Urbit

The technical problem

- the high-level deterministic computer
  - official definition
    - entire lifecycle defined by a single frozen function
    - lifecycle semantics defined at the programmer level
  - existing approximations
    - machine VM: inherently low-level
    - JS, JVM, etc: transient, and not quite frozen
    - Lisp, Smalltalk machines: no functional definition
  - user experience: integrated OS/interpreter/DB
  - when a deterministic computer hits an undecidable problem?
- kelvin versioning
  - decreases by integers to absolute zero; Urbit is 5K

The human problem (1)

- the Internet as a client-server network has won
  - (HTTP = ATDT) $\Longrightarrow$ (FB = AOL)
- the Internet as a peer-to-peer network has failed
  - new wide-area protocols can no longer be introduced
    - even SMTP survives only by inertia
  - no one wants to self-host on a personal Linux server
    - Linux is layer 7 of the Internet
  - the Linux/Internet platform is unsalvageable as a P2P network

The human problem (2)

- the old platform is a fine substrate to layer over
  - on the client side, the browser already did this
- there's a plausible need for a new platform
  - we don't know that people don't want personal servers
  - we just know they don't want personal Unix servers on the Internet
- the obstacle to the personal server is admin cost
  - technical simplicity is a plausible therapy
- this "browser for the server side" is a clean slate
  - since we're layering over both the OS and the network

The one-function computer

- the simplest network: a global broadcast ethernet
  - routing is an optimization (content-centric networking)
  - packets are facts; the event log is a list of facts heard
- the simplest computer: a packet transceiver
- two ways to define a one-function computer:
  - lifecycle function: L(input history) → resulting state
  - transition function: T(input event, state) → (output actions, new state)
- any practical L will converge to some T
  - output in a lifecycle function is an optional hint
  - a lifecycle function can define a boot sequence

Practical implementation

- event sourcing is popular these days
  - good tools for low-latency reliable logs (Kafka)
  - snapshot and append-only log is the normal DB design
  - every packet is a transaction, finalized when complete
- non-packet I/O can be event-sourced (libuv)
- decidability is a heuristic problem
  - interrupt a console; time out a packet
  - the log is an existence proof of computability
- nondeterminism feeds back into the event stream

Let’s try it in Lisp

- it’s easy to define a lifecycle function in Lisp
  - (defun lifecycle (log) ((car log) (cdr log)))
    - the first event is the function, the rest of history is the argument
      - of course we still have to write the function…
- now all we need is the one true perfect frozen Lisp
  - a lifecycle function makes extreme demands on interpreter precision
  - probably no “one-stage” interpreter can satisfy these demands
  - two stages: axiomatic untyped VM, user-level typed compiler
- lambda puts high-level features in the axioms
  - symbols, functions, variables, scopes, are all user-level features

The Urbit stack

- Nock: typeless, frozen interpreter
  - defined in 200 words, on a (readable) T-shirt
- Hoon: pure, strict, typed functional language
  - does not use category theory
  - compiles its own compiler to Nock
- Arvo: nonpreemptive operating system
  - written in Hoon
  - defines a transition function T(event, state) -> (actions, new state)
  - defines a referentially transparent global namespace
  - interprets and sends untrusted, unreliable network packets
  - can update everything except Nock over the wire

Nock ideals

- a “functional assembly language”
- a Lisp without high-level affordances
  - symbols, variables, scope, syntax, etc
- no cyclic or infinite data structures
  - acyclic strict data is much easier to persist and transport
- efficient execution strategy
- fits on a T-shirt
- obviously perfect and will never need to change

Nock concepts

- a value in Nock is a *noun*
- a noun is an *atom* or a *cell*
  - an atom is an unsigned integer of any size
  - a cell is an ordered pair of any two nouns
    - cells are strict and acyclic and compare by value
- Nock is a function **[subject formula] → product*
  - the *subject* is the data; the *formula* is the function; the *product* is the result
  - any error produces nontermination, bottom, ⊥

Nock spec (intrinsics)

```
?[a b]              0
?a                  1

+[a b]              +[a b]
+a                  1 + a

=[a a]              0
=[a b]              1
=a                  =a

/[1 a]              a
/[2 a b]            a
/[3 a b]            b
/[(a + a) b]        /[2 /[a b]]
/[(a + a + 1) b]    /[3 /[a b]]
/a                  /a
```

Nock spec (instructions)

```
*[a [b c] d]      [*[a b c] *[a d]]

*[a 0 b]          /[b a]
*[a 1 b]          b
*[a 2 b c]        *[*[a b] *[a c]]
*[a 3 b]          ?*[a b]
*[a 4 b]          +*[a b]
*[a 5 b]          =*[a b]

*[a 6 b c d]      *[a 2 [0 1] 2 [1 c d] [1 0] 2 [1 2 3] [1 0] 4 4 b]
*[a 7 b c]        *[a 2 b 1 c]
*[a 8 b c]        *[a 7 [[7 [0 1] b] 0 1] c]
*[a 9 b c]        *[a 7 c 2 [0 1] 0 b]
*[a 10 [b c] d]   *[a 8 c 7 [0 3] d]
*[a 10 b c]       *[a c]

*a                *a
```

Decrement in Nock + Hoon

```
[8
  [1 0]
  [ 8
    [ 1
      [ 6
        [5 [0 7] [4 0 6]]
        [0 6]
        [9
          2
          [[0 2] [4 0 6] [0 7]]
        ]
      ]
    ]
    [9 2 0 1]
  ]
]
```

```
:per   a=.
:pin   b=0
:loop
:if  =(a +(b))
  b
:moar(b +(b))
```

```
=>  a=.
=+  b=0
|-
?:  =(a +(b))
  b
$(b +(b))
```

Nock optimization

- O(n) decrement is nifty, but not practical
- solution: a sufficiently smart interpreter
  - just recognize any decrement formula, and execute it efficiently
    - also add, multiply, and all other common intrinsics
- wait, we don’t have to recognize *every* decrement
  - just the decrement in the standard library
- solution: hint-register and match important functions
  - these *jets* are like software device drivers
  - user code remains pure, but declares a semantic identity
  - an interpreter which recognizes this identity can optimize it

Hoon ideals

- compiler compiles itself from source to Nock
  - does *not* have to be frozen; anything above Nock must self-update
- pure, strict, higher-order typed functional language
- transformation to Nock is simple, like C to assembly
- requires no particular mathematical aptitude
  - and does not use category theory
- encourages lower-order, more imperative style
  - DSLs considered harmful
- still almost as expressive as Haskell

Hoon concepts

- Hoon is a typed macro assembler for Nock
  - type inference and code generation combined are 1.500 lines of Hoon
- Computes [subject expression] → product
  - “subject-oriented programming”
- the Hoon compiler is a parser and a generator
  - the parser (vast) computes source → expression
  - the generator (ut) computes [type expression] → [type formula]
    - its input is the subject type (domain) and the expression
    - its output is the product type (range) and the Nock formula
- Hoon infers only forward, without unification
  - a (slightly) less intelligent inference algorithm is a UI win
  - it’s good to coerce the product of any entry point, just for doc reasons

Never say type

- Learn the secret language of Hoon
  - an expression (AST noun) is a twig
  - a type (as in set of nouns) is a span
  - a type (as in constructor/declaration) is a mold
  - a type (as in MIME type) is a mark
- Hoon is a pure prototype language
  - there is no syntax for a span; it is only defined as the range of a computation
- A mold is a normalizing function on an arbitrary noun
  - a true mold is idempotent, so =((mold x) (mold (mold x)))
  - the only time we actually *call* a mold is to validate network data

Basic span concepts

- twigs are boring once you understand spans
- mold for a slightly simplified span:

```
++  span
$@  $?  $noun
        $void
==  $%  {$atom p/term q/(unit atom)}
        {$cell p/span q/span}
        {$core p/span q/(map term span)}
        {$face p/term q/span}
        {$fork p/(set span)}
        {$hold p/span q/twig}
    ==
```

- missing only: aliases, variance and typeclasses

Unboring spans: `$atom`

- `{$atom p/term q/(unit atom))}`
  - if q is set, u.q is the only atom in the range (constant)
  - p is an aura, a symbol which describes units/presentation/constraint
    - auras are unenforced conventions
    - auras specialize by extension right
      - @t for UTF-8 text, @ta ASCII, @tas ASCII with symbol constraint
      - @s for signed integer, @sx for signed hexadecimal integer
    - auras can be cast upward or downward, but not across

Unboring spans: `$core`

- `{$core p/span q/(map term twig)}`
  - a core is a [battery payload] cell
  - p is the span of the payload
  - q is a table of computed attributes, or *arms*
    - the battery is the tree of the arm formulas
    - the subject of each arm is the core
  - single namespace searches battery first, then payload
- a core is the general case of functions and objects
  - a function in Hoon is a gate: a special case of core
    - a gate has one nameless arm ($) and a payload [sample context]
  - a method in Hoon is an arm which produces a gate
  - an object in Hoon is a core whose arms are all methods

Advanced span theory

- the `$hold` form is manual laziness
  - but also lets conservative worklist algorithms prune recurrent traverses
- branch conditions are analyzed for type inference
  - this allows classic functional pattern matching
- two kinds of polymorphism: variance and genericity
  - polymorphism is about compatibility of mutated cores (Liskov substitution)
  - variance: does mutant payload match original payload?
    - collect all four: covariance, contravariance, invariance, bivariance
  - genericity: does battery formula work with mutant payload?
    - generic (wet) arms expand inline like macros

Syntax design

- Hoon syntax is twig syntax
- twig syntax is functionally complex and looks gnarly
  - but everyone who learns it is surprised at how easy it was
- Hoon solves three problems with functional syntax
  - expressions grow downward and to the right
    - solution: backstep indentation
  - either terminator piles or significant whitespace
    - solution: self-terminating form
  - hard to distinguish special forms from symbols
    - solution: no macros, marked keywords / runes

Twig structure

- a twig is the AST expression
  - which compiles to a Nock formula, which defines a function of the subject
- any cell [twig twig] is also a twig
  - which constructs the cell of its subtrees, like Lisp cons
  - Hoon is tuple-centric, not list-centric, because types work
- any other twig is a tagged union, [stem bulb]
  - the head of the twig is an atom, its stem
  - the stem is a 2-4 byte term (ASCII symbol)
  - the shape of the tail depends on the stem
    - but usually a tuple or list of twigs

Regular forms, flat + tall

- every stem defines its own bulb
  - most are 1-ary, 2-ary, 3-ary or 4-ary tuples; some are n-ary lists
  - every stem with a tuple/list bulb has a regular form
- regular forms come in two forms: *flat* and *tall*
  - a flat twig is delimited by parens, and separates subtwigs by one space:

```
:if(a b c)
```

  - a tall twig has no delimiters, and separates subtwigs by 2+ spaces:

```
:if  a
  b
c
```

- goal: look like a procedural expression-statement mix

Runes and irregular forms

- irregular form: always flat, always ASCII, and always a twig
- a regular form can use a keyword or a *rune*
  - keyword is colon-prefixed: :if(a b c)
  - rune is a digraph: ?:(a b c)
    - first character defines role (eg, all ? runes are conditionals)
- ASCII reloaded:

```
ace [1 space]   gal <                 pal (
bar |           gap [>1 space, nl]    par )
bas \           gar >                 sel [
buc $           hax #                 sem ;
cab _           hep -                 ser ]
cen %           kel {                 sig ~
col :           ker }                 soq '
com ,           ket ^                 tar *
doq "           lus +                 tec `
dot .           pam &                 tis =
fas /           pat @                 wut ?
zap !
```

Two forms of FizzBuzz

```
:gate  end/atom
:var   count  1
:loop
:cast  (list tape)
:if  =(end count)
   ~
:cons
  :if  =(0 (mod count 15))
    "FizzBuzz"
  :if  =(0 (mod count 5))
    "Fizz"
  :if  =(0 (mod count 3))
    "Buzz"
  "{<count>}"
:moar(count (add 1 count))
```

```
|=  end/atom
=/  count  1
|-  ^-  (list tape)
?:  =(end count)
    ~
:_  $(count (add 1 count))
?:  =(0 (mod count 15))
  "FizzBuzz"
?:  =(0 (mod count 5))
  "Fizz"
?:  =(0 (mod count 3))
  "Buzz"
"{<count>}"
```

Arvo concepts

- the Arvo kernel is a Hoon core
  - with a fixed battery exporting a few arms
  - including a transition function T(event) → (actions, new core)
  - the kernel ABI is frozen at the Nock level
    - so any event can replace Arvo or even Hoon
    - as long as it can build a core that looks the same to Nock
- Arvo proper is ~600 lines
  - internal event cascade with causal stack
  - global typed referentially transparent namespace
  - load and reload kernel modules (vanes), like baby kernels

Arvo vanes (kernel modules)

- %ames, encrypted packet network
- %behn, timers
- %clay, revision-controlled typed filesystem
- %dill, console
- %eyre, HTTP client/server with reactive apps
- %ford, functional build system
- %gall, applications

Urbit identity concepts

- “Urbit” just means the Nock/Hoon/Arvo stack
  - and more specifically the PKI / identity model
  - “an urbit” means one event history / state / instance/ node, with one identity
    - identity creation is part of the boot process
    - the identity is a ship, the instance is a *pier*
- one system solves:
  - human-memorable cryptographic identity
  - P2P packet routing address
  - base of global immutable namespace path
- assault on Zooko’s triangle
  - trivial solution: identity is 128-bit public-key hash
  - tradeoff: memorable, but not meaningful

A civil address space

- phonemic base-256 makes numbers memorable
  - 0x802a.136d in @ux; .128.42.19.109 in @if; ~patnub-tarlud in @p
- a 128-bit “wild” ship is called a comet
  - but shorter numbers are more memorable, hence more valuable
- a 64-bit “civil” address space can overload it
  - 64-bit ship: *moon*; ~padfes-sopden-difmyl-padtul; connected device
  - 32-bit ship: *planet*; -difmyl-padtul; human being
  - 16-bit ship: *star*; ~mocryg; minor infrastructure
  - 8-bit ship: *galaxy*; ~num; major infrastructure
- each tier is initially signed by its half-width parent
  - but signs its own key update; renewal is revocation
  - ships can be traded like bitcoin, but low-frequency “spends” don’t need a blockchain
  - galaxies are “premined” in the kernel source

On purpose and hindsight

- but was it really necessary to invent all this crap?
- a personal server is a social server
  - when two humans socialize, they should exchange messages directly
  - and not fall back into the degenerate case of a central server
- centralized programming is always easier
  - in today's infrastructure it's orders of magnitude easier
- the criterion: *difficulty of distributed programming*
- a true personal server must solve this problem
  - some impersonal servers wouldn't mind as well

Programmer experience

- dereference the global immutable typed namespace
- application state is permanent, no database required
- update source code propagates reactively via revision control
    - application type changes require typed state adapters
- two messaging patterns, poke and peer
    - no abstraction leakage versus local communication
    - %eyre lets web clients poke and peer over HTTP
- a poke is a typed, transactional message
    - exactly-once delivery (even though it's impossible)
    - no return data if transaction succeeds
    - message is automatically validated
        - backward-compatible protocol updates cause no errors in a live network
    - passed to the receiving program as a typed event
    - end-to-end acknowledgments mean single error mode
    - sender queues until message delivered or rejected
    - P2P network delivers anywhere
    - authenticated and encrypted, of course
- a peer creates a subscription which streams typed diffs

System status

- 30,000 lines of Hoon (Hoon, Arvo, all vanes, and basic apps)
- open source and patent-free
- urbit.org
  - served by Urbit (behind nginx :-)
- github.com/urbit
- somewhere between alpha and beta quality
  - small live network, “used in anger”, working shell and chat apps
  - Arvo internal interfaces mostly stable
  - global flag day (“continuity breach”) every few months
- documentation historically sucks, but getting better
  - Hoon documentation is now adequate
  - next phase is Arvo documentation
- ready for self-hosted address-space crowdsale