
• Bench supply (p. 250) 02021-03-19 (updated 02021-12-30) (25 minutes)
• Failing to stabilize the amplitude of an opamp phase-delay oscillator (p. 298) 02021-03-23 (updated 02021-03-24) (10 minutes)
• Three phase logic (p. 364) 02021-04-30 (updated 02021-07-27) (9 minutes)

Notes concerning “Facepalm”

• Trying and failing to design an efficient index for folksonomy data based on BDDs (p. 108) 02021-01-26 (updated 02021-01-27) (7 minutes)
• Garbage-collected allocation performance on current computers (p. 245) 02021-03-13 (updated 02021-04-08) (4 minutes)
• Recursive residue number systems? (p. 259) 02021-03-20 (updated 02021-03-22) (8 minutes)
• Failing to stabilize the amplitude of an opamp phase-delay oscillator (p. 298) 02021-03-23 (updated 02021-03-24) (10 minutes)
• Manually writing code in static single assignment (SSA) form, inspired by Kemeny’s DOPE, isn’t worth it (p. 353) 02021-04-21 (updated 02021-06-12) (3 minutes)
• Simple linear-time linear-space nested delimiter parsing (p. 459) 02021-06-24 (updated 02021-12-30) (1 minute)
• Micro ramjet (p. 1038) 02021-11-22 (updated 02021-12-30) (3 minutes)

Notes concerning “Argentina”

• Why Bitcoin is puzzling to people in rich countries (p. 312) 02021-03-31 (updated 02021-07-27) (10 minutes)
• Notes on pricing of locally available oscilloscopes (p. 346) 02021-04-16 (updated 02021-07-27) (2 minutes)
• Spreadtool (p. 543) 02021-07-10 (updated 02021-12-30) (30 minutes)
• Potential local sources and prices of refractory materials (p. 566) 02021-07-14 (updated 02021-09-11) (9 minutes)
• Material observations (p. 633) 02021-07-29 (updated 02021-12-30) (194 minutes)
• Argentine pricing of PEX pipe and alternatives for phase-change fluids (p. 699) 02021-08-07 (updated 02021-12-30) (2 minutes)
• Dense fillers (p. 772) 02021-08-25 (updated 02021-12-30) (7 minutes)

Notes concerning “2-D cutting”

• Cheap cutting jig (p. 373) 02021-05-06 (updated 02021-12-30) (1 minute)
• Cutting clay (p. 400) 02021-05-19 (updated 02021-12-30) (10 minutes)
• Spreadtool (p. 543) 02021-07-10 (updated 02021-12-30) (30 minutes)
• Trying to quantify relative speeds of different digital fabrication processes with “matter bandwidth” (p. 946) 02021-10-15 (updated 02021-12-30) (5 minutes)
• Electrolytic 2-D cutting and related electrolytic digital fabrication processes (p. 1085) 02021-12-16 (updated 02021-12-30) (48 minutes)
• Layers plus electroforming (p. 1100) 02021-12-16 (updated 02021-12-30) (7 minutes)
• Toggling eccentrics for removing preload from spring clamps (p. 1129) 02021-12-28 (updated 02021-12-31) (22 minutes)

Notes concerning “Terminals”

• First class locations (p. 27) 02021-01-04 (3 minutes)
• Layout typescript (p. 29) 02021-01-04 (5 minutes)
• ASCII art, but in Unicode, with Braille and other alternatives (p. 128) 02021-02-10 (updated 02021-02-24) (9 minutes)
• Beyond overstrike (p. 922) 02021-10-10 (updated 02021-12-30) (13 minutes)
• Pipelined piece chain painting (p. 926) 02021-10-10 (updated 02021-12-30) (23 minutes)
• Beyond op streams (p. 935) 02021-10-11 (updated 02021-12-30) (3 minutes)

Notes concerning “Solar”

• The use of silver in solar cells (p. 112) 02021-02-02 (updated 02021-09-11) (8 minutes)
• Energy autonomous computing (p. 143) 02021-02-18 (updated 02021-12-30) (58 minutes)
• Statistics on the present and future of energy in the People’s Republic of China (p. 316) 02021-04-01 (updated 02021-04-08) (10 minutes)
• Aluminum foil (p. 413) 02021-05-24 (updated 02021-09-11) (14 minutes)
• The relation between solar-panel efficiency for air conditioning and insulation thickness (p. 941) 02021-10-11 (updated 02021-12-30) (3 minutes)
• Photoemissive power (p. 1124) 02021-12-23 (updated 02021-12-28) (15 minutes)

Notes concerning “Self replication”

• Forming steel with copper instead of vice versa (p. 344) 02021-04-16 (updated 02021-06-12) (2 minutes)
• How fast do von Neumann probes need to reproduce to colonize space in our lifetimes? (p. 368) 02021-05-04 (updated 02021-06-12) (5 minutes)
• Cutting clay (p. 400) 02021-05-19 (updated 02021-12-30) (10 minutes)
• Firing talc (p. 576) 02021-07-14 (updated 02021-12-30) (17 minutes)
• Trying to quantify relative speeds of different digital fabrication processes with “matter bandwidth” (p. 946) 02021-10-15 (updated 02021-12-30) (5 minutes)
• Redundancy in self-replicating systems such as hundred-eyed chickens (p. 1016) 02021-11-12 (updated 02021-12-30) (4 minutes)

Notes concerning “Systems architecture”

- Energy autonomous computing (p. 143) 02021-02-18 (updated 02021-12-30) (58 minutes)
- A kernel you can type commands to (p. 474) 02021-06-27 (updated 02021-12-30) (1 minute)
- Smolsay: the Ur-Lisp, but with dicts instead of conses (p. 552) 02021-07-12 (updated 02021-07-27) (22 minutes)
- Faygoo: a yantra-smashing ersatz version of Piumarta and Warth’s COLA (p. 570) 02021-07-14 (updated 02021-12-30) (17 minutes)
- A simple 2-D programmable graphics pipeline to unify tiles and palettes (p. 1022) 02021-11-18 (updated 02021-12-30) (6 minutes)
- Safe decentralized cloud storage (p. 1135) 02021-12-30 (10 minutes)

Notes concerning "Protocols"

• Notes on what would be needed to drive a PS/2 keyboard from an Arduino (p. 447) 02021-06-20 (updated 02021-12-30) (12 minutes)
• Does USB bitstuffing create a timing-channel vulnerability? (p. 456) 02021-06-22 (updated 02021-12-31) (1 minute)
• Residual stream windowing (p. 752) 02021-08-21 (updated 02021-09-11) (5 minutes)
• Three phase differential data (p. 843) 02021-09-22 (updated 02021-12-30) (4 minutes)
• Pipelined piece chain painting (p. 926) 02021-10-10 (updated 02021-12-30) (23 minutes)
• Safe decentralized cloud storage (p. 1135) 02021-12-30 (10 minutes)

Notes concerning “Post-teletype terminal design”

• First class locations (p. 27) 02021-01-04 (3 minutes)
• Layout typescript (p. 29) 02021-01-04 (5 minutes)
• A kernel you can type commands to (p. 474) 02021-06-27 (updated 02021-12-30) (1 minute)
• Beyond overstrike (p. 922) 02021-10-10 (updated 02021-12-30) (13 minutes)
• Pipelined piece chain painting (p. 926) 02021-10-10 (updated 02021-12-30) (23 minutes)
• Beyond op streams (p. 935) 02021-10-11 (updated 02021-12-30) (3 minutes)

Notes concerning “Physical computation”

• Snap logic, revisited, and four-phase logic (p. 115) 02021-02-08 (9 minutes)
• When is it better to compute by moving atoms rather than electrons? (p. 265) 02021-03-21 (updated 02021-03-22) (5 minutes)
• Geneva wheel stopwork (p. 321) 02021-04-07 (updated 02021-04-08) (6 minutes)
• Three phase logic (p. 364) 02021-04-30 (updated 02021-07-27) (9 minutes)
• Three phase differential data (p. 843) 02021-09-22 (updated 02021-12-30) (4 minutes)
• Simplest blinker (p. 1053) 02021-12-01 (updated 02021-12-30) (9 minutes)

Notes concerning “Optics”

• Fresnel mirror electropolishing (p. 377) 02021-05-08 (updated 02021-12-30) (6 minutes)
• Aluminum foil (p. 413) 02021-05-24 (updated 02021-09-11) (14 minutes)
• Subnanosecond thermochromic light modulation for real-time holography and displays (p. 531) 02021-07-06 (updated 02021-12-30) (8 minutes)
• Ranking MOSFETs for, say, rapid localized electrolysis to make optics (p. 938) 02021-10-11 (updated 02021-12-30) (8 minutes)
• Rendering 3-D graphics with PINNs and GANs? (p. 1010) 02021-11-11 (updated 02021-12-30) (10 minutes)
• Ghetto electrochromic displays for ultra-low-power computing? (p. 1082) 02021-12-16 (updated 02021-12-30) (9 minutes)

Notes concerning “Minerals”

- The use of silver in solar cells (p. 112) 02021-02-02 (updated 02021-09-11) (8 minutes)
- Electrolytic berlinite (p. 561) 02021-07-12 (updated 02021-12-30) (7 minutes)
- Making mirabilite and calcite from drywall (p. 564) 02021-07-12 (updated 02021-12-30) (4 minutes)
- Firing talc (p. 576) 02021-07-14 (updated 02021-12-30) (17 minutes)
- Synthesizing amorphous magnesium silicate (p. 617) 02021-07-25 (updated 02021-08-15) (6 minutes)
- At small scales, electrowinning may be cheaper than smelting (p. 1029) 02021-11-21 (updated 02021-12-30) (25 minutes)

Notes concerning “Microcontrollers”

• Can you do direct digital synthesis (DDS) at over a gigahertz? (p. 119) 02021-02-08 (updated 02021-02-24) (30 minutes)
• Energy autonomous computing (p. 143) 02021-02-18 (updated 02021-12-30) (58 minutes)
• How do you fit a high-level language into a microcontroller? Let’s look at BBN Lisp (p. 160) 02021-02-23 (updated 02021-08-18) (76 minutes)
• A bargain-basement Holter monitor with a BOM under US$2.50 (p. 323) 02021-04-07 (updated 02021-07-27) (33 minutes)
• Arc maker (p. 695) 02021-08-07 (updated 02021-12-30) (11 minutes)
• A compact bytecode sketch that should average about 3 bytes per line of C (p. 720) 02021-08-17 (updated 02021-09-13) (66 minutes)

Notes concerning “Metrology”

• iPhone replacement cameras as 6-μs streak cameras (p. 80) 02021-01-22 (updated 02021-12-30) (2 minutes)
• Phased-array imaging sonar from a mesh network of self-localizing sensor nodes (p. 358) 02021-04-27 (updated 02021-12-30) (8 minutes)

• Weighing an eyelash on an improvised Kibble balance (p. 382) 02021-05-08 (updated 02021-12-30) (3 minutes)
• Base 3 gage blocks (p. 468) 02021-06-27 (updated 02021-12-30) (5 minutes)
• Weighing balance design (p. 799) 02021-09-06 (updated 02021-12-30) (9 minutes)
• Vernier indicator (p. 1040) 02021-11-22 (updated 02021-12-30) (6 minutes)

Notes concerning "Magnesium"

• Metal welding fuel (p. 411) 02021-05-23 (updated 02021-12-30) (6 minutes)
• SHS of magnesium phosphate (p. 608) 02021-07-22 (updated 02021-07-27) (3 minutes)
• Synthesizing reactive magnesia? (p. 615) 02021-07-25 (updated 02021-08-15) (4 minutes)
• Synthesizing amorphous magnesium silicate (p. 617) 02021-07-25 (updated 02021-08-15) (6 minutes)
• Fast electrolytic mineral accretion (seacrete) for digital fabrication? (p. 779) 02021-09-01 (updated 02021-12-30) (52 minutes)
• Electrolytic 2-D cutting and related electrolytic digital fabrication processes (p. 1085) 02021-12-16 (updated 02021-12-30) (48 minutes)

Notes concerning “Instruction sets”

• Some preliminary notes on the amazing RISC-V architecture (p. 82) 02021-01-24 (updated 02021-07-27) (29 minutes)
• Variable length unaligned bytecode (p. 199) 02021-03-02 (updated 02021-03-03) (4 minutes)
• A survey of imperative programming operations’ prevalence (p. 201) 02021-03-02 (updated 02021-09-11) (61 minutes)
• Veskeno is a “fantasy platform” like TIC-80 (p. 267) 02021-03-21 (updated 02021-03-22) (3 minutes)
• .xosm: experimental obvious stack machine (p. 274) 02021-03-21 (updated 02021-03-24) (20 minutes)
• A compact bytecode sketch that should average about 3 bytes per line of C (p. 720) 02021-08-17 (updated 02021-09-13) (66 minutes)

Notes concerning "Independence"

- Energy autonomous computing (p. 143) 02021-02-18 (updated 02021-12-30) (58 minutes)
- Can you get JLCPCB to fabricate a CPU for you affordably from "basic" parts? (p. 347) 02021-04-17 (updated 02021-12-30) (9 minutes)
- How little code can a filesystem be? (p. 438) 02021-06-16 (updated 02021-07-27) (1 minute)
- Self hosting kernel (p. 452) 02021-06-21 (updated 02021-12-30) (1 minute)
- Aluminum fuel (p. 603) 02021-07-17 (updated 02021-12-30) (2 minutes)
- Trying to quantify relative speeds of different digital fabrication processes with "matter bandwidth" (p. 946) 02021-10-15 (updated 02021-12-30) (5 minutes)

Notes concerning “GUIs”

• Running scripts once per frame for guaranteed GUI responsivity (p. 303) 02021-03-23 (updated 02021-10-12) (7 minutes)
• List of random GUI ideas (p. 370) 02021-05-04 (updated 02021-07-27) (6 minutes)
• Leaf hypertext (p. 380) 02021-05-08 (updated 02021-12-30) (3 minutes)
• Bead hypertext (p. 455) 02021-06-22 (updated 02021-12-30) (1 minute)
• Rator-port GUIs (p. 496) 02021-06-29 (updated 02021-12-30) (26 minutes)
• Two finger multitouch (p. 1059) 02021-12-11 (updated 02021-12-30) (3 minutes)

Notes concerning "End user programming"

• First class locations (p. 27) 02021-01-04 (3 minutes)
• :fqozl, a normal-order text macro language (p. 336) 02021-04-09 (updated 02021-07-27) (14 minutes)
• Leaf hypertext (p. 380) 02021-05-08 (updated 02021-12-30) (3 minutes)
• Wiki models (p. 751) 02021-08-19 (updated 02021-12-30) (1 minute)
• The astounding UI responsivity of PDP-10 DDT on ITS (p. 972) 02021-10-22 (updated 02021-10-23) (28 minutes)
• Embedding runnable code in text paragraphs for numerical modeling (p. 1002) 02021-11-06 (updated 02021-12-30) (6 minutes)

Notes concerning “Anisotropic fillers”

- Acicular low binder pastes (p. 399) 02021-05-19 (updated 02021-12-30) (1 minute)
- Stochastically generated self-amalgamating tape variations for composite fabrication (p. 510) 02021-07-02 (updated 02021-12-30) (26 minutes)
- Spin-coating clay-filled plastics to make composites with high anisotropic filler loadings (p. 521) 02021-07-02 (updated 02021-12-30) (4 minutes)
- Glass wood (p. 755) 02021-08-21 (updated 02021-12-30) (4 minutes)
- Electrolytic 2-D cutting and related electrolytic digital fabrication processes (p. 1085) 02021-12-16 (updated 02021-12-30) (48 minutes)
- Xerogel compacting (p. 1119) 02021-12-22 (updated 02021-12-30) (12 minutes)

Notes concerning “Thermodynamics”

- A boiler for submillisecond steam pulses (p. 361) 02021-04-28 (updated 02021-12-30) (10 minutes)
- Pocket kiln (p. 704) 02021-08-09 (updated 02021-08-15) (7 minutes)
- Heating a shower tank with portable TCES? (p. 714) 02021-08-11 (updated 02021-08-15) (6 minutes)
- Spot welding (p. 805) 02021-09-09 (updated 02021-12-30) (8 minutes)
- Regenerative muffle kiln (p. 1108) 02021-12-21 (updated 02021-12-30) (19 minutes)

Notes concerning “The future”

• The use of silver in solar cells (p. 112) 02021-02-02 (updated 02021-09-11) (8 minutes)
• Statistics on the present and future of energy in the People’s Republic of China (p. 316) 02021-04-01 (updated 02021-04-08) (10 minutes)
• How fast do von Neumann probes need to reproduce to colonize space in our lifetimes? (p. 368) 02021-05-04 (updated 02021-06-12) (5 minutes)
• New nuclear power in the People’s Republic of China (p. 1007) 02021-11-09 (updated 02021-12-30) (2 minutes)
• Photoemissive power (p. 1124) 02021-12-23 (updated 02021-12-28) (15 minutes)

Notes concerning “Syntax”

• Some notes on compiling and notations for grammars, starting from the inspiring RPN example in Parson (p. 57) 02021-01-15 (updated 02021-12-31) (15 minutes)
• Minor improvements to pattern matching (p. 306) 02021-03-24 (updated 02021-04-08) (10 minutes)
• Stack syntax (p. 453) 02021-06-22 (updated 02021-07-27) (4 minutes)
• Smolsay: the Ur-Lisp, but with dicts instead of conses (p. 552) 02021-07-12 (updated 02021-07-27) (22 minutes)
• Qfitzah: a minimal term-rewriting language (p. 809) 02021-09-10 (updated 02021-12-31) (62 minutes)

Notes concerning “Steel”

• Forming steel with copper instead of vice versa (p. 344) 02021-04-16 (updated 02021-06-12) (2 minutes)
• Spreadtool (p. 543) 02021-07-10 (updated 02021-12-30) (30 minutes)
• An aluminum pencil for marking iron? (p. 1001) 02021-11-06 (updated 02021-12-30) (2 minutes)
• At small scales, electrowinning may be cheaper than smelting (p. 1029) 02021-11-21 (updated 02021-12-30) (25 minutes)
• Exotic steel analogues in other metals (p. 1050) 02021-12-01 (updated 02021-12-30) (8 minutes)

Notes concerning “Small things”

• Fresnel mirror electropolishing (p. 377) 02021-05-08 (updated 02021-12-30) (6 minutes)
• Weighing an eyelash on an improvised Kibble balance (p. 382) 02021-05-08 (updated 02021-12-30) (3 minutes)
• Scaling laws (p. 404) 02021-05-19 (updated 02021-12-30) (8 minutes)
• At small scales, electrowinning may be cheaper than smelting (p. 1029) 02021-11-21 (updated 02021-12-30) (25 minutes)
• Micro ramjet (p. 1038) 02021-11-22 (updated 02021-12-30) (3 minutes)

Notes concerning “Security”

• Does USB bitstuffing create a timing-channel vulnerability? (p. 456) 02021-06-22 (updated 02021-12-31) (1 minute)
• PBKDF content addressing with keyphrase hashcash: a non-blockchain attack on Zooko’s Triangle (p. 896) 02021-10-08 (24 minutes)
• Wordlists for maximum drama (p. 904) 02021-10-08 (updated 02021-12-30) (16 minutes)
• Constant weight dithering (p. 991) 02021-10-28 (updated 02021-12-30) (5 minutes)
• Safe decentralized cloud storage (p. 1135) 02021-12-30 (10 minutes)

Notes concerning “Refractory”

• Verstickulite (p. 457) 02021-06-23 (updated 02021-07-27) (3 minutes)
• Super ghetto digital fabrication (p. 479) 02021-06-28 (updated 02021-12-30) (33 minutes)
• Material observations (p. 633) 02021-07-29 (updated 02021-12-30) (194 minutes)
• The ayurvedic “fire mud” of Bhudeb Mookerji and modern castable refractories (p. 688) 02021-08-05 (updated 02021-08-15) (22 minutes)
• Pocket kiln (p. 704) 02021-08-09 (updated 02021-08-15) (7 minutes)

Notes concerning “Powder-bed 3-D printing processes”

• Precisely measuring out particulates with a trickler (p. 384) 02021-05-09 (updated 02021-12-30) (17 minutes)
• Acicular low binder pastes (p. 399) 02021-05-19 (updated 02021-12-30) (1 minute)
• Glass powder-bed 3-D printing (p. 490) 02021-06-29 (updated 02021-12-30) (20 minutes)
• Powder-bed 3-D printing with a sacrificial binder (p. 506) 02021-06-30 (updated 02021-12-30) (12 minutes)
• Selective laser sintering of copper (p. 775) 02021-08-30 (updated 02021-12-30) (6 minutes)

Notes concerning “The Portable Document Format (PDF)”

• PEG-like flexibility for parsing right-to-left? (p. 437) 02021-06-16 (updated 02021-07-27) (2 minutes)
• Notes on the PDF file format (p. 439) 02021-06-16 (updated 02021-07-27) (15 minutes)
• Bead hypertext (p. 455) 02021-06-22 (updated 02021-12-30) (1 minute)
• Memory view (p. 539) 02021-07-09 (updated 02021-12-30) (6 minutes)
• Compressed appendable file (p. 606) 02021-07-19 (updated 02021-07-27) (5 minutes)

Notes concerning “Parsing”

• Trie PEGs (p. 53) 02021-01-15 (4 minutes)
• Some notes on compiling and notations for grammars, starting from the inspiring RPN example in Parson (p. 57) 02021-01-15 (updated 02021-12-31) (15 minutes)
• PEG-like flexibility for parsing right-to-left? (p. 437) 02021-06-16 (updated 02021-07-27) (2 minutes)
• Simple linear-time linear-space nested delimiter parsing (p. 459) 02021-06-24 (updated 02021-12-30) (1 minute)
• Memory view (p. 539) 02021-07-09 (updated 02021-12-30) (6 minutes)

Notes concerning “Numerical modeling”

• Wiki models (p. 751) 02021-08-19 (updated 02021-12-30) (1 minute)
• Finite element analysis with sparse approximations (p. 959) 02021-10-20 (updated 02021-12-30) (2 minutes)
• Implementation and applications of low-voltage Marx generators with solid-state avalanche breakdown? (p. 960) 02021-10-20 (updated 02021-12-31) (39 minutes)
• Embedding runnable code in text paragraphs for numerical modeling (p. 1002) 02021-11-06 (updated 02021-12-30) (6 minutes)
• Rendering 3-D graphics with PINNs and GANs? (p. 1010) 02021-11-11 (updated 02021-12-30) (10 minutes)

Notes concerning “Incentives”

• Intel engineering positions considered as a dollar auction (p. 78) 02021-01-21 (updated 02021-01-27) (1 minute)
• Why Bitcoin is puzzling to people in rich countries (p. 312) 02021-03-31 (updated 02021-07-27) (10 minutes)
• Economic history (p. 460) 02021-06-25 (updated 02021-07-27) (17 minutes)
• Planning Apples to Apples, instead of Planning Poker (p. 851) 02021-09-29 (updated 02021-12-30) (6 minutes)
• Safe decentralized cloud storage (p. 1135) 02021-12-30 (10 minutes)

Notes concerning "FORTH"

• A survey of imperative programming operations' prevalence (p. 201) 02021-03-02 (updated 02021-09-11) (61 minutes)
• Safe FORTH with the FORTRAN memory model? (p. 351) 02021-04-21 (updated 02021-06-12) (2 minutes)
• Manually writing code in static single assignment (SSA) form, inspired by Kemeny's DOPE, isn't worth it (p. 353) 02021-04-21 (updated 02021-06-12) (3 minutes)
• Compilation of block arguments to high-performance code (p. 624) 02021-07-29 (updated 02021-12-30) (19 minutes)
• A compact bytecode sketch that should average about 3 bytes per line of C (p. 720) 02021-08-17 (updated 02021-09-13) (66 minutes)

Notes concerning "Flexures"

- Spreadtool (p. 543) 02021-07-10 (updated 02021-12-30) (30 minutes)
- Flexural mounts for self-aligning bushings (p. 952) 02021-10-18 (updated 02021-12-30) (3 minutes)
- Ivan Miranda's snap-pin fasteners and similar snaps (p. 1009) 02021-11-11 (updated 02021-12-30) (3 minutes)
- Vernier indicator (p. 1040) 02021-11-22 (updated 02021-12-30) (6 minutes)
- Toggling eccentrics for removing preload from spring clamps (p. 1129) 02021-12-28 (updated 02021-12-31) (22 minutes)

Notes concerning “File formats”

• Veskeno is a “fantasy platform” like TIC-80 (p. 267) 02021-03-21 (updated 02021-03-22) (3 minutes)
• Diskstrings: Bernstein’s netstrings for single-pass streaming output (p. 356) 02021-04-21 (updated 02021-07-27) (4 minutes)
• Notes on the PDF file format (p. 439) 02021-06-16 (updated 02021-07-27) (15 minutes)
• Stack syntax (p. 453) 02021-06-22 (updated 02021-07-27) (4 minutes)
• Compressed appendable file (p. 606) 02021-07-19 (updated 02021-07-27) (5 minutes)

Notes concerning “Copper”

• Forming steel with copper instead of vice versa (p. 344) 02021-04-16 (updated 02021-06-12) (2 minutes)
• Dense fillers (p. 772) 02021-08-25 (updated 02021-12-30) (7 minutes)
• Selective laser sintering of copper (p. 775) 02021-08-30 (updated 02021-12-30) (6 minutes)
• At small scales, electrowinning may be cheaper than smelting (p. 1029) 02021-11-21 (updated 02021-12-30) (25 minutes)
• Ghetto electrochromic displays for ultra-low-power computing? (p. 1082) 02021-12-16 (updated 02021-12-30) (9 minutes)

Notes concerning “Cements”

- Verstickulite (p. 457) 02021-06-23 (updated 02021-07-27) (3 minutes)
- More cements (p. 466) 02021-06-26 (updated 02021-08-15) (5 minutes)
- Powder-bed 3-D printing with a sacrificial binder (p. 506) 02021-06-30 (updated 02021-12-30) (12 minutes)
- Electrolytic berlinite (p. 561) 02021-07-12 (updated 02021-12-30) (7 minutes)
- SHS of magnesium phosphate (p. 608) 02021-07-22 (updated 02021-07-27) (3 minutes)

Notes concerning “Bytecode”

• How do you fit a high-level language into a microcontroller? Let’s look at BBN Lisp (p. 160) 02021-02-23 (updated 02021-08-18) (76 minutes)
• A survey of imperative programming operations’ prevalence (p. 201) 02021-03-02 (updated 02021-09-11) (61 minutes)
• Smolsay: the Ur-Lisp, but with dicts instead of conses (p. 552) 02021-07-12 (updated 02021-07-27) (22 minutes)
• A compact bytecode sketch that should average about 3 bytes per line of C (p. 720) 02021-08-17 (updated 02021-09-13) (66 minutes)
• Qfitzah: a minimal term-rewriting language (p. 809) 02021-09-10 (updated 02021-12-31) (62 minutes)

Notes concerning “Aluminum foil”

• Aluminum foil (p. 413) 02021-05-24 (updated 02021-09-11) (14 minutes)
• Super ghetto digital fabrication (p. 479) 02021-06-28 (updated 02021-12-30) (33 minutes)
• Material observations (p. 633) 02021-07-29 (updated 02021-12-30) (194 minutes)
• Fast electrolytic mineral accretion (seacrete) for digital fabrication? (p. 779) 02021-09-01 (updated 02021-12-30) (52 minutes)
• Solid rock on a gossamer skeleton through exponential deposition (p. 1076) 02021-12-15 (updated 02021-12-30) (11 minutes)

Notes concerning “Vermiculite”

• Verstickulite (p. 457) 02021-06-23 (updated 02021-07-27) (3 minutes)
• Leaf vein roof (p. 600) 02021-07-16 (updated 02021-09-11) (9 minutes)
• Material observations (p. 633) 02021-07-29 (updated 02021-12-30) (194 minutes)
• Maximizing phosphate density from aqueous reaction (p. 757) 02021-08-21 (updated 02021-12-30) (8 minutes)

Notes concerning “Transactions”

• Principled APL redux (p. 22) 02021-01-03 (updated 02021-12-31) (12 minutes)
• Transactional editor (p. 35) 02021-01-14 (updated 02021-01-15) (73 minutes)
• Can transactions solve the N+1 performance problem on web pages? (p. 67) 02021-01-16 (8 minutes)
• Running scripts once per frame for guaranteed GUI responsivity (p. 303) 02021-03-23 (updated 02021-10-12) (7 minutes)

Notes concerning “Sparks”

• The spark-pen pointing device (p. 921) 02021-10-10 (updated 02021-10-12) (1 minute)
• Triggering a spark gap with an exploding wire (p. 953) 02021-10-19 (updated 02021-12-30) (1 minute)
• Triggering a spark gap with low jitter using ultraviolet LEDs? (p. 954) 02021-10-20 (updated 02021-10-23) (8 minutes)
• Implementation and applications of low-voltage Marx generators with solid-state avalanche breakdown? (p. 960) 02021-10-20 (updated 02021-12-31) (39 minutes)

Notes concerning “Self-propagating high-temperature synthesis (SHS)”

- Boosters for self-propagating high-temperature synthesis (SHS) (p. 604) 02021-07-17 (updated 02021-12-30) (4 minutes)
- SHS of magnesium phosphate (p. 608) 02021-07-22 (updated 02021-07-27) (3 minutes)
- Glass wood (p. 755) 02021-08-21 (updated 02021-12-30) (4 minutes)
- A construction set using SHS (p. 765) 02021-08-24 (updated 02021-09-11) (5 minutes)

Notes concerning “Scanning probe microscopy”

- Minkowski deconvolution (p. 428) 02021-06-02 (updated 02021-12-30) (6 minutes)
- Compliance spectroscopy (p. 849) 02021-09-29 (updated 02021-12-30) (4 minutes)
- Wire brush microscope (p. 1006) 02021-11-06 (updated 02021-12-30) (1 minute)
- Aqueous scanning probe microscopy (p. 1013) 02021-11-12 (updated 02021-12-30) (7 minutes)

Notes concerning “Reverse Polish notation (RPN)”

• Some notes on compiling and notations for grammars, starting from the inspiring RPN example in Parson (p. 57) 02021-01-15 (updated 02021-12-31) (15 minutes)
• How do you fit a high-level language into a microcontroller? Let’s look at BBN Lisp (p. 160) 02021-02-23 (updated 02021-08-18) (76 minutes)
• Rator-port GUIs (p. 496) 02021-06-29 (updated 02021-12-30) (26 minutes)
• A compact bytecode sketch that should average about 3 bytes per line of C (p. 720) 02021-08-17 (updated 02021-09-13) (66 minutes)

Notes concerning “Reading”

• Some notes on perusing the Udanax Green codebase (p. 860) 02021-10-05 (updated 02021-10-08) (12 minutes)
• Some notes on reading parts of Reuleaux’s engineering handbook (p. 1019) 02021-11-17 (updated 02021-12-30) (7 minutes)
• Interesting works that entered the public domain in 02021, in the US and elsewhere (p. 1024) 02021-11-20 (updated 02021-12-30) (15 minutes)
• Some notes on Bhattacharyya’s ECM book (p. 1043) 02021-11-25 (updated 02021-12-30) (11 minutes)

Notes concerning “Poly(vinyl alcohol) (PVA)”

• Multicolor filament (p. 397) 02021-05-17 (updated 02021-12-30) (5 minutes)
• Material observations (p. 633) 02021-07-29 (updated 02021-12-30) (194 minutes)
• 3-D printing in poly(vinyl alcohol) (p. 1080) 02021-12-15 (updated 02021-12-30) (2 minutes)
• Electrolytic 2-D cutting and related electrolytic digital fabrication processes (p. 1085) 02021-12-16 (updated 02021-12-30) (48 minutes)

Notes concerning “Program calculator”

- Vaughan Pratt and Henry Baker’s COMFY control-flow combinators (p. 234) 02021-03-04 (updated 02021-03-20) (8 minutes)
- The algebra of N-ary relations (p. 432) 02021-06-14 (updated 02021-07-27) (4 minutes)
- An algebra of partial functions for interactively composing programs (p. 933) 02021-10-10 (updated 02021-12-30) (3 minutes)
- Example based regexp (p. 984) 02021-10-24 (updated 02021-12-30) (5 minutes)

Notes concerning “Pascal”

• First class locations (p. 27) 02021-01-04 (3 minutes)
• Improvements on C for low-level programming such as block arguments (p. 584) 02021-07-14 (updated 02021-12-30) (8 minutes)
• Compilation of block arguments to high-performance code (p. 624) 02021-07-29 (updated 02021-12-30) (19 minutes)
• A compact bytecode sketch that should average about 3 bytes per line of C (p. 720) 02021-08-17 (updated 02021-09-13) (66 minutes)

Notes concerning “Operating systems”

• Layout typescript (p. 29) 02021-01-04 (5 minutes)
• How little code can a filesystem be? (p. 438) 02021-06-16 (updated 02021-07-27) (1 minute)
• Self hosting kernel (p. 452) 02021-06-21 (updated 02021-12-30) (1 minute)
• A kernel you can type commands to (p. 474) 02021-06-27 (updated 02021-12-30) (1 minute)

Notes concerning “OCaml”

• Garbage-collected allocation performance on current computers (p. 245) 02021-03-13 (updated 02021-04-08) (4 minutes)
• Open coded primitives (p. 283) 02021-03-22 (26 minutes)
• Minor improvements to pattern matching (p. 306) 02021-03-24 (updated 02021-04-08) (10 minutes)
• Improvements on C for low-level programming such as block arguments (p. 584) 02021-07-14 (updated 02021-12-30) (8 minutes)

Notes concerning “Memory hardware”

• Energy autonomous computing (p. 143) 02021-02-18 (updated 02021-12-30) (58 minutes)
• Refreshing Flash memory periodically for archival (p. 198) 02021-03-02 (1 minute)
• A bargain-basement Holter monitor with a BOM under US$2.50 (p. 323) 02021-04-07 (updated 02021-07-27) (33 minutes)
• Beyond overstrike (p. 922) 02021-10-10 (updated 02021-12-30) (13 minutes)

Notes concerning “Life support”

• Notes on Richards et al.’s nascent catalytic ROS water treatment process (p. 534) 02021-07-07 (updated 02021-07-27) (14 minutes)
• Leaf vein roof (p. 600) 02021-07-16 (updated 02021-09-11) (9 minutes)
• The relation between solar-panel efficiency for air conditioning and insulation thickness (p. 941) 02021-10-11 (updated 02021-12-30) (3 minutes)
• The Habitaculum: a modular dwelling machine (p. 1061) 02021-12-13 (updated 02021-12-31) (16 minutes)

Notes concerning “Input devices”

• A four-dimensional keyboard matrix made of linear voltage differential transformers (LVDTs) to get 30 or 180 keys on five pins (p. 390) 02021-05-12 (updated 02021-12-30) (4 minutes)
• Planetary screw potentiometer (p. 392) 02021-05-12 (updated 02021-12-30) (1 minute)
• Notes on what would be needed to drive a PS/2 keyboard from an Arduino (p. 447) 02021-06-20 (updated 02021-12-30) (12 minutes)
• The spark-pen pointing device (p. 921) 02021-10-10 (updated 02021-10-12) (1 minute)

Notes concerning “Heating”

• Material observations (p. 633) 02021-07-29 (updated 02021-12-30) (194 minutes)
• Pocket kiln (p. 704) 02021-08-09 (updated 02021-08-15) (7 minutes)
• Heating a shower tank with portable TCES? (p. 714) 02021-08-11 (updated 02021-08-15) (6 minutes)
• Regenerative muffle kiln (p. 1108) 02021-12-21 (updated 02021-12-30) (19 minutes)

Notes concerning “Glass”

- Electrolytic glass machining (p. 477) 02021-06-28 (updated 02021-12-30) (6 minutes)
- Glass powder-bed 3-D printing (p. 490) 02021-06-29 (updated 02021-12-30) (20 minutes)
- Glass foam (p. 595) 02021-07-16 (updated 02021-08-15) (17 minutes)
- Material observations (p. 633) 02021-07-29 (updated 02021-12-30) (194 minutes)

Notes concerning “Garbage collection”

• Garbage-collected allocation performance on current computers (p. 245) 02021-03-13 (updated 02021-04-08) (4 minutes)
• Some notes on reading Chris Seaton’s TruffleRuby dissertation (p. 269) 02021-03-21 (updated 02021-03-22) (16 minutes)
• Open coded primitives (p. 283) 02021-03-22 (26 minutes)
• Smolsay: the Ur-Lisp, but with dicts instead of conses (p. 552) 02021-07-12 (updated 02021-07-27) (22 minutes)

Notes concerning “Encoding”

• Notes on what would be needed to drive a PS/2 keyboard from an Arduino (p. 447) 02021-06-20 (updated 02021-12-30) (12 minutes)
• Does USB bitstuffing create a timing-channel vulnerability? (p. 456) 02021-06-22 (updated 02021-12-31) (1 minute)
• Three phase differential data (p. 843) 02021-09-22 (updated 02021-12-30) (4 minutes)
• Constant weight dithering (p. 991) 02021-10-28 (updated 02021-12-30) (5 minutes)

Notes concerning “Editors”

• Transactional editor (p. 35) 02021-01-14 (updated 02021-01-15) (73 minutes)
• Wiki models (p. 751) 02021-08-19 (updated 02021-12-30) (1 minute)
• Embedding runnable code in text paragraphs for numerical modeling (p. 1002) 02021-11-06 (updated 02021-12-30) (6 minutes)
• Chording commands (p. 1047) 02021-11-26 (updated 02021-12-30) (7 minutes)

Notes concerning “Economics”

- Intel engineering positions considered as a dollar auction (p. 78) 02021-01-21 (updated 02021-01-27) (1 minute)
- The use of silver in solar cells (p. 112) 02021-02-02 (updated 02021-09-11) (8 minutes)
- Why Bitcoin is puzzling to people in rich countries (p. 312) 02021-03-31 (updated 02021-07-27) (10 minutes)
- Economic history (p. 460) 02021-06-25 (updated 02021-07-27) (17 minutes)

Notes concerning “Dynamic dispatch”

• Smolsay: the Ur-Lisp, but with dicts instead of conses (p. 552) 02021-07-12 (updated 02021-07-27) (22 minutes)
• Faygoo: a yantra-smashing ersatz version of Piumarta and Warth’s COLA (p. 570) 02021-07-14 (updated 02021-12-30) (17 minutes)
• Improvements on C for low-level programming such as block arguments (p. 584) 02021-07-14 (updated 02021-12-30) (8 minutes)
• Qfitzah: a minimal term-rewriting language (p. 809) 02021-09-10 (updated 02021-12-31) (62 minutes)

Notes concerning “Domain-specific languages (DSLs)”

• Some notes on compiling and notations for grammars, starting from the inspiring RPN example in Parson (p. 57) 02021-01-15 (updated 02021-12-31) (15 minutes)
• Memory view (p. 539) 02021-07-09 (updated 02021-12-30) (6 minutes)
• Example based regexp (p. 984) 02021-10-24 (updated 02021-12-30) (5 minutes)
• DSLs for calculations on dates (p. 1018) 02021-11-14 (updated 02021-12-30) (1 minute)

Notes concerning “Displays”

• Energy autonomous computing (p. 143) 02021-02-18 (updated 02021-12-30) (58 minutes)
• Subnanosecond thermochromic light modulation for real-time holography and displays (p. 531) 02021-07-06 (updated 02021-12-30) (8 minutes)
• A simple 2-D programmable graphics pipeline to unify tiles and palettes (p. 1022) 02021-11-18 (updated 02021-12-30) (6 minutes)
• Ghetto electrochromic displays for ultra-low-power computing? (p. 1082) 02021-12-16 (updated 02021-12-30) (9 minutes)

Notes concerning “Control (cybernetics)”

• Clay wire cutter (p. 409) 02021-05-21 (updated 02021-12-30) (2 minutes)
• Super ghetto digital fabrication (p. 479) 02021-06-28 (updated 02021-12-30) (33 minutes)
• Negative feedback control to prevent runaway positive feedback in 3-D MIG welding printing (p. 777) 02021-08-30 (updated 02021-12-30) (3 minutes)
• Adversarial control (p. 987) 02021-10-25 (updated 02021-12-30) (13 minutes)

Notes concerning “Compression”

• Compressed appendable file (p. 606) 02021-07-19 (updated 02021-07-27) (5 minutes)
• Residual stream windowing (p. 752) 02021-08-21 (updated 02021-09-11) (5 minutes)
• Spanish phonology (p. 867) 02021-10-05 (updated 02021-12-31) (15 minutes)
• Paeth prediction and vector quantization (p. 1005) 02021-11-06 (updated 02021-12-30) (1 minute)

Notes concerning “Communication”

• Can you do direct digital synthesis (DDS) at over a gigahertz? (p. 119) 02021-02-08 (updated 02021-02-24) (30 minutes)
• A bargain-basement Holter monitor with a BOM under US$2.50 (p. 323) 02021-04-07 (updated 02021-07-27) (33 minutes)
• Subnanosecond thermochromic light modulation for real-time holography and displays (p. 531) 02021-07-06 (updated 02021-12-30) (8 minutes)
• Constant weight dithering (p. 991) 02021-10-28 (updated 02021-12-30) (5 minutes)

Notes concerning “Ceramic-matrix composites (CMCs)”

• Electrolytic berlinite (p. 561) 02021-07-12 (updated 02021-12-30) (7 minutes)
• Fiberglass CMCs? (p. 588) 02021-07-15 (updated 02021-07-27) (8 minutes)
• Glass wood (p. 755) 02021-08-21 (updated 02021-12-30) (4 minutes)
• Exotic steel analogues in other metals (p. 1050) 02021-12-01 (updated 02021-12-30) (8 minutes)

Notes concerning “Caching”

• Principled APL redux (p. 22) 02021-01-03 (updated 02021-12-31) (12 minutes)
• Relayout with heaps (p. 32) 02021-01-10 (updated 02021-01-15) (6 minutes)
• Trie PEGs (p. 53) 02021-01-15 (4 minutes)
• Leaf hypertext (p. 380) 02021-05-08 (updated 02021-12-30) (3 minutes)

Notes concerning “Weighing”

• Weighing an eyelash on an improvised Kibble balance (p. 382) 02021-05-08 (updated 02021-12-30) (3 minutes)
• The ayurvedic “fire mud” of Bhudeb Mookerji and modern castable refractories (p. 688) 02021-08-05 (updated 02021-08-15) (22 minutes)

• Weighing balance design (p. 799) 02021-09-06 (updated 02021-12-30) (9 minutes)

Notes concerning “Unix”

- How little code can a filesystem be? (p. 438) 02021-06-16 (updated 02021-07-27) (1 minute)
- Self hosting kernel (p. 452) 02021-06-21 (updated 02021-12-30) (1 minute)
- A short list of the most useful Unix CLI tools (p. 841) 02021-09-15 (updated 02021-09-16) (2 minutes)

Notes concerning “Tiled graphics”

• Skew tilesets (p. 135) 02021-02-14 (updated 02021-02-24) (7 minutes)
• Pipelined piece chain painting (p. 926) 02021-10-10 (updated 02021-12-30) (23 minutes)
• A simple 2-D programmable graphics pipeline to unify tiles and palettes (p. 1022) 02021-11-18 (updated 02021-12-30) (6 minutes)

Notes concerning “Term rewriting”

• Qfitzah: a minimal term-rewriting language (p. 809) 02021-09-10 (updated 02021-12-31) (62 minutes)
• Qfitzah internals (p. 846) 02021-09-24 (updated 02021-12-30) (2 minutes)
• Deriving binary search (p. 855) 02021-10-01 (updated 02021-12-30) (5 minutes)

Notes concerning “Sugar”

• 3-D printing in carbohydrates (p. 393) 02021-05-16 (updated 02021-12-30) (10 minutes)
• Cutting clay (p. 400) 02021-05-19 (updated 02021-12-30) (10 minutes)
• Verstickulite (p. 457) 02021-06-23 (updated 02021-07-27) (3 minutes)

Notes concerning “Sorting”

• A compact bytecode sketch that should average about 3 bytes per line of C (p. 720) 02021-08-17 (updated 02021-09-13) (66 minutes)
• Lazy heapsort (p. 761) 02021-08-22 (updated 02021-09-11) (6 minutes)
• Fung’s “I can’t believe it can sort” algorithm and others (p. 864) 02021-10-05 (updated 02021-12-30) (5 minutes)

Notes concerning “Solubility”

• ECM for machining nonmetals? (p. 523) 02021-07-05 (updated 02021-07-27) (11 minutes)
• Fast electrolytic mineral accretion (seacrete) for digital fabrication? (p. 779) 02021-09-01 (updated 02021-12-30) (52 minutes)
• Xerogel compacting (p. 1119) 02021-12-22 (updated 02021-12-30) (12 minutes)

Notes concerning “Scheme”

• A survey of imperative programming operations’ prevalence (p. 201) 02021-03-02 (updated 02021-09-11) (61 minutes)
• Minor improvements to pattern matching (p. 306) 02021-03-24 (updated 02021-04-08) (10 minutes)
• Qfitzah: a minimal term-rewriting language (p. 809) 02021-09-10 (updated 02021-12-31) (62 minutes)

Notes concerning “Roller screws”

• Planetary roller screw worm drive (p. 375) 02021-05-07 (updated 02021-12-30) (4 minutes)
• Planetary screw potentiometer (p. 392) 02021-05-12 (updated 02021-12-30) (1 minute)
• Thread rolling roller screw (p. 999) 02021-11-04 (updated 02021-12-30) (1 minute)

Notes concerning “RISC-V”

• Some preliminary notes on the amazing RISC-V architecture (p. 82) 02021-01-24 (updated 02021-07-27) (29 minutes)
• Energy autonomous computing (p. 143) 02021-02-18 (updated 02021-12-30) (58 minutes)
• Compilation of block arguments to high-performance code (p. 624) 02021-07-29 (updated 02021-12-30) (19 minutes)

Notes concerning “Reproducibility”

- Veskeno is a “fantasy platform” like TIC-80 (p. 267) 02021-03-21 (updated 02021-03-22) (3 minutes)
- Leaf hypertext (p. 380) 02021-05-08 (updated 02021-12-30) (3 minutes)
- The nature of mathematical discourse (p. 418) 02021-05-27 (updated 02021-12-30) (5 minutes)

Notes concerning “Radio”

• Can you do direct digital synthesis (DDS) at over a gigahertz? (p. 119) 02021-02-08 (updated 02021-02-24) (30 minutes)
• A bargain-basement Holter monitor with a BOM under US$2.50 (p. 323) 02021-04-07 (updated 02021-07-27) (33 minutes)
• Flux-gate downconversion in a loopstick antenna? (p. 436) 02021-06-15 (updated 02021-07-27) (2 minutes)

Notes concerning “Politics”

• Why Bitcoin is puzzling to people in rich countries (p. 312) 02021-03-31 (updated 02021-07-27) (10 minutes)
• Is liberal democracy’s stability conditioned on historical conditions that no longer obtain? (p. 1114) 02021-12-22 (updated 02021-12-30) (16 minutes)
• Safe decentralized cloud storage (p. 1135) 02021-12-30 (10 minutes)

Notes concerning “Illinois PLATO”

- Skew tilesets (p. 135) 02021-02-14 (updated 02021-02-24) (7 minutes)
- Beyond overstrike (p. 922) 02021-10-10 (updated 02021-12-30) (13 minutes)
- Pipelined piece chain painting (p. 926) 02021-10-10 (updated 02021-12-30) (23 minutes)

Notes concerning “Poly(lactic acid) (PLA)”

• Clay-filled PLA filament for firing to ceramic (p. 396) 02021-05-17 (updated 02021-12-30) (1 minute)
• Multicolor filament (p. 397) 02021-05-17 (updated 02021-12-30) (5 minutes)
• Rock-wool-filled composites (p. 798) 02021-09-03 (updated 02021-12-30) (2 minutes)

Notes concerning "Patterning"

• Iodine patterning (p. 713) 02021-08-11 (updated 02021-08-15) (1 minute)
• Patterning metal surfaces by coating decomposition with lasers or plasma? (p. 795) 02021-09-03 (updated 02021-12-30) (7 minutes)
• Electrolytic 2-D cutting and related electrolytic digital fabrication processes (p. 1085) 02021-12-16 (updated 02021-12-30) (48 minutes)

Notes concerning “Oscillators”

• Trying to design a simple switchmode power supply using Schmitt-trigger relaxation oscillators (p. 92) 02021-01-26 (updated 02021-01-27) (32 minutes)
• Failing to stabilize the amplitude of an opamp phase-delay oscillator (p. 298) 02021-03-23 (updated 02021-03-24) (10 minutes)
• Three phase logic (p. 364) 02021-04-30 (updated 02021-07-27) (9 minutes)

Notes concerning “Natural-language processing”

• Generating novel unique pronounceable identifiers with letter frequency data (p. 239) 02021-03-10 (updated 02021-03-22) (11 minutes)
• Spanish phonology (p. 867) 02021-10-05 (updated 02021-12-31) (15 minutes)
• Wordlists for maximum drama (p. 904) 02021-10-08 (updated 02021-12-30) (16 minutes)

Notes concerning “Memory models”

• How do you fit a high-level language into a microcontroller? Let’s look at BBN Lisp (p. 160) 02021-02-23 (updated 02021-08-18) (76 minutes)
• Safe FORTH with the FORTRAN memory model? (p. 351) 02021-04-21 (updated 02021-06-12) (2 minutes)
• Improvements on C for low-level programming such as block arguments (p. 584) 02021-07-14 (updated 02021-12-30) (8 minutes)

Notes concerning “LEDs”

• Triggering a spark gap with low jitter using ultraviolet LEDs? (p. 954) 02021-10-20 (updated 02021-10-23) (8 minutes)
• Implementation and applications of low-voltage Marx generators with solid-state avalanche breakdown? (p. 960) 02021-10-20 (updated 02021-12-31) (39 minutes)
• Simplest blinker (p. 1053) 02021-12-01 (updated 02021-12-30) (9 minutes)

Notes concerning “Kleene algebras”

• Some notes on compiling and notations for grammars, starting from the inspiring RPN example in Parson (p. 57) 02021-01-15 (updated 02021-12-31) (15 minutes)
• The algebra of N-ary relations (p. 432) 02021-06-14 (updated 02021-07-27) (4 minutes)
• Example based regexp (p. 984) 02021-10-24 (updated 02021-12-30) (5 minutes)

Notes concerning “Kingery, the father of modern ceramics”

• Material observations (p. 633) 02021-07-29 (updated 02021-12-30) (194 minutes)
• The ayurvedic “fire mud” of Bhudeb Mookerji and modern castable refractories (p. 688) 02021-08-05 (updated 02021-08-15) (22 minutes)

• Maximizing phosphate density from aqueous reaction (p. 757) 02021-08-21 (updated 02021-12-30) (8 minutes)

Notes concerning “Keyboards”

• A four-dimensional keyboard matrix made of linear voltage differential transformers (LVDTs) to get 30 or 180 keys on five pins (p. 390) 02021-05-12 (updated 02021-12-30) (4 minutes)
• Notes on what would be needed to drive a PS/2 keyboard from an Arduino (p. 447) 02021-06-20 (updated 02021-12-30) (12 minutes)
• Chording commands (p. 1047) 02021-11-26 (updated 02021-12-30) (7 minutes)

Notes concerning “Insulation”

• Leaf vein roof (p. 600) 02021-07-16 (updated 02021-09-11) (9 minutes)
• Pocket kiln (p. 704) 02021-08-09 (updated 02021-08-15) (7 minutes)
• Regenerative muffle kiln (p. 1108) 02021-12-21 (updated 02021-12-30) (19 minutes)

Notes concerning "Hypertext"

• Leaf hypertext (p. 380) 02021-05-08 (updated 02021-12-30) (3 minutes)
• Bead hypertext (p. 455) 02021-06-22 (updated 02021-12-30) (1 minute)
• Some notes on perusing the Udanax Green codebase (p. 860) 02021-10-05 (updated 02021-10-08) (12 minutes)

Notes concerning “Humor”

• Greek operating systems (p. 430) 02021-06-04 (updated 02021-06-12) (4 minutes)
• Nuclear energy is the Amiga of energy sources (p. 434) 02021-06-14 (updated 02021-07-27) (3 minutes)
• Stochastically generated self-amalgamating tape variations for composite fabrication (p. 510) 02021-07-02 (updated 02021-12-30) (26 minutes)

Notes concerning “Hashing”

• Ropes with constant-time concatenation and equality comparisons with monoidal hash consing (p. 619) 02021-07-27 (15 minutes)
• PBKDF content addressing with keyphrase hashcash: a non-blockchain attack on Zooko’s Triangle (p. 896) 02021-10-08 (24 minutes)
• Hashing text with recursive shingling to find duplication efficiently (p. 993) 02021-10-30 (updated 02021-12-30) (6 minutes)

Notes concerning “Glutaraldehyde”

• 3-D printing in carbohydrates (p. 393) 02021-05-16 (updated 02021-12-30) (10 minutes)
• 3-D printing in poly(vinyl alcohol) (p. 1080) 02021-12-15 (updated 02021-12-30) (2 minutes)
• Electrolytic 2-D cutting and related electrolytic digital fabrication processes (p. 1085) 02021-12-16 (updated 02021-12-30) (48 minutes)

Notes concerning “Forming”

• Forming steel with copper instead of vice versa (p. 344) 02021-04-16 (updated 02021-06-12) (2 minutes)
• Cutting clay (p. 400) 02021-05-19 (updated 02021-12-30) (10 minutes)
• Electroforming rivets (p. 410) 02021-05-22 (updated 02021-12-30) (2 minutes)

Notes concerning “Flying”

• Methane bag (p. 710) 02021-08-10 (updated 02021-08-15) (8 minutes)
• Micro ramjet (p. 1038) 02021-11-22 (updated 02021-12-30) (3 minutes)
• Against subjectivism (p. 1066) 02021-12-15 (updated 02021-12-30) (36 minutes)

Notes concerning “Fasteners”

• A construction set using SHS (p. 765) 02021-08-24 (updated 02021-09-11) (5 minutes)
• Better screw head designs? (p. 770) 02021-08-25 (updated 02021-09-11) (4 minutes)
• Ivan Miranda’s snap-pin fasteners and similar snaps (p. 1009) 02021-11-11 (updated 02021-12-30) (3 minutes)

Notes concerning “Emacs”

- Layout typescript (p. 29) 02021-01-04 (5 minutes)
- Transactional editor (p. 35) 02021-01-14 (updated 02021-01-15) (73 minutes)
- Chording commands (p. 1047) 02021-11-26 (updated 02021-12-30) (7 minutes)

Notes concerning “Control flow”

• Vaughan Pratt and Henry Baker’s COMFY control-flow combinators (p. 234) 02021-03-04 (updated 02021-03-20) (8 minutes)
• Subset of C for the simplest self-compiling compiler (p. 717) 02021-08-12 (updated 02021-12-30) (6 minutes)
• Example based regexp (p. 984) 02021-10-24 (updated 02021-12-30) (5 minutes)

Notes concerning “COMFY-*”

• Vaughan Pratt and Henry Baker’s COMFY control-flow combinators (p. 234) 02021-03-04 (updated 02021-03-20) (8 minutes)
• Open coded primitives (p. 283) 02021-03-22 (26 minutes)
• Example based regexp (p. 984) 02021-10-24 (updated 02021-12-30) (5 minutes)

Notes concerning “Cameras”

• iPhone replacement cameras as 6-μs streak cameras (p. 80) 02021-01-22 (updated 02021-12-30) (2 minutes)
• Phased-array imaging sonar from a mesh network of self-localizing sensor nodes (p. 358) 02021-04-27 (updated 02021-12-30) (8 minutes)

• Differential filming (p. 374) 02021-05-07 (updated 02021-12-30) (1 minute)

Notes concerning “Batteries”

- Energy autonomous computing (p. 143) 02021-02-18 (updated 02021-12-30) (58 minutes)
- Aluminum fuel (p. 603) 02021-07-17 (updated 02021-12-30) (2 minutes)
- Methane bag (p. 710) 02021-08-10 (updated 02021-08-15) (8 minutes)

Notes concerning "BASIC"

• Skew tilesets (p. 135) 02021-02-14 (updated 02021-02-24) (7 minutes)
• Manually writing code in static single assignment (SSA) form, inspired by Kemeny's DOPE, isn't worth it (p. 353) 02021-04-21 (updated 02021-06-12) (3 minutes)
• My Heathkit H8 (p. 996) 02021-11-03 (updated 02021-12-30) (2 minutes)

Notes concerning “Audio”

• Brute force speech (p. 262) 02021-03-21 (updated 02021-03-22) (7 minutes)
• Phased-array imaging sonar from a mesh network of self-localizing sensor nodes (p. 358) 02021-04-27 (updated 02021-12-30) (8 minutes)

• The spark-pen pointing device (p. 921) 02021-10-10 (updated 02021-10-12) (1 minute)

Notes concerning “ASCII art”

• ASCII art, but in Unicode, with Braille and other alternatives (p. 128) 02021-02-10 (updated 02021-02-24) (9 minutes)
• Skew tilesets (p. 135) 02021-02-14 (updated 02021-02-24) (7 minutes)
• Thumbnail views in a Unicode character-cell terminal with Braille (p. 142) 02021-02-17 (updated 02021-02-24) (1 minute)

Notes concerning "Art"

• ASCII art, but in Unicode, with Braille and other alternatives (p. 128) 02021-02-10 (updated 02021-02-24) (9 minutes)
• Threechet (p. 140) 02021-02-16 (updated 02021-02-24) (4 minutes)

• How Lao Yuxi painted a cock (p. 247) 02021-03-19 (updated 02021-04-14) (7 minutes)

Notes concerning “Artificial neural networks”

• Designing curiosity and dreaming into optimizing systems (p. 420) 02021-05-30 (updated 02021-12-30) (6 minutes)
• Adversarial control (p. 987) 02021-10-25 (updated 02021-12-30) (13 minutes)
• Rendering 3-D graphics with PINNs and GANs? (p. 1010) 02021-11-11 (updated 02021-12-30) (10 minutes)

Notes concerning “Allocation performance”

- Garbage-collected allocation performance on current computers (p. 245) 02021-03-13 (updated 02021-04-08) (4 minutes)
- Open coded primitives (p. 283) 02021-03-22 (26 minutes)
- Running scripts once per frame for guaranteed GUI responsivity (p. 303) 02021-03-23 (updated 02021-10-12) (7 minutes)

Notes concerning “Alabaster”

• Duplicating Durham’s Rock-Hard Putty (p. 79) 02021-01-22 (updated 02021-01-27) (1 minute)
• Leaf vein roof (p. 600) 02021-07-16 (updated 02021-09-11) (9 minutes)
• Material observations (p. 633) 02021-07-29 (updated 02021-12-30) (194 minutes)

Notes concerning “X rays”

• Can you 3-D print Sorel cement by inhibiting setting with X-rays? (p. 592) 02021-07-16 (updated 02021-07-27) (1 minute)
• MOSFET body diodes as Geiger counter avalanche detectors? (p. 1103) 02021-12-17 (updated 02021-12-30) (1 minute)

Notes concerning “Wiki”

• Leaf hypertext (p. 380) 02021-05-08 (updated 02021-12-30)
(3 minutes)
• Wiki models (p. 751) 02021-08-19 (updated 02021-12-30)
(1 minute)

Notes concerning “Video”

• Layout typescript (p. 29) 02021-01-04 (5 minutes)
• Residual stream windowing (p. 752) 02021-08-21 (updated 02021-09-11) (5 minutes)

Notes concerning “The Veskeno virtual machine”

- A survey of imperative programming operations’ prevalence (p. 201) 02021-03-02 (updated 02021-09-11) (61 minutes)
- Veskeno is a “fantasy platform” like TIC-80 (p. 267) 02021-03-21 (updated 02021-03-22) (3 minutes)

Notes concerning “The United States of America (USA)”

• Economic history (p. 460) 02021-06-25 (updated 02021-07-27) (17 minutes)
• Interesting works that entered the public domain in 02021, in the US and elsewhere (p. 1024) 02021-11-20 (updated 02021-12-30) (15 minutes)

Notes concerning “Unicode”

• ASCII art, but in Unicode, with Braille and other alternatives (p. 128) 02021-02-10 (updated 02021-02-24) (9 minutes)
• Thumbnail views in a Unicode character-cell terminal with Braille (p. 142) 02021-02-17 (updated 02021-02-24) (1 minute)

Notes concerning “Toxicology”

• Dipropylene glycol (p. 687) 02021-08-01 (updated 02021-08-15) (2 minutes)
• Patterning metal surfaces by coating decomposition with lasers or plasma? (p. 795) 02021-09-03 (updated 02021-12-30) (7 minutes)

Notes concerning “Thixotropy”

• Duplicating Durham’s Rock-Hard Putty (p. 79) 02021-01-22 (updated 02021-01-27) (1 minute)
• Thixotropic electrodeposition (p. 372) 02021-05-04 (updated 02021-12-31) (2 minutes)

Notes concerning “Tcl”

• A survey of imperative programming operations’ prevalence (p. 201) 02021-03-02 (updated 02021-09-11) (61 minutes)
• Compilation of block arguments to high-performance code (p. 624) 02021-07-29 (updated 02021-12-30) (19 minutes)

Notes concerning “Talc”

• Duplicating Durham’s Rock-Hard Putty (p. 79) 02021-01-22 (updated 02021-01-27) (1 minute)
• Firing talc (p. 576) 02021-07-14 (updated 02021-12-30) (17 minutes)

Notes concerning “Stack machines”

• .xosm: experimental obvious stack machine (p. 274) 02021-03-21 (updated 02021-03-24) (20 minutes)
• Stack syntax (p. 453) 02021-06-22 (updated 02021-07-27) (4 minutes)

Notes concerning “Spreadtools”

• Spreadtool (p. 543) 02021-07-10 (updated 02021-12-30) (30 minutes)
• Toggling eccentrics for removing preload from spring clamps (p. 1129) 02021-12-28 (updated 02021-12-31) (22 minutes)

Notes concerning “Speech synthesis”

• Brute force speech (p. 262) 02021-03-21 (updated 02021-03-22) (7 minutes)
• Spanish phonology (p. 867) 02021-10-05 (updated 02021-12-31) (15 minutes)

Notes concerning “Space”

• How fast do von Neumann probes need to reproduce to colonize space in our lifetimes? (p. 368) 02021-05-04 (updated 02021-06-12) (5 minutes)
• Photoemissive power (p. 1124) 02021-12-23 (updated 02021-12-28) (15 minutes)

Notes concerning “Sonic screwdrivers”

• Micro impact driver (p. 427) 02021-06-02 (updated 02021-06-12) (2 minutes)
• Sonic screwdriver resonance (p. 527) 02021-07-06 (updated 02021-12-30) (11 minutes)

Notes concerning “Snaps”

• Spreadtool (p. 543) 02021-07-10 (updated 02021-12-30) (30 minutes)
• Ivan Miranda’s snap-pin fasteners and similar snaps (p. 1009) 02021-11-11 (updated 02021-12-30) (3 minutes)

Notes concerning “Smalltalk”

• Improvements on C for low-level programming such as block arguments (p. 584) 02021-07-14 (updated 02021-12-30) (8 minutes)
• Compilation of block arguments to high-performance code (p. 624) 02021-07-29 (updated 02021-12-30) (19 minutes)

Notes concerning “Spatial light modulators (SLMs)”

• Subnanosecond thermochromic light modulation for real-time holography and displays (p. 531) 02021-07-06 (updated 02021-12-30) (8 minutes)
• Ghetto electrochromic displays for ultra-low-power computing? (p. 1082) 02021-12-16 (updated 02021-12-30) (9 minutes)

Notes concerning “Silver”

• The use of silver in solar cells (p. 112) 02021-02-02 (updated 02021-09-11) (8 minutes)
• Ghetto electrochromic displays for ultra-low-power computing? (p. 1082) 02021-12-16 (updated 02021-12-30) (9 minutes)

Notes concerning "Silicone"

• Selectively curing one-component silicone by injecting water (p. 408) 02021-05-19 (updated 02021-12-30) (2 minutes)
• Super ghetto digital fabrication (p. 479) 02021-06-28 (updated 02021-12-30) (33 minutes)

Notes concerning “Steel Bank Common Lisp”

• Garbage-collected allocation performance on current computers (p. 245) 02021-03-13 (updated 02021-04-08) (4 minutes)
• Open coded primitives (p. 283) 02021-03-22 (26 minutes)

Notes concerning “Sapphire”

• More cements (p. 466) 02021-06-26 (updated 02021-08-15) (5 minutes)
• Super ghetto digital fabrication (p. 479) 02021-06-28 (updated 02021-12-30) (33 minutes)

Notes concerning “Sandblasting”

• Super ghetto digital fabrication (p. 479) 02021-06-28 (updated 02021-12-30) (33 minutes)
• ECM for machining nonmetals? (p. 523) 02021-07-05 (updated 02021-07-27) (11 minutes)

Notes concerning “Ropes”

• Ropes with constant-time concatenation and equality comparisons with monoidal hash consing (p. 619) 02021-07-27 (15 minutes)
• Balanced ropes (p. 948) 02021-10-16 (updated 02021-12-30) (7 minutes)

Notes concerning “Regenerators”

• A boiler for submillisecond steam pulses (p. 361) 02021-04-28 (updated 02021-12-30) (10 minutes)
• Regenerative muffle kiln (p. 1108) 02021-12-21 (updated 02021-12-30) (19 minutes)

Notes concerning "Refining"

• At small scales, electrowinning may be cheaper than smelting (p. 1029) 02021-11-21 (updated 02021-12-30) (25 minutes)
• Aluminum refining (p. 1106) 02021-12-20 (updated 02021-12-30) (3 minutes)

Notes concerning “Randomness”

• Generating novel unique pronounceable identifiers with letter frequency data (p. 239) 02021-03-10 (updated 02021-03-22) (11 minutes)
• Wordlists for maximum drama (p. 904) 02021-10-08 (updated 02021-12-30) (16 minutes)

Notes concerning “Qfitzah”

• Qfitzah: a minimal term-rewriting language (p. 809) 02021-09-10 (updated 02021-12-31) (62 minutes)
• Qfitzah internals (p. 846) 02021-09-24 (updated 02021-12-30) (2 minutes)

Notes concerning “Prefix sums”

• Fibonacci scan (p. 31) 02021-01-10 (updated 02021-01-15)
(1 minute)
• Relayout with heaps (p. 32) 02021-01-10 (updated 02021-01-15)
(6 minutes)

Notes concerning “Plasma”

• Patterning metal surfaces by coating decomposition with lasers or plasma? (p. 795) 02021-09-03 (updated 02021-12-30) (7 minutes)
• The sol-gel transition and selective gelling for 3-D printing (p. 858) 02021-10-03 (updated 02021-12-30) (6 minutes)

Notes concerning “Piezoelectrics”

• Ranking MOSFETs for, say, rapid localized electrolysis to make optics (p. 938) 02021-10-11 (updated 02021-12-30) (8 minutes)
• Viscoelastic probing (p. 1000) 02021-11-04 (updated 02021-12-30) (2 minutes)

Notes concerning “Photoemission”

• Triggering a spark gap with low jitter using ultraviolet LEDs? (p. 954) 02021-10-20 (updated 02021-10-23) (8 minutes)
• Photoemissive power (p. 1124) 02021-12-23 (updated 02021-12-28) (15 minutes)

Notes concerning “Perl”

- A survey of imperative programming operations’ prevalence (p. 201) 02021-03-02 (updated 02021-09-11) (61 minutes)
- Generating novel unique pronounceable identifiers with letter frequency data (p. 239) 02021-03-10 (updated 02021-03-22) (11 minutes)

Notes concerning “Parsing expression grammars (PEGs)”

• Trie PEGs (p. 53) 02021-01-15 (4 minutes)
• PEG-like flexibility for parsing right-to-left? (p. 437) 02021-06-16 (updated 02021-07-27) (2 minutes)

Notes concerning “Passwords”

• PBKDF content addressing with keyphrase hashcash: a non-blockchain attack on Zooko’s Triangle (p. 896) 02021-10-08 (24 minutes)
• Wordlists for maximum drama (p. 904) 02021-10-08 (updated 02021-12-30) (16 minutes)

Notes concerning "The Paeth predictor"

• Residual stream windowing (p. 752) 02021-08-21 (updated 02021-09-11) (5 minutes)
• Paeth prediction and vector quantization (p. 1005) 02021-11-06 (updated 02021-12-30) (1 minute)

Notes concerning “Memory ownership”

- First class locations (p. 27) 02021-01-04 (3 minutes)
- Some notes on IPL-VI, Lisp’s 01958 precursor (p. 196) 02021-03-02 (4 minutes)

Notes concerning “Overstrike”

• Skew tilesets (p. 135) 02021-02-14 (updated 02021-02-24) (7 minutes)
• Beyond overstrike (p. 922) 02021-10-10 (updated 02021-12-30) (13 minutes)

Notes concerning “Mathematical optimization”

• Designing curiosity and dreaming into optimizing systems (p. 420) 02021-05-30 (updated 02021-12-30) (6 minutes)
• Adversarial control (p. 987) 02021-10-25 (updated 02021-12-30) (13 minutes)

Notes concerning “Oogoo”

• Super ghetto digital fabrication (p. 479) 02021-06-28 (updated 02021-12-30) (33 minutes)
• Material observations (p. 633) 02021-07-29 (updated 02021-12-30) (194 minutes)

Notes concerning “Ontology”

• The nature of mathematical discourse (p. 418) 02021-05-27 (updated 02021-12-30) (5 minutes)
• Against subjectivism (p. 1066) 02021-12-15 (updated 02021-12-30) (36 minutes)

Notes concerning “Namespaces”

• Some notes on perusing the Udanax Green codebase (p. 860) 02021-10-05 (updated 02021-10-08) (12 minutes)
• PBKDF content addressing with keyphrase hashcash: a non-blockchain attack on Zooko’s Triangle (p. 896) 02021-10-08 (24 minutes)

Notes concerning “m4”

• A survey of imperative programming operations’ prevalence (p. 201) 02021-03-02 (updated 02021-09-11) (61 minutes)
• :fqozl, a normal-order text macro language (p. 336) 02021-04-09 (updated 02021-07-27) (14 minutes)

Notes concerning "LuaJIT"

• Garbage-collected allocation performance on current computers (p. 245) 02021-03-13 (updated 02021-04-08) (4 minutes)
• Smolsay: the Ur-Lisp, but with dicts instead of conses (p. 552) 02021-07-12 (updated 02021-07-27) (22 minutes)

Notes concerning “Lua”

• A survey of imperative programming operations’ prevalence (p. 201) 02021-03-02 (updated 02021-09-11) (61 minutes)
• Smolsay: the Ur-Lisp, but with dicts instead of conses (p. 552) 02021-07-12 (updated 02021-07-27) (22 minutes)

Notes concerning “LiDAR”

• Subnanosecond thermochromic light modulation for real-time holography and displays (p. 531) 02021-07-06 (updated 02021-12-30) (8 minutes)
• Implementation and applications of low-voltage Marx generators with solid-state avalanche breakdown? (p. 960) 02021-10-20 (updated 02021-12-31) (39 minutes)

Notes concerning “Length”

• Base 3 gage blocks (p. 468) 02021-06-27 (updated 02021-12-30) (5 minutes)
• Vernier indicator (p. 1040) 02021-11-22 (updated 02021-12-30) (6 minutes)

Notes concerning “Layout”

• Layout typescript (p. 29) 02021-01-04 (5 minutes)
• Relayout with heaps (p. 32) 02021-01-10 (updated 02021-01-15) (6 minutes)

Notes concerning “Latency”

• Running scripts once per frame for guaranteed GUI responsivity (p. 303) 02021-03-23 (updated 02021-10-12) (7 minutes)
• Lazy heapsort (p. 761) 02021-08-22 (updated 02021-09-11) (6 minutes)

Notes concerning “The JS programming language”

- A survey of imperative programming operations’ prevalence (p. 201) 02021-03-02 (updated 02021-09-11) (61 minutes)
- Smolsay: the Ur-Lisp, but with dicts instead of conses (p. 552) 02021-07-12 (updated 02021-07-27) (22 minutes)

Notes concerning “JLCPCB (JiaLiChuang)”

• Can you get JLCPCB to fabricate a CPU for you affordably from “basic” parts? (p. 347) 02021-04-17 (updated 02021-12-30) (9 minutes)
• Three phase logic (p. 364) 02021-04-30 (updated 02021-07-27) (9 minutes)

Notes concerning “Interrupts”

• Running scripts once per frame for guaranteed GUI responsivity (p. 303) 02021-03-23 (updated 02021-10-12) (7 minutes)
• Notes on what would be needed to drive a PS/2 keyboard from an Arduino (p. 447) 02021-06-20 (updated 02021-12-30) (12 minutes)

Notes concerning “Incremental search”

• Transactional editor (p. 35) 02021-01-14 (updated 02021-01-15) (73 minutes)
• Example based regexp (p. 984) 02021-10-24 (updated 02021-12-30) (5 minutes)

Notes concerning “Household”

- Fan noise would be less annoying if intermittent (p. 21) 02021-01-03 (updated 02021-01-04) (1 minute)
- The Habitaculum: a modular dwelling machine (p. 1061) 02021-12-13 (updated 02021-12-31) (16 minutes)

Notes concerning “Gradient descent”

• Designing curiosity and dreaming into optimizing systems (p. 420) 02021-05-30 (updated 02021-12-30) (6 minutes)
• Adversarial control (p. 987) 02021-10-25 (updated 02021-12-30) (13 minutes)

Notes concerning “Gears”

• Geneva wheel stopwork (p. 321) 02021-04-07 (updated 02021-04-08) (6 minutes)
• Planetary roller screw worm drive (p. 375) 02021-05-07 (updated 02021-12-30) (4 minutes)

Notes concerning “Generative adversarial networks (GANs)”

• Adversarial control (p. 987) 02021-10-25 (updated 02021-12-30) (13 minutes)
• Rendering 3-D graphics with PINNs and GANs? (p. 1010) 02021-11-11 (updated 02021-12-30) (10 minutes)

Notes concerning “Galileo”

• Scaling laws (p. 404) 02021-05-19 (updated 02021-12-30)
(8 minutes)
• Against subjectivism (p. 1066) 02021-12-15 (updated 02021-12-30)
(36 minutes)

Notes concerning “Fiction”

• How Lao Yuxi painted a cock (p. 247) 02021-03-19 (updated 02021-04-14) (7 minutes)
• My ideal workshop (unfinished) (p. 591) 02021-07-16 (updated 02021-07-27) (2 minutes)

Notes concerning “Enthalpy”

• Boosters for self-propagating high-temperature synthesis (SHS) (p. 604) 02021-07-17 (updated 02021-12-30) (4 minutes)
• Heating a shower tank with portable TCES? (p. 714) 02021-08-11 (updated 02021-08-15) (6 minutes)

Notes concerning “Employment”

• Intel engineering positions considered as a dollar auction (p. 78)
02021-01-21 (updated 02021-01-27) (1 minute)
• Planning Apples to Apples, instead of Planning Poker (p. 851)
02021-09-29 (updated 02021-12-30) (6 minutes)

Notes concerning “Electropolishing”

• Fresnel mirror electropolishing (p. 377) 02021-05-08 (updated 02021-12-30) (6 minutes)
• Aluminum foil (p. 413) 02021-05-24 (updated 02021-09-11) (14 minutes)

Notes concerning “Electroforming”

• Electroforming rivets (p. 410) 02021-05-22 (updated 02021-12-30) (2 minutes)
• Super ghetto digital fabrication (p. 479) 02021-06-28 (updated 02021-12-30) (33 minutes)

Notes concerning “Dreaming”

• Designing curiosity and dreaming into optimizing systems (p. 420) 02021-05-30 (updated 02021-12-30) (6 minutes)
• Adversarial control (p. 987) 02021-10-25 (updated 02021-12-30) (13 minutes)

Notes concerning “Decentralization”

- PBKDF content addressing with keyphrase hashcash: a non-blockchain attack on Zooko’s Triangle (p. 896) 02021-10-08 (24 minutes)
- Safe decentralized cloud storage (p. 1135) 02021-12-30 (10 minutes)

Notes concerning “Debugging”

• Smolsay: the Ur-Lisp, but with dicts instead of conses (p. 552)
02021-07-12 (updated 02021-07-27) (22 minutes)
• The astounding UI responsivity of PDP-10 DDT on ITS (p. 972)
02021-10-22 (updated 02021-10-23) (28 minutes)

Notes concerning “Databases”

• Trying and failing to design an efficient index for folksonomy data based on BDDs (p. 108) 02021-01-26 (updated 02021-01-27) (7 minutes)
• The algebra of N-ary relations (p. 432) 02021-06-14 (updated 02021-07-27) (4 minutes)

Notes concerning “Cross linking”

• 3-D printing in carbohydrates (p. 393) 02021-05-16 (updated 02021-12-30) (10 minutes)
• 3-D printing in poly(vinyl alcohol) (p. 1080) 02021-12-15 (updated 02021-12-30) (2 minutes)

Notes concerning “Command-line interfaces (CLI)”

• A short list of the most useful Unix CLI tools (p. 841) 02021-09-15 (updated 02021-09-16) (2 minutes)
• DSLs for calculations on dates (p. 1018) 02021-11-14 (updated 02021-12-30) (1 minute)

Notes concerning “China”

• Statistics on the present and future of energy in the People’s Republic of China (p. 316) 02021-04-01 (updated 02021-04-08) (10 minutes)
• New nuclear power in the People’s Republic of China (p. 1007) 02021-11-09 (updated 02021-12-30) (2 minutes)

Notes concerning “Ccn”

• Chat over a content-centric network (p. 55) 02021-01-15 (updated 02021-01-16) (3 minutes)
• PBKDF content addressing with keyphrase hashcash: a non-blockchain attack on Zooko’s Triangle (p. 896) 02021-10-08 (24 minutes)

Notes concerning “Carborundum”

• Can you use stabilized cubic zirconia as an ECM cathode in molten salt? (p. 475) 02021-06-27 (updated 02021-12-30) (3 minutes)
• Super ghetto digital fabrication (p. 479) 02021-06-28 (updated 02021-12-30) (33 minutes)

Notes concerning “Call by name”

• Improvements on C for low-level programming such as block arguments (p. 584) 02021-07-14 (updated 02021-12-30) (8 minutes)
• Compilation of block arguments to high-performance code (p. 624) 02021-07-29 (updated 02021-12-30) (19 minutes)

Notes concerning “Block arguments”

• Improvements on C for low-level programming such as block arguments (p. 584) 02021-07-14 (updated 02021-12-30) (8 minutes)
• Compilation of block arguments to high-performance code (p. 624) 02021-07-29 (updated 02021-12-30) (19 minutes)

Notes concerning “Bicicleta”

• Principled APL redux (p. 22) 02021-01-03 (updated 02021-12-31) (12 minutes)
• Embedding runnable code in text paragraphs for numerical modeling (p. 1002) 02021-11-06 (updated 02021-12-30) (6 minutes)

Notes concerning “Barcodes”

• Constant weight dithering (p. 991) 02021-10-28 (updated 02021-12-30) (5 minutes)
• The user interface potentialities of a barcoded paper notebook (p. 1104) 02021-12-18 (updated 02021-12-30) (6 minutes)

Notes concerning “Azane”

• Methane bag (p. 710) 02021-08-10 (updated 02021-08-15) (8 minutes)
• Blowing agents (p. 847) 02021-09-29 (updated 02021-12-30) (4 minutes)

Notes concerning “AVR8 microcontrollers”

• A compact bytecode sketch that should average about 3 bytes per line of C (p. 720) 02021-08-17 (updated 02021-09-13) (66 minutes)
• Pipelined piece chain painting (p. 926) 02021-10-10 (updated 02021-12-30) (23 minutes)

Notes concerning “Arduino”

• Notes on what would be needed to drive a PS/2 keyboard from an Arduino (p. 447) 02021-06-20 (updated 02021-12-30) (12 minutes)
• A compact bytecode sketch that should average about 3 bytes per line of C (p. 720) 02021-08-17 (updated 02021-09-13) (66 minutes)

Notes concerning “Archival”

• Refreshing Flash memory periodically for archival (p. 198) 02021-03-02 (1 minute)
• Leaf hypertext (p. 380) 02021-05-08 (updated 02021-12-30) (3 minutes)

Notes concerning “Apl”

• Principled APL redux (p. 22) 02021-01-03 (updated 02021-12-31) (12 minutes)
• An algebra of partial functions for interactively composing programs (p. 933) 02021-10-10 (updated 02021-12-30) (3 minutes)

Notes concerning “Ambiq”

• Energy autonomous computing (p. 143) 02021-02-18 (updated 02021-12-30) (58 minutes)
• A compact bytecode sketch that should average about 3 bytes per line of C (p. 720) 02021-08-17 (updated 02021-09-13) (66 minutes)