Urbit: A Solid-State Interpreter

Curtis Yarvin, Philip Monk, Anton Dyudin, Raymond Pasco
Tlon Corporation

May 26, 2016

Abstract

A "solid-state interpreter" (SSI) is an interpreter which is also an ACID database. An "operating function" (OF) is a general-purpose OS defined as a pure function of its input stream. Urbit is an SSI defined as an OF. Functionally, Urbit is a full execution stack. VM, compiler, OS, network, web server, and core apps are 30kloc; the system is alpha-grade with a live, fairly stable network.

1 Introduction

What is Urbit? We can answer six ways: functionally, formally, mechanically, productively, securely and practically.

1.1 Functionally

Urbit is a new clean-slate system software stack. A nonpreemptive OS (Arvo), written in a strict, typed functional language (Hoon) which compiles itself to a combinator VM (Nock), drives an encrypted packet network (Ames) and defines a global version-control system (Clay). Including basic apps, the whole stack is about 30,000 lines of Hoon.

1.2 Formally

Urbit is an "operating function": a general-purpose OS whose entire lifecycle is defined as a pure, frozen, small function on its input stream. The function is an interpreter; it compiles an initial language, which boots an initial kernel. Everything but the lifecycle function can upgrade itself from source code in the input stream.

1.3 Structurally

Urbit is a "solid-state interpreter": an interpreter with no transient state. The interpreter is an ACID database; an event is a transaction in a log-checkpoint system. The Urbit interpreter runs on a normal Unix server; it interacts with

the Unix user, on the console or via FUSE; with other Urbit nodes, in its own protocol over UDP; with the internets, as an HTTP client or server.

1.4 Securely

Security problems are an inevitable consequence of unrigorous and/or complex computing semantics. Rigor and simplicity are hard properties to enforce retrospectively on the classical Unix/IETF/W3C stack.

A new stack, designed as a unit, learning from all the mistakes of 20th-century software and repeating none of them, should be simpler and more rigorous than what we use now. Otherwise, why bother?

One example of painful heterogeneity inherited from the current stack: the relationship between a database schema, a MIME type, a network protocol, and a data structure. The same semantics may be reimplemented incompatibly at four layers of this stack. The programmer is constantly hand-translating values. Helpful tools abound ("ORM is the Vietnam of computer science" [17]), but no tool can fix architectural impedance mismatches. There are always cracks; bugs and attackers slip between them.

1.5 Productively

Urbit is designed to work as a "personal server." This is not yet an established product category, so let's explain it.

Everyone has a personal client, a secondary kernel isolated from the OS: the browser. Almost no one has a personal server, but they should. Urbit is a "browser for the server side"; it replaces multiple developer-hosted web services on multiple foreign servers, with multiple self-hosted applications on one personal server.

Why attack this problem with a clean-slate stack? Arguably, what stops people from running their own servers is the usability challenge of administering a Unix box on the public Internet. The Unix/Internet platform is 40 years old and has built up a lot of complexity. Mechanical simplicity does not guarantee humane usability, but it's a good start.

A clean-slate personal server, with its own encrypted packet overlay network, can forget that Unix and the Internet exist. It can run on Linux and send packets over UDP; it can both provide and consume HTTP services; but its native stack is not affected by 20th-century design decisions.

HTTP services? Since a new network has no network effect, a node must find its first niche as a client and/or server on the old network. For its earliest users, Urbit has two main roles: as an HTTP client, a personal API aggregator (which organizes and controls your own data on your existing cloud accounts); (2) as an HTTP server, personal publishing and identity management (blogging, archiving, SSO, etc). But Urbit still differs from existing Web application platforms in that an instance is designed to serve one primary user for its whole lifespan.

1.6 Practically

Urbit is working code in semi-closed alpha test. It self-hosts, maintains a small live network, runs a distributed chat service, and serves its own website (`urbit.org`). Urbit is MIT licensed and patent-free.

Urbit is not yet secure or reliable. It's not yet useful for anything, except developing Urbit.

This paper is a work-in-progress presentation, not a final report. It contains numerous small errata relative to the running codebase. We'd rather describe what the finished Urbit will do than what an incomplete version happens to do now.

2 Highlights and novelties

Anything that calls itself a clean-slate stack should be indistinguishable from a boot ROM from a crashed alien spaceship. Sometimes this is impractical; the aliens probably wouldn't use ASCII (let alone Unicode), AES, etc. (There's a fine line between a metaphor and an actual *hoax*.) And some designs are genuinely universal; an alien linked list is a linked list.

(Urbit also invents a lot of alien technical jargon. We admit that this is annoying. Abstractly, it's is part of the clean-slate program: words smuggle in a lot of assumptions. Concretely, changing the words does indeed change the concepts; For instance, we say *type* informally to mean any one of three quite different concepts: `span`, `mold`, or `mark`.)

Before we review the design more systematically, let's take a quick tour of Urbit's more unusual architectural features, working upward through the layers covered in this paper (Nock, Hoon, Arvo and Ames).

2.0.1 Nock (combinator interpreter, page 11)

Nock, a non-lambda combinator interpreter, is Urbit's VM and axiomatic definition. Its informal spec is 340 bytes compressed. Nock is nowhere near the simplest possible definition of computing, but it's quite simple for a practical one. Nock is permanently frozen and will never need updating.

A Nock value, or "noun", is an unsigned integer of any size, or an ordered pair of any two nouns. Nock is homoiconic, like Lisp; a Nock function is a noun. Nock cannot create cycles, so nouns can be managed without a tracing garbage collector.

Nock is a little like Lisp, if Lisp was only a compiler target and not a high-level language. As an axiomatic model of computing, Lisp and/or lambda are an imperfect fit, because the reduction axioms define high-level programming concepts such as variables and functions, which (a) should not be coupled to the lower layer, and (b) are properly human-interface features belonging in the upper layer. Nock defines no variables, environments, functions, etc; it doesn't even have a syntax.

Nock's only intrinsic integer operation is increment. Fancier functions are accelerated not by escaping to an intrinsic, extension, or native method, but by registering and recognizing specific Nock code blocks in the standard library, then matching their semantics with custom interpreter modules or "jets." A jet may be implemented in native code or even custom hardware; it can have no side effects, since all it computes is one special case of the Nock function; it is easily tested against the pure code. A more advanced system might use actual equivalence proofs.

2.0.2 Hoon (typed functional language, page 15)

Hoon is a strict, higher-order typed functional language which compiles itself to Nock. Nock acts as a sort of "functional assembly language." The Hoon-to-Nock transformation is designed to be minimal, like the C-to-assembly transformation.

Users often complain that the mathematical abstractions beneath functional languages are difficult to learn and master. Hoon is a low-tech FP language which does not use category theory, lambda calculus or unification inference. The Hoon style discourages aggressive abstraction and promotes a concrete, almost imperative style.

Hoon inference works forward only; it cannot infer input type from output type. The inference algorithm uses manual laziness and a conservative work-list algorithm, which terminates except in case of epic screwup, and even then is easily traced. Hoon's capacity for expressing polymorphism (variance and genericity) approaches that of Haskell, though the underlying mechanisms are quite different. The full back end (type inference plus code generation) is about 1500 lines of Hoon.

A Hoon data structure is defined as an idempotent function that normalizes an arbitrary noun. Therefore, defining a structure implies defining a protocol validator for untrusted network data. Types, formats, protocols, and schemas are all the same thing.

Hoon's syntax is also unusual, and *not* simple. Hoon works hard to distinguish intrinsic operators from symbols and functions. To avoid write-only code, there are no user-defined macros or other gestures in the DSL direction.

Whitespace in Hoon is not significant, as in Haskell. There are no piles of parentheses or other right terminators, as in Lisp. Hoon treats indentation as a scarce resource; well-indented Hoon with a deep AST flows down the screen, not down and to the right as in Lisp.

2.0.3 Arvo (nonpreemptive kernel, page 23)

The formal state of an Arvo instance is an event history, as a linked list of nouns from first to last. The history starts with a bootstrap sequence that delivers Arvo itself, first as an inscrutable kernel, then as the self-compiling source for that kernel. After booting, we break symmetry by delivering identity and entropy. The rest of the log is actual input.

The only semantics outside the event history are Nock itself, plus a "functional BIOS": the noun `[2 [0 3] [0 2]]`. This noun is a Nock function; it calls the first event as a Nock function on the rest of history. The result of this function is the current operating state, a noun.

This noun is the Arvo kernel, an object with one method which is the main event loop. Arvo proper is not large: the kernel proper is only about 600 lines of Hoon. Most system semantics are in kernel modules ("vanes"), loaded as source. User-level code runs either at a third layer within the sandbox vane `%gall`, or a fourth layer within the shell (`:dojo`) application.

A vanilla event loop scales poorly in complexity. A system event is the trigger for a cascade of internal events; each event can schedule any number of future events. This easily degenerates into "event spaghetti." Arvo has "structured events"; it imposes a stack discipline on event causality, much like imposing subroutine structure on gotos.

User-level code is virtualized within a Nock interpreter written in Hoon (with zero virtualization overhead, thanks to a jet). Arvo defines a typed, global, referentially transparent namespace with the Ames network identity (see below) at the root of the path. User-level code has an extended Nock operator that dereferences this namespace and blocks until results are available. So the Hoon programmer can use any data in the Urbit universe as a typed constant.

Most Urbit data is in Clay, a distributed revision-control vane. Clay is like a typed `git`. If we know the format of an object, we can do a much better job of diffing and patching files. Clay is also good at subscribing to updates and maintaining one-way or two-way automatic synchronization.

Nock is frozen, but Arvo can hotpatch any other semantics at any layer in the system (apps, vanes, Arvo or Hoon itself) with automatic over-the-air updates. Updated code is loaded from Clay and triggered by Clay updates. If the type of live state changes, the new code must supply a function from old to new state.

As a "solid-state interpreter", Arvo is designed to replace both the interpreter and the database in the modern server execution stack. Its event log is its transaction log. It persists by saving dirty pages to a snapshot and/or replaying log entries.

Urbit is built with the explicit assumption that a perfectly reliable, low-latency, high-bandwidth log in the cloud is a service we know how to deliver. The log can be pruned at a reliable checkpoint, however.

A nonpreemptive OS remains an OS. Every OS needs interrupts; a preemptive OS is interrupted every 20ms; a nonpreemptive OS is interrupted, whether by hand or by a heuristic timer, every time it seems to be taking too long. Interrupted calculations are either abandoned or replaced with an error. Since only completed events are logged, log replay terminates by definition.

2.0.4 Ames (P2P packet protocol, page 26)

Ames is an encrypted peer-to-peer network running as an overlay over UDP. Ames does not have separate addressing and identity layers (like IP and DNS).

An Ames address is an identity, mapped to a phonemic string to create a memorable pseudonym, and bound to a public key for encrypted communication.

Ames addresses are 128-bit atoms. Addresses above 64-bit are hashes of a public key. 64, 32, and 16-bit addresses are initially signed by their half-width prefix. Initial fingerprints of 8-bit addresses are hardcoded. 8, 16, and 32-bit addresses sign their own key updates and are "sovereign." Prefix servers are also supernodes for P2P routing.

Ames delivers messages of arbitrary length, broken into MTU-shaped fragments. Because Urbit nodes are uniformly persistent, they maintain persistent sessions; message delivery is exactly-once. Every message is a transaction, and acknowledgments are end-to-end; the packet-level acknowledgment that completes a message also reports transaction success. A successful transaction has no result; a failed transaction is a negative ack and can contain an error dump.

Ames messages are typed; the type itself is not sent, just a label (like a MIME type) that the recipient must map to a local source path. Validation failure causes a silent packet drop, because its normal cause is a recipient that has not yet received a new protocol update; we want the sender to back off. Ames also silently drops packets for encryption failure; error reports are just an attack channel.

3 The solid-state interpreter

Let's design a system software stack from scratch. We'll call our design a *solid-state interpreter*, or *SSI*. Urbit is one of many possible SSIs.

Briefly, an SSI is an interpreter which is also a database. In more depth, an SSI combines three properties: *uniform persistence*, *source-independent packet networking*, and *high-level determinism*.

3.1 Uniform persistence

Storage in an SSI is flat and *uniformly persistent*. "Uniform persistence" is a degenerate case of *orthogonal persistence* [10]: there is no separation into "memory" which is fast but volatile, and "disk" which is slow and stable. The system's state is one stable data structure.

A hardware implementation of uniform persistence could use emerging technologies such as NVDIMMS [3]. Today, Urbit runs on normal hardware, using a log-snapshot mechanism like a normal database. Some low-latency stable storage (even just an SSD) remains highly desirable.

A kernel which presents the abstraction of a single layer of permanent state is also called a *single-level store* [12]. One way to describe a single-level store is that it *never reboots*; a formal model of the system does not contain an operation which unpredictably erases half its brain.

(It's especially unforgivable to expose this weird split-brain design to the user. For instance, modern mobile OSes try to to isolate the user from the stateful nature of app instantiation and termination, but never quite succeed.)

3.2 Source-independent packet networking

A pure SSI has no I/O except unreliable, insecure packet networking. Why isn't HDMI just an Ethernet cable? USB? PCI? History and not much else.

This is not just a connector issue. There is a categorical difference between a *bus*, which transports *commands*, and a *network*, which transports *packets*. You can drop a packet but not a command; a packet is a *fact* and a command is an *order*. To send commands over a wire is unforgivable: you've turned your network into a bus. Buses are great, but networks are magic.

Facts are inherently idempotent; learning fact X twice is the same as learning it once. You can drop a packet, because you can ignore a fact. Orders are inherently sequential; if you get two commands to do thing X, you do thing X twice.

It's possible to layer a command bus (TCP) on top of a packet network (IP), but the abstraction leaks. If the socket closes, has the remote endpoint received my last command? This is a case of the unsolvable "two generals" problem [2]. Should a systems layer punt unsolvable problem up to the developer?

Another IP mistake is to interpret a packet as a function of its routing topology. A source address is interesting *metadata*, but it is not *data*. Fact X means fact X, whoever you heard it from. Packet semantics should be *source-independent* (aka *content-centric* [4]).

Logically, we could imagine the ideal network as a global broadcast Ethernet. Everyone hears every packet; every packet is signed and encrypted. If you have the correct keys to authenticate and decode a packet, you learn from it; otherwise, you discard it. Semantics are independent of topology.

3.3 High-level determinism

Computation in an SSI is defined as a high-level deterministic interpreter. Any computer worthy of the name is deterministic, but its determinism is normally defined as a CPU step function on a memory/register state.

If you interrupt an OS defined as a CPU and a memory image, you produce undefined semantic state. At the high level, such a machine is effectively nondeterministic.

High-level determinism is determinism at the semantic level. The operating computation is a function call, not a clock cycle. The system state is a data structure, not a memory image.

3.4 Decidability, determinism and interruption

One LangSec technique is to reduce the expressiveness of a component (such as an input recognizer) to render it more rigorous and tractable. Urbit does not use this trick; it has no sub-Turing calculation model.

Can a solid-state interpreter (or any system) be both Turing-complete, and deterministic/repeatable? This may seem theoretically impossible, but it's actually quite practical.

Any practical nonpreemptive OS is interruptible. In exchange for general simplicity across the whole system, the cooperative approach to multitasking pushes the problem of identifying and managing long-running computations up to the programmer. A complex computation can be divided by the programmer into fine-grained subtasks; it can be handled as fake I/O on a "worker thread" (a feature Urbit doesn't have but should); it can just be a tolerated unavailability period.

As CPUs get faster, cooperative multitasking becomes more practical, because the length of tolerated unavailability is computed on a human scale; this is why we see `node.js` returning to the supposedly obsolete design of System 7 and Windows 3.1.

But programmers screw up, and an infinite loop is a common bug. Any nonpreemptive OS needs a way to interrupt a wayward event. This is easy when the event is a human-generated console event: give the user a secure attention key (eg, control-C).

For all other events, a timer is required. Most events are packets, and network stacks have a lot of timers already. A timer on network packet processing should be seen as a network element; it detects congestion at the CPU level, and drops the packet. To the network, an infinite loop is indistinguishable from a cable cut.

Most important, an interrupted event *never happened*. The computer is deterministic; an event is a transaction; the event log is a log of successful transactions. In a sense, replaying this log is *not* Turing-complete. The log is an existence proof that every event within it terminates.

(Of course, a packet that times out remains a DoS attack. Because Ames addresses are scarce, DoS attacks via authenticated packets should be relatively easy to track and throttle.)

3.5 Solid-state interpreter: intuitive description

We call an execution platform with these three properties (universal persistence, source-independent packet I/O, and high-level determinism) a *solid-state interpreter* (SSI).

A solid-state interpreter is a stateful packet transceiver. Imagine it as a chip. Plug this chip into power and network; packets go in and out, sometimes changing its state. The chip never loses data and has no concept of a reboot; every packet is an ACID transaction.

A practical SSI, though in a philosophical sense an OS, actually runs as a Unix process. (At some point a move down to the hypervisor might be advisable.) For low-latency determinism, it logs its event stream to a Raft/Paxos cluster. So that every restore doesn't involve replaying all its history, it periodically flushes dirty pages to a snapshot image. (This log-and-snapshot pattern is the way most modern databases work.)

A deterministic interpreter can't call out to the OS it's running on; it has to be isolated by definition. A new layer in the execution stack, isolated from the legacy OS, has proven useful on the client side: we call it a browser.

4 The operating function

An *operating function* (*OF*) is a single frozen function that defines the complete semantics of a computer. Not every SSI (as defined above) is an OF, but Urbit is. Specifically, the Urbit operating function is a *lifecycle function*, which maps input history to current state.

4.1 Lifecycle function

Any SSI needs to define its semantics as a pure function. The obvious way to define a stateful I/O transceiver is a transition function:

$$T : (State, Input) \rightarrow (State, Output)$$

Again, T must be frozen; if we want to change the computer's semantics, it has to be by changing the state.

While this transition function will exist in practice, it's not the right way to define our interpreter in theory.

First, output doesn't belong in the most fundamental definition of the interpreter; output is optional. The system can't tell whether its outgoing packets are dropped or delivered. Output is in a semantic sense a *hint*; the transceiver is not ordering a packet to be sent; it is *suggesting* a packet that *might* be sent by the runtime system.

Second, since the interpreter's semantics are frozen, we might as well take advantage of this to define not just the transition function, but the *entire system lifespan*, as a pure function. The lifecycle function

$$L : History \rightarrow State$$

is a more general and useful definition.

In practice, any realistic lifecycle function L will have a transition function T somewhere inside it, but the relationship may be complex.

4.2 Larval stage

We have two problems in designing our lifecycle function L. One, we need to *break symmetry*. Even if our network is (logically) a global party line, and everyone hears every packet, not everyone has the keys to decode every packet.

The entropy and/or secrets that establish a node's identity and break symmetry need to be delivered as packets. But they can't be delivered over the party line; then they wouldn't be secrets. Our model seems too simple.

There's also a tension between our desire for minimal semantics and the actual complexity of decoding and interpreting packets. Since we're defining the entire interpreter as a frozen function, which is supposed to be a general-purpose computer, it should be very obvious that this function never needs updating. So its definition should be very small and simple – ideally, fitting (readably) on a T-shirt.

But there's simply no way a small simple function can decode and interpret packets. For a packet transceiver, this is the entire functionality of the OS. We can imagine a very small OS in a few thousand lines of code, but this is still much too big for a plausible solid-state lifecycle function.

Rather, we'd prefer to keep the lifecycle function as a trivial interpreter, and deliver the higher levels of the stack (such as a high-level language, an OS, etc) in the packet stream. When we combine high-level code delivery with symmetry breaking, we have two problems that both suggest the same solution: a larval stage.

Before we plug the newborn node into the network, we feed it a series of bootstrap or "larval" packets that prepare it for adult life as a packet transceiver on the public network. The larval sequence is private, solving the secret delivery problem, and can contain as much code as we like.

But isn't this cheating? If the larval sequence is essential, isn't the true definition of the lifecycle function not L proper, but L plus the larval sequence?

It's cheating if and only if the code we deliver in the larval stage can't be updated by later packets in the adult stage. Unless it's so perfect it can be frozen forever, a solid-state interpreter is inherently dependent on remote updates. Another way to describe its invariant is that L is the only *non-updatable* semantics.

Any component above the L level that can be frozen should be; we certainly strive for this ideal. But eliminating the need for a trusted update server is a very long-term goal. Therefore, everything besides L must be remotely updatable.

Symmetry breaking – the event that defines the identity of the computer – is exempt from this requirement. Once identity is established, it can't be updated. If you want a new identity, create a new instance.

4.3 Step model

Finally, let's relax the criterion that all events are packets. We'll also let the host OS send true commands. We'll call this quantum of input, packet or command, a `step`.

A packet can be trivially dropped; a command has to be handled (it has no default semantics). Also, a packet is a bitstring; a command is a symbolic data structure. Intuitively, a packet is a fact and a command is an order.

Commands let the interpreter talk to the host OS. The interpreter is still isolated from the OS, but this just means their semantic definitions are not entangled.

An isolated functional interpreter can't call OS APIs in the usual impure fashion. It can process OS commands. In response to these commands, it can suggest actions. Since output is out of scope for L, the real T can do anything. And most real-world OS services can be adapted into this step-action pattern.

When step processing fails or is interrupted, we can either discard the step or replace it with an error command. For instance, it's very hard to know where code is spinning without a stack trace. This is nondeterministic information by definition. However, one synonym for "nondeterministic information" is "input."

We simply write our step history with the crash report, not the original step; if we replay, we replay the crash report.

5 The Urbit lifecycle function

Now we're ready to outline Urbit's solution to the lifecycle function problem. Let's define L and outline the initial semantics produced by the larval stage, also describing the transition function 'T'.

5.1 Nouns

A value in Urbit 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.

Nouns are comparable to Lisp S-expressions [8], but are simpler. Sexprs are designed for direct use in a typeless language; nouns are designed as a substrate for a statically typed language at a higher level. S-expressions need dynamic type flags on atoms (for instance, the "S" stands for "symbol"), because you can't print an atom without knowing whether it's an integer, a string, a float, etc. If a static type system is expected, these flags are just cruft.

For similar reasons, there is no one noun syntax. However, we normally write cells in brackets `[a b]` that associate right: `[a b c d]` means `[a [b [c d]]]`. Note that `[a b]` is not Lisp `(a b)`, but rather `(cons a b)` – that is, it's a cons cell, not a list. Noun conventions favor tuples or "improper lists," another choice that makes more sense in a typed environment.

Real-world S-expression systems tend to be leaky abstractions, supporting mutation, cycles, pointer equality tests, etc. The noun abstraction is opaque; for example, equality is a full tree comparison. For fast comparisons and associative containers, we do compute a 31-bit lazy Merkle hash (Murmur3 [1]) on all indirect nouns, but this remains semantically opaque.

5.2 Nock

You can think of Nock as a "functional assembly language." It's a simple combinator interpreter defined by the spec in Listing 1.

The spec is a system of reduction rules. As appropriate for a system of axioms, it's written in pseudocode.

Variables match any noun. Rules closer to the top match with higher priority. A rule that reduces to itself is an infinite loop, which is semantically equivalent to a crash – they're both $\perp$. (Of course, a practical interpreter reports this crash rather than enacting it.)

$Nock(a)$ is the full reduction of $*a$, where a is any noun.

5.2.1 Operators

The pseudocode notation defines five prefix operators: `?`, `+`, `=`, `/`, and `*`.

`?[x]`, the depth test operator, produces `0` if `x` is a cell, `1` if it is an atom.

Listing 1: Nock

```
?[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

*[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
```

`+[x]`, the increment operator, produces the atom `x` + 1, crashing if `x` is a cell.

`=[x y]`, the comparison operator, produces `0` if `x` and `y` are the same noun, `1` otherwise. `=x`, where `x` is an atom, crashes.

`/` is a tree addressing operator. The root of the tree is `1`; the left child of any node `n` is `2n`; the right child is `2n+1`. `/[x y]` is the subtree of `y` at address `x`. 0 is not a valid address, and any attempt to take the child of an atom crashes.

(For instance, `/[1 [541 25 99]]` is `[541 25 99]`; `/[2 [541 25 99]]` is `541`; `/[3 [541 25 99]]` is `[25 99]`; `/[6 [541 25 99]]` is `25`; `/[12 [541 25 99]]` crashes.)

`*[x y]` is the Nock interpreter function. `x`, called the *subject*, is the data. `y`, called the *formula*, is the code.

5.2.2 Instructions

A valid formula is always a cell. If the head of the formula is a cell, Nock treats both head and tail as formulas, resolves each against the subject, and produces the cell of their products. In other words, the Lisp program `(cons x y)` becomes the Nock formula `[x y]`.

If the head of the formula is an atom, it's an instruction from `0` to `10`.

A formula `[0 b]` produces the noun at tree address `b` in the subject.

A formula `[1 b]` produces the constant noun `b`.

A formula `[2 b c]` treats `b` and `c` is formulas, resolves each against the subject, then computes Nock again with the product of `b` as the subject, `c` as the formula. Without `2`, Nock is not Turing-complete.

In formulas `[3 b]`, `[4 b]`, and `[5 b]`, `b` is another formula, whose product against the subject becomes the input to an axiomatic operator. `3` is `?`, `4` is `+`, `5` is `=`.

Instructions `6` through `10` are macros; deleting them from Nock would decrease compactness, but not expressiveness.

`[6 b c d]` is if `b`, then `c`, else `d`. Each of `b`, `c`, `d` is a formula against the subject. In `b`, `0` is true and `1` is false.

`[7 b c]` composes the formulas `b` and `c`.

`[8 b c]` produces the product of formula `c`, against a subject whose head is the product of formula `b` with the original subject, and whose tail is the original subject. (Think of `8` as a "variable declaration" or "stack push.")

`[9 b c]` computes the product of formula `c` with the current subject; from that product `d` it extracts a formula `e` at tree address `b`, then computes `*[d e]`. (Think, very roughly, of `d` as an object and `e` as a method.)

`[10 b c]` is a hint semantically equivalent to the formula `c`. If `b` is an atom, it's a static hint, which is just discarded. If `b` is a cell, it's a dynamic hint; the head of `b` is discarded, and the tail of `b` is executed as a formula against the current subject; the product of this is discarded.

A practical interpreter can do anything with discarded data, so long as the result it computes complies with the semantics of Nock. For example, the

simplest use of hints is the ordinary "debug printf." It is erroneous to skip computing dynamic hints; they might crash.

5.2.3 Motivation

Nock is Lisp without upholstery. All user-level features have been stripped for use as a functional assembly language: syntax, variables, even scope and functions. Tree addressing replaces the whole architecture of free and bound variables. There is nothing like a gensym. The Lisp environment, a rare exception to Lisp's homoiconicity (except in very naive Lisps, the environment is not a user-level data structure), is replaced by the Nock subject, a ordinary noun.

Other typed functional languages (such as Tarver's Qi [13] and Shen [14]) have used unmodified Lisps as a compilation target; this is a great strategy for maturity, but it leaves significant simplicity gains uncaptured.

5.2.4 Optimization

The reader may have noticed one problem with Nock: its only arithmetic operation is increment.

How can an interpreter whose only arithmetic operator is increment compute efficiently? For example, the only way to decrement n is to count up to $n-1$.

Obviously, the solution is: a sufficiently smart compiler [22]. It is possible to imagine a special-purpose optimizer that analyzed a Nock formula and determined that it actually computed decrement. We could also imagine an optimizer for addition, etc. Each of these optimizers would be nontrivial.

But a sufficiently smart optimizer doesn't need to optimize every Nock formula that could calculate a decrement function. It only needs to optimize one: the one we actually run. This will be the decrement in the kernel library. Any other decrement will be $\mathcal{O}(n)$; and anyone who rolls their own decrement deserves it.

Again: the optimizer need not *analyze* formulas to see if they're decrements. It only needs to *recognize* the set of standard decrements which it actually executes. In the common case, this set has one member.

To declare itself as a candidate for optimization, the kernel `dec` function applies a hint that tells the interpreter its official name. The interpreter can check this assertion by checking a hash of the formula against its table of known formulas.

The C module that implements the efficient decrement is called a "jet." The jet system should not be confused with an FFI: a jet has no reason to make system calls, must never be used to produce side effects, and is always bound to a pure equivalent.

Jets separate mechanism and policy in Nock execution. Except for perceived performance, neither programmer nor user has any control over whether any formula is jet-propelled. A jet can be seen as a sort of "software device driver," although invisible integration of exotic hardware (like FPGAs) is another use case.

Unlike intrinsic instructions, in a classic interpreter design, jets do not have to be correlated with built-in or low-level functionality. They also can be written in reverse from existing low-level code with tightly defined semantics; for instance, Urbit crypto is jetted by C reference implementations from well-known libraries.

At present all jets are shipped with the interpreter and there is no way to install them from userspace, but this could change. Jets are of course the main exploit vector for both computational correctness and security intrusion. Fortunately, jets don't make system calls, so sandboxing policy issues are trivial. But the sandbox transition needs to be very low-latency.

A Nock interpreter written in Rust [7], with Rust jets, is in early development by a third party. A safe systems language like Rust is is probably the right long-term solution.

5.2.5 L, the lifecycle function

The Urbit lifecycle function, as a Nock formula, is `[2 [0 3] [0 2]]`. If E is the step history of an Urbit instance, the instance's current state is `*[`E `[2 [0 3] [0 2]]]`.

This simply uses the first (larval) step as a formula, whose subject is the rest of the step history.

5.2.6 T, transition function

T, the step transition function, is easy to understand now that we know Nock.

In instruction `9`, the `c` part produces a noun which is a sort of "object" (a Hoon `core`, to get ahead of ourselves). The core (which is in fact the Arvo kernel) is canonically a `[code data]` cell. The `b` part is the address of a Nock formula within the the core (not unlike a vtable index in C++). The intended subject of this formula is the entire core.

Our state S is one of these cores. T is a "method" on the core. The method's product is a new state core S and a list of output actions.

It's also easy to see, at least in principle, how we achieve the goal of ensuring that all semantics are updatable. The new core produced by T has to match the Nock interface of the old state (in the C++ sense, the vtable indexes can't change). Nock can't change. Anything else can change – Hoon could be replaced by a different language, so long as it can generate a Nock core with the right shape.

6 Hoon: a strict functional language

Hoon is a strict, typed, higher-order pure functional language which compiles itself to Nock. While Hoon is not much less expressive than Haskell, it avoids category theory and aspires to a relatively mechanical, concrete style. A very rough comparison: Hoon is to Haskell as C is to Pascal. Like C, Hoon is a relatively thin layer over the compiler target (Nock).

6.1 Semantics

The semantic core of Hoon is the back-end compiler function, `mint`, which is 1500 lines of code for type inference and code generation.

`mint` works with three kinds of noun: `nock`, a Nock formula; `twig`, a Hoon AST; and `span`, a Hoon type, which defines a set of valid nouns and their semantics. The type signature of `mint` is `$-({span twig} {span nock})`; that is, `mint` takes the subject type and the expression source, and produces the product type and the Nock formula.

Understanding the `span` system gets us most of the way to understanding Hoon. We'll do this in two steps. First we'll explain a simplified version of `span` (Listing 2) that doesn't support adequate polymorphism; next we'll extend it to the real thing.

6.1.1 Simplified spans

Listing 2: 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}
      ==
```

`$noun` is the set of all nouns; `$void` is the set of no nouns. `{$cell p/span q/span}` is the set of all cells with head `p` and tail `q`; `{$fork p/{set span}}` is the union of all spans in the set `p`.

A `$hold` span, with span `p` and twig `q`, is just a lazy reference to the span of `(mint p q)`. Since Hoon is a strict language, infinite data structures are always expanded manually.

A `{$face p/term q/span}` wraps the label `p` around the span `q`. Remember that Hoon has no concept of scope or environment; therefore, it has nothing like a symbol table. For a discussion of name resolution, see below.

An `$atom` is an atom, with two twists. `q` is a `unit`, Hoon's equivalent of `Maybe` in Haskell or a nullable in Rust. If `q` is `~`, null, 0, any atom is in the span. If `q` is `[~ x]`, where `x` is any atom, the span is a constant; its only legal value is `x`.

`p` in the atom is a terminal used as an *aura*, or soft atom type. Auras are a lightweight, advisory representation of the units, semantics, and/or syntax of an atom. Two auras are compatible if one is a prefix of the other.

For instance, `@t` means UTF-8 text (LSB low), `@ta` means ASCII text, and `@tas` means an ASCII symbol. `@u` means an unsigned integer, `@ud` an unsigned decimal, `@ux` an unsigned hexadecimal. You can use a `@ud` atom as a `@u` or vice versa, but not as a `@tas`.

But auras are truly soft; you can turn any aura into any other, statically, by casting through the empty aura `@`. Again, auras are advisory; these definitions are just conventions. Hoon is not dependently typed and can't enforce data constraints.

A `$core` is a code-data cell. The data (or *payload*) is the tail; the code (or *battery*) is the head. `p`, a span, is the span of the payload. `q`, a name-twig table, is the source code for the battery.

Each twig in the battery source is compiled to a formula, with the core itself as the subject. The battery is a tree of these formulas, or *arms*. An arm is a computed attribute against its core.

All code-data structures in normal languages (functions, objects, modules, etc) become cores in Hoon. A Hoon battery looks a bit like a method table, but not every arm is a "method" in the OO sense. An arm is a computed attribute. A method is an arm whose product is a function.

A Hoon function (or *gate*) is a core with one arm, whose name is the empty symbol `$`, and a payload whose shape is `[sample context]`. The *context* is the subject in which the gate was defined; the *sample* is the argument. To call the gate on an argument `x`, replace the sample (at tree address `6` in the core) with `x`, then compute the arm. (Of course, we are not mutating the noun, but creating a mutant copy.)

6.1.2 Limb and wing resolution

Understanding spans, without any specific knowledge of twigs, for the most part constrains Hoon enough to understand it in principle. One exception is the `limb` and `wing` twigs that reference the subject.

An individual reference into the subject is a `limb`. A limb can be a name (like `foo`) or a tree address (like `+13`, which is Nock `/13`).

Again, there are no symbol tables in Hoon. Name resolution is a tree search in the subject: depth-first, head-first in cells. There is no separate resolution model for arms and legs; the first match we find, arm or leg, we return.

The search is stopped by a `$face` unless the limb matches the label. If we find a face `foo` and we're looking for a `bar`, we fail, instead of looking inside the `foo` for a `bar`. But in a core where the search limb matches no arm, we descend into the payload.

A basic name reference returns the first match, but a limb can carry a skip counter (expressed by a unary `^` prefix; eg, `^foo` to skip one `foo`, `^^^foo` to skip three) to find deeper matches that have been masked.

A `wing` is a search path, a list of limbs, syntactically separated by `.`. Wings associate right, not left as in classical languages; `foo.bar.baz` in Java is `baz.bar.foo`, hence "baz in bar in foo", in Hoon.

6.1.3 Forward tracing

Type inference in Hoon uses only forward tracing, not unification (tracing backward) as in Hindley-Milner (Haskell, ML). Hoon needs more user annotation than a unification language, but it's easier for the programmer to follow what the algorithm is doing - just because Hoon's algorithm isn't as smart. But the Hoon type system can solve most of the same problems as Haskell's.

Understanding `$hold` explains most of the type system. Computing an arm on a core always produces a `$hold`; the compiler never calculates a "type signature" in any sense. Whenever type analysis encounters a `$hold`, we expand it. This can be and is cached, of course.

Effective analysis of interesting data structures is possible, because non-perverse code *repeats* its span as we trace through it. Consider the simplest recursive span: a list.

Lists in Hoon are not a built-in language feature, but userspace code; they use genericity, which we haven't covered yet. But suppose the span of some list is expressed as a `[%hold p q]`. Call this noun `x`. What span should we expect when we expand `x`?

A list is either null (the span `[%atom %n 0 0]`) or a cell (`[%cell i x]`, where `i` is the list item span, and `x` is our original hold). "Either" means a `$fork` with a set containing these two spans. So as we trace into the list, we see `x` recur.

This is how a finite span can describe an indefinitely long list. Now, suppose we're comparing two list spans `x` and `y`, to see if they're compatible types? Can we compare these infinite expansions in finite time? Yes – as we trace into them, we just have to keep a working set of the comparisons we're trying to prove, and prune the search tree instead of looping back into them.

What if we're tracing through a twig that is "irrational" and doesn't repeat its span? Then the compiler goes into an infinite loop. This is actually a hard mistake to make, and the developer UX is similar to that of a runtime infinite loop: you press `^C` and get a stack trace.

This design has no trouble with tail recursion. Head recursion requires an explicit cast (which is still verified). Broadly, it's good practice to cast the product of any arm.

6.1.4 Pattern matching and branch specialization

The other piece needed to understand Hoon type inference is *branch specialization*. Hoon has a pattern-match twig `{$fits p q}`, emitting a formula that tests whether the value of `q` is within the span of `p`.

A branch with a test which is a `$fits`, or *arbitrary boolean algebra* using `$fits`, where `q` is a fragment of the subject, compiles each side of the branch with a subject specialized to what the branch test has learned. We use this to specialize unions, then manipulate each special case directly.

If the compiler detects that one branch is never taken, it's an error; this is invaluable when, for example, refactoring a tagged-union traverse.

6.2 Type definition

In most languages, types and data structures are the same thing. In Hoon they are different things. A type is a `span`; a data structure is a `mold`.

There is no Hoon syntax for a span. We don't declare spans. We infer them. Why not? There is no span that can't be defined as the `mint` of some twig and subject. So type inference solves type declaration.

We like to constrain the nouns we produce to match regular and rigorous data structures. We define these data structures with type inference. But we define them as the ranges of construction functions, and Hoon has macros specialized for expressing these constructors. A data constructor, or `mold`, is easily mistaken for a type declaration in a more conventional language. For instance, the definition of `span` above is a mold.

A mold is omnivorous; the argument to a mold is always `$noun`. Molds are also idempotent. If the argument was already in the mold's range, the product is the argument; `=((mold x) (mold (mold x)))`. In other words, a mold *normalizes* its argument.

Since Hoon is a typed language and it's bad form to discard type, you rarely actually *call* a mold function. Molds are normally only called to validate untrusted raw nouns from the network. Anywhere else, "remolding" data shows sloppiness.

We do often *bunt* a mold: create a default or example noun. Logically, we bunt by passing the mold its default argument. But normally, this call can be folded at compile time and generates a constant.

6.3 Syntax

Hoon's syntax approach is the opposite of Lisp's. Hoon is full of syntax. The front end (parser and macro expansion) is almost as complex as the back end.

To understand Hoon syntax, we need a brief overview of the AST or `twig` structure. To simplify slightly, a twig has a head which is a tag (`stem`), and a tail (`bulb`) which is a structure containing more twigs, most often a tuple of twigs.

The set of these twig stems is fixed. They are all "special forms" in the Lisp sense. Most (all but 28) are implemented internally as macros, but Hoon has no user-level macros.

Each stem has its own bulb syntax. But most follow a pattern called *regular form*, which we'll explain first.

6.3.1 Regular form: sigils and runes

A twig in regular form starts with one of two syntactic choices: a keyword, which is just the stem with `:` prefixed; or a *rune*, an ASCII digraph.

For example, the twig form `{$if p/twig q/twig r/twig}` is written `:if(p q r)` in keyword form, and `?:(p q r)` in rune form. Experienced programmers find runes much more readable, partly because the first glyph of the rune defines a functional category (like `?` for conditionals).

Because Hoon uses ASCII symbols so heavily (there is some resemblance to Perl or even the APL family, although Hoon is more regular than both), it presents a serious vocalization problem. To aid in *speaking* Hoon – both for actual communication, and to assist in subvocalized reading – we have our own one-syllable names for ASCII tokens (Listing 3).

Listing 3: Phonetic names for symbols

```
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 !
```

6.3.2 Regular form: tall and flat modes

There are two common syntactic problems in functional languages: closing terminator piles (eg, right parens in Lisp) and indentation creep. A complex function will have a deep AST; if every child node in that AST is indented past its parent, any interesting code tends to creep toward the right margin.

To solve terminator piles, there are two forms of every Hoon twig: "tall" and "flat", ie, multiline and single-line. Tall twigs can contain flat twigs, but not vice versa, mimicking the look of "statements" and "expressions" in an imperative language. Flat form is enclosed by parentheses and separated by a single space; tall form is separated by multiple spaces or a newline, and (in most cases) not enclosed at all. (Note that except to distinguish one space from many, Hoon does not use significant whitespace.)

The trick is that most twigs have fixed fanout. For example, we know that `{$if p/twig q/twig r/twig}`, if-then-else, has three children. Since the parser has a custom rule for each twig form, rather than shared syntax as in Lisp, that rule can parse three children and then stop. Some twigs are n-ary; tall form has no choice but to indent each child, and use a terminator `==`.

Right-margin creep is prevented by *backstep indentation*; where a classical language might write

```
?:  test
  then
  else
```

Hoon writes

```
?:  test
  then
else
```

Ideally the most complex twig is the `else`; if not, there's an alternate form, `?.`, which reverses `q` and `r`.

Backstepping is just the standard pattern, of course, and the programmer is free to lay out their code in the most readable fashion. The only actual rule is that single spaces are not sufficient to separate tall-form twigs.

6.3.3 Irregular forms

Fortunately or unfortunately, any twig can also have an irregular form. For example, `(add 2 2‘` is the irregular form of `%+(add 2 2)` or `:calt(add 2 2)`. Irregular forms are always wide, and there's nothing to do but learn them.

6.4 Example

FizzBuzz with rune sigils:

```
=+  count  1
|-  ^-  (list tape)
?:  =(100 count)
    ~
:_  $(count (add 1 count))
?:  =(0 (mod count 15))
  "FizzBuzz"
?:  =(0 (mod count 5))
  "Fizz"
?:  =(0 (mod count 3))
  "Buzz"
"{<count>}"
```

Further examples using both keyword and rune syntax are in section 14.

6.5 Advanced polymorphism

As we noted, the definition of `span` in Listing 2 is simplified. The full version is in Listing 4.

If cores never changed, we wouldn't need polymorphism. Of course, nouns are immutable and never change, but we can use one as a template to construct a new noun.

Suppose we take a core, a cell whose head is a battery (tree of arm formulas) and whose tail is a payload (any kind of data), and *replace its tail with a different noun.* Then, we invoke an arm from the battery.

Is this legal? Does it make sense? Actually, every function call in Hoon does this, so we'd better make it work well.

Listing 4: Full ++span

```
++  span  $@  $?  $noun
                  $void
            ==  $%  {$atom p/term q/(unit atom)}
                    {$cell p/span q/span}
                    {$core p/span q/coil}
                    {$face p/term q/span}
                    {$fork p/(set span)}
                    {$hold p/span q/twig}
                ==
::
++  coil  $:  p/?($gold $iron $lead $zinc)
              q/span
              r/{p/?($~ ^) q/(map term foot)}
            ==
++  foot  $%  {$dry p/twig}
              {$wet p/twig}
            ==
```

The full core stores both payload spans. The span that describes the payload currently in the core is `p`. The span that describes the payload the core was compiled with is `q.q`.

In the Bertrand Meyer tradition of type theory [9], there are two forms of polymorphism: variance and genericity. In Hoon this choice is per arm, which is why our battery went from `(map term twig)` to `(map term foot)` when it went into the coil. A `foot` is `$dry` or `$wet`. A dry arm uses variance; a wet arm uses genericity.

6.5.1 Dry arms (variance)

For a dry arm, we apply the Liskov substitution principle [6]: we ask, "can we use any `p` as if it was a `q.q`"? This is the same test we apply in `:cast` or any type comparison (`nest`).

Often cores themselves need to be checked for nesting. The semantics of this test are the well-established rules of variance: arguments are contravariant and write-only, results are covariant and read-only. To be exact, cores can be invariant (`$gold`), bivariant (`$lead`), covariant (`$zinc`), or contravariant (`$gold`).

6.5.2 Wet arms (genericity)

For a wet arm, we ask: suppose this core was actually compiled using `p` instead of `q.q`? Would the Nock formulas we generated for `q.q` actually work for a `p` payload?

If so, we can effectively customize the type signature of the core for the payload we're using. Consider a function like `turn` (Haskell: `(flip map)`),

which transforms each element of a list. To use `turn`, we install a list and a transformation function in a generic core. The span of the list we produce depends on the span of the list and the span of the transformation function. But the Nock formulas for transforming each element of the list will work on any function and any list, so long as the function's argument is the list item.

When we call a wet arm, we're essentially using the twig as a macro. We are not generating new code for every call site; we are creating a new type analysis path, which works as if we expanded the callee with the caller's context.

Again, will this work? A simple (and not quite right) way to ask this question is to compile all the twigs in the battery for both a payload of `p` and a payload of `q.q`, and see if they generate exactly the same Nock. The actual algorithm is a little more interesting, but not much.

(A Haskeller might say that in a sense, `q.q` and `q.r.q` (the original payload and the battery) define a sort of implicit typeclass. And indeed, Hoon uses wet arms for the same kinds of problems as Haskell typeclasses.)

6.5.3 Constant folding

There's only one field of the `coil` we haven't explained yet: `p.r.q`. This is simply the compiled battery, if available. (Of course, we compile the twigs in a core against the core itself, and the formulas can't be available while we're compiling them.) External users of the core want this battery constant, though: it lets us fold constants by executing arms at compile time.

7 Arvo: an event-driven kernel

Arvo is a single-threaded event interpreter. In many ways it resembles other such event dispatchers, such as `node.js`. In others it's more unusual. We don't have space to explore Arvo deeply, but let's touch on a few of the design choices.

7.1 Kernel interface

Arvo is the iteratively updated core we created at the end of the larval stage. The core (Listing 5) has two arms, named (in honor of BASIC) `peek` and `poke`.

The `poke` gate applies the next step. It produces a list of actions and a new Arvo core.

The `peek` gate defines a global referentially transparent namespace. If its product `val` is `~` (null), the path `pax` is mysterious; its value cannot be determined synchronously. If it produces `[~ ~]`, `pax` is known to be unbound (and can never become bound). Otherwise, the namespace produces a mark (external type label, like a MIME type) and a general noun.

Steps and actions share the `ovum` mold, which is a `card` (an untyped tagged union case) and a `wire` (a list of symbols representing a cause). Effects are routed to the same wire as their cause.

For example, an HTTP request is sent to Arvo with a wire that uniquely identifies the request socket (keeping in mind that Arvo doesn't care if Unix

Listing 5: The Arvo interface

```
++  card  {@tas *}
++  path  (list @tas)
++  wire  path
++  ovum  {p/wire q/card}
++  work  ovum
++  step  {p/@da q/ovum}
++  mark  @tas
++  arvo
  |%
  ++  poke
    $-  nex/step
    {act/(list work) _arvo}
  ::
  ++  peek
    $-  pax/path
    val/(unit (unit {p/mark q/*}))
  --
```

reboots). Arvo must produce a response ovum on the same wire. It does not parse the wire; it just uses it as an opaque cause identifier.

7.2 Kernel modules

Arvo is just an event distributor; most Arvo functionality is in a system of kernel modules or *vanes*. The kernel proper is about 600 lines of Hoon.

Arvo loads vanes as source and holds each as a `{span noun}` cell, or `vase`. Dynamic type in Urbit means using the statically typed compiler on runtime vases. Using vases means we can reload vanes from source, obviously a necessity.

A vane is a core on the same general pattern as Arvo itself, with arms that handle events, bind names, etc. To update a vane, we compile the new source and ask the resulting core to copy the relevant state from the old core. This pattern is also reused for updating user-level applications within the `%gall` vane.

Vanes have "routing addresses" which are single letters; they are implemented by source files starting with that latter. Vane `%a`, `%ames` handles networking; `%b`, `%behn`, timers and initialization; `%c`, `%clay`, git style revision control; `%d`, `%dill`, consoles; `%e`, `%eyre`, HTTP; `%f`, `%ford`, resource assembly; `%g`, `%gall`, application control.

7.3 Structured events

Each step that enters Arvo becomes a vane-level event, or `move` (Listing 6). Moves can cause other moves; one external step becomes a cascade of moves, processed depth-first as a transaction.

Listing 6: `++move`

```
++  duct  (list wire)
++  gang  (unit (set ship))
++  mask  {p/gang q/gang}
++  leap
  $%  {$give p/duct q/card}
      {$pass p/duct q/wire r/card}
  --
++  move  {p/mask q/leap}
```

A single-threaded, nonpreemptive event dispatcher, like node or Arvo, is analogous to a multithreaded preemptive scheduler. There's a duality (in the mathematical sense) between event flow and control flow. One disadvantage of many event systems is unstructured event flow, often amounting to "event spaghetti." The control-flow dual of an unstructured event system is `goto`.

The Arvo move system is the dual of a call stack. Wires work as in the external `step` event, but a `duct` is a stack of wires. A `$pass` move is a call; a `$give` move is a return.

To use an event-oriented service, the caller encodes the cause, reason, context or purpose of the request in the `$pass` wire. When the service completes, it returns the result on the calling duct. The caller is passed the return value and the saved wire, which provides context to the return handler.

Since wires are symbolic, causes have to be encoded as symbols, which is not ideal for efficiency but is ideal for debugging. A saved duct becomes the dual of a continuation, but is much more printable than a continuation. It's also easily used as a table key, not a common use of continuations.

It's straightforward to build a promise system on top of moves, but it's not conventional Arvo style. The ideal Arvo application core (whether vane or user-level app) core contains a pure data structure without any functional state. For example, if the state machine contains promises, closures, which represent in some sense blocked sequential processes, upgrading it may be an unnecessarily interesting problem.

7.4 Security mask

As described above, Arvo is a "single-homed" OS; an instance has one identity (a `ship`; see the network architecture below) assigned at birth. All vane code must be fully trusted by this identity, a subject leg called `our`.

But since sandboxed user-level apps are a necessity, and apps can generate moves, we need a security model assertion for moves. Thus the `mask` noun in every move.

A security mask is a cell of two `gang` nouns, each defining a set of ships: `p` or *who*, and `q` or *whom*. The mask states *who* (other than us) caused this move (tainted by), and *whom* (other than us) it is allowed to affect (leaked to).

Each mask is a unit; if *who* is `~`, it means that anyone could have tainted this move (it's completely untrusted, like an incoming packet); if *whom* is `~`, it means that data from it can leak to anyone (it's public information). Whereas the empty set, `[~ ~]`, means "totally trusted" and "totally private" respectively.

Normally, a move produces effects with the same mask as its cause. Only well-defined authentication or sandboxing semantics should violate this default.

8 Ames packet protocol, part 1: principles

We don't have space to look at every Arvo vane, but let's take a core sample of the most important one: the networking vane, `%ames`. After all, Urbit is a packet transceiver. If we understand how raw bits on the wire are authenticated, decrypted, validated, and presented to the application as typed data structures, we probably understand Urbit.

We'll follow a specific case of packet networking: a `poke`, or one-way transactional RPC, between two user-level apps. This isn't the only use case of Ames, but it solves all the important problems of secure peer-to-peer communication.

Before we explain the bits-on-the-wire protocol, we need to describe the concepts and assumptions of Urbit networking, which are quite divergent from the classical TCP/IP stack.

8.1 Routable identity

An Urbit identity is called a `ship`. A ship is both a digital identity and a routing address – imagine an Internet in which DNS and IP were a single routable identity. Every Urbit interpreter instance (unique event history)

An Urbit interpreter instance (ie, a unique event history) is called a `ship`. The terms `ship` and `ship` are often mixed up, because the ship-to-ship mapping is 1:1; a ship has one unique ship, set permanently in its larval sequence. "An urbit" is also used as a synonym for both.

A ship is a 128-bit number. A 128-bit number *per se* is not a very useful digital identity. An identity does not need to be human-meaningful, but it does need to be human-memorable.

Ships are printed with the `@p` aura, a base-256 phonetic encoding with short formats for size classes. `@p` divides 128-bit ships into five length classes: an 8-bit, 1-syllable "galaxy"; a 16-bit, 2-syllable "star"; a 32-bit, 4-syllable "planet"; a 64-bit, 8-syllable "moon"; a 128-bit, 16-syllable "comet".

The intended role of a ship is defined by its length class. Galaxies and stars are network infrastructure; planets are personal servers; moons are clients/appliances; comets are bots.

Planets (for example, `~dabnec-forfem`) are short enough to fit in the brain's "human name in a foreign language" slot. They are not meaningful, but assigned meaningless names have their own quality: unlike either real names or self-selected handles, they're opaque and impersonal. Not all social interactions on

the network are naturally impersonal, but many are. And it's not hard to layer local nicknames ("petnames" [15]) over global impersonal names.

There are enough planets for a healthy global network (the same number as IPv4), but few enough that they remain scarce. New identities on a social network should be nontrivial to obtain, to discourage spam and other Sybil attacks.

Three questions remain: how are ships authenticated? How are packets between them actually routed? And how are they allocated?

8.2 Cryptography

A public key in Urbit is called a `pass`; a private key is called a `ring`. Both pass and ring are atoms. They start with a cryptosuite byte for algorithm update, so we can think of all algorithms as a single meta-cryptosystem.

An Urbit cryptosystem is stateless and deterministic. It supports hashing; two public-key models, signed-only and signed-plus-encrypted; and deterministic symmetric authenticated encryption. To avoid stateful nonces and entropy, it restricts its users to contexts where duplicate plaintexts (which map to duplicate ciphertexts) leak no information. Again, packets must be facts and not commands.

The public-key encryption step encrypts a symmetric key, which the caller (who presumably has entropy) generates, and encrypts the payload with this key. Decryption produces both plaintext and the symmetric key, now a shared secret.

The current cryptosuite A (`crua`) is strange hand-rolled garbage. It might be secure, but who knows? Its replacement B (`crub`) is AES in SIV mode [11], SHA-256, Curve25519 and Ed25519.

8.3 PKI

Any ship is bound to at most one keypair at any time. So the simplest possible registry is logically `(map ship (unit pass))`. But this is a snapshot; in Urbit we prefer histories. What we really want is a self-validating list of the public keys that this ship has used for its entire lifetime.

This list is an Urbit certificate or `will`; an item in the list is a `deed`. The deed's index in the list is its `life`. A deed is an inception date, an optional expiration date, and a pass (public key), signed by one or more *parent* ships.

Who has to sign what deed? In *normal succession*, each deed is signed by its predecessor, ie, previous life of the same ship. Effectively, key revocation is key renewal; the key signs over its power to its successor. But the rules vary by ship class and, of course, for the initial or *launch* deed.

Every ship except a galaxy has a *parent* which defaults to its low half bits (or low 8 bits for a 128-bit comet). The parent of most moons is a planet; the parent of most planets is a star; the parent of all stars and all comets is a galaxy.

Any ship except a moon can escape (change parents), but only to another galaxy or star, respectively, and with its active consent. An escape deed is a

normal succession, but adds the signature of the new parent.

A comet (128-bit ship) signs its own launch deed. The ship must equal the hash of the launch pass. Subsequent deeds use normal succession. Cometary space is decentralized; anyone can create a comet.

A moon (64-bit ship) is launched by its parent. Subsequent deeds are signed by the parent. Lunar space is totally captive; a moon is the property of its parent, like a user in a multiuser system.

A planet or star (32-bit or 16-bit ship) is launched by its parent. Subsequent deeds use normal succession. The planet is the lowest level of allodial title; it owns itself, like Bitcoin.

A galaxy is launched by self-signing. Subsequent deeds use normal succession. The hash of the launch pass must match the value in the *galaxy table*, which is hardcoded as a magic constant in the Arvo source. Galaxies are "premined" in the Bitcoin sense.

The will update rule is simple: a will can be updated if and only if it extends a will already in the registry, or if it updates a parent signature to a new life. All parent signatures in the update must match the current local parent life.

Intuitively, a ship that accepts a new deed invalidates the private keys of all previous deeds to that ship. To make sure your latest deed is inviolable, and all previous deeds are unusable, broadcast the latest globally. The details of deed update are in the section after next.

8.4 Update, routing, and parental trust

Normally, the child automatically syncs and loads source updates from the parent. Subscribing to an update server requires almost complete trust; this problem is not unique to Urbit, however. Also, since we have paid this price in extending trust, we might as well use this relationship wherever it's useful.

Routing is an obvious example. Urbit is a P2P network, so fully connected routing on the real internets requires a NAT traversal server. This server is the parent. Both STUN and TURN fall out of the forwarding protocol naturally. To bootstrap, galaxies are DNSed at `$galaxy.urbit.org`,

For planets and above, the threat model between parent and child is that the parent can effectively deny service to the child, but the child can escape from the parent to another parent of the same class (eg, a planet's parent is some star).

So there is no strong economic motive for the parent to abuse or starve the child – quite the contrary. A planet which *no* star is willing to host must be quite an nasty planet. A star which turns nasty will lose the planets it's already issued, and lose capital in the planets it hasn't. Everyone has an economic incentive to behave responsibly.

8.5 Deed distribution

In general principle, Byzantine global broadcast of deeds is the same consensus problem that Bitcoin solves. If eager broadcast were perfect and trusted, Bitcoin

would not need proof-of-work.

But Urbit's threat model is very different from Bitcoin's. Bitcoin is a trust-free system; Urbit has a central trust hierarchy. Spends in Bitcoin are high-frequency, fungible, and must be low-friction, much like monetary transactions in real life; ship transfers or rekeys in Urbit are low-frequency, non-fungible, and can be high-friction, like real-estate transactions in real life.

One advantage of the real-estate model is that Urbit ships, like real estate, have far fewer criminal use cases. Existing systems of digital real estate, such as DNS and IP addresses, tend to be unregulated, simply because they haven't caused any problems.

In theory, Urbit distributes deeds along three paths. Only the first is active right now, because Urbit is a testnet.

8.5.1 Lazy peer update

First: fresh deeds are piggybacked on peer-to-peer packets. A message sent to a past identity, even a cleartext message when the sender has no will for the receiver, remains authentic.

Lazy peer update is a backstop to make sure communication is always possible and resolves to a fully encrypted channel. Alone, it remains vulnerable if old keys are stolen.

8.5.2 Hierarchical synchronization

Second: fresh deeds are distributed hierarchically with the `%clay` revision control system. Parents and children sync a key desk (branch) to each other. Galaxies sync mutually.

Hierarchical update depends on the integrity of the parent hierarchy. If this trust is broken, censorship attacks become possible and independence is compromised. But this is a minor consequence of a breach in parent integrity.

8.5.3 Eager gossip with conflict detection

Third: if the parent hierarchy (which, because of the escape mechanism, is designed to be very resilient), somehow becomes compromised, an eager gossip protocol like Bitcoin's is needed.

However, this does not imply a proof-of-work requirement. As [5] suggests, well-connected gossip networks are hard to censor in practice, and double-spend alerts are effective.

A "double sell" attack on Urbit would involve simultaneously signing two different successor deeds, and distributing them both on the gossip network. The double sell works if and only if each alternate reaches the intended "buyer," and triggers an irrevocable settlement, before a conflict notification or the other alternate. This essentially requires the network to be partitioned, also very unlikely in a gossip network. In practice, unless buyer and seller have mutual trust, use a trusted escrow agent.

If all else fails, a new or existing blockchain is an option. But as presently analyzed, a blockchain for digital real estate on the Urbit design looks like overkill.

8.6 Permanent networking

Protocol design is much easier with the assumption of uniform persistence. In a protocol designed for uniformly persistent nodes, we can assume that no node involuntarily loses state. We call this assumption *permanent networking*.

In the classical stack, a basic result of protocol design is that you can't have exactly-once message delivery [21]. To put it in the terms we used earlier: you can't build a bus on a network. With permanent networking between solid-state interpreters, this feature is straightforward. Why? Because two uniformly persistent nodes can maintain a *permanent session*.

When classical nodes reboot, their TCP sockets (session layers) break, because session state is in RAM. If an HTTP POST has been sent on a TCP socket but the HTTP response has not yet been received, and then the socket closes, there is no way for the client to know if the server applied the POST. When a new socket is opened, the client can resend (at-least-once delivery) or fail to resend (at-most-once). The programmer has to understand that the socket is not *really* a bus, and make sure the POST is actually an idempotent fact rather than an imperative command. (The idempotence problem is often punted to the human layer: "please click only once to make your purchase.")

With uniform persistence, any pair of nodes can maintain a persistent shared sequence number. A message in this permanent session is both a command and a fact. Bob declares that message number `x` in his permanent conversation with Alice is the noun `y`. If Alice hears this declaration twice, she learns nothing more than if she heard it once. Once she acknowledges message `x`, and Bob hears this acknowledgment, Bob will never share this fact with her again.

One cost of permanent networking is that brain damage is fatal by default. There is no trivial way to recover or recreate a ship whose ship has sunk. By losing persistent session numbers, it has lost its ability to even communicate. Even a "backup" makes no sense; you've lost data or you haven't.

Another cost of permanent networking is that we add persistence latency to the network roundtrip time. This is not a problem in theory, but in practice it is. If we define "persisted" as "stored in RAM on three machines in separate availability zones," persistence latency can be reduced to a few milliseconds for a cloud server: manageable but nontrivial.

Urbit is designed to run in a data center; it assumes that a high-reliability, low-latency cloud log is a service we know how to deliver. On mobile hardware, low-latency NVRAM would be ideal, but mobile hardware and critical data remain a bad fit.

8.7 Permanent applications

There is not just one persistent conversation between Alice and Bob. Since Arvo is an OS, it's not one monolithic codebase. Multiple components need multiple conversations.

Arvo has two layers of modularity, both permanent: kernel modules (*vanes*) and user-level applications (within the `%gall` vane). An Arvo "socket" identity in the formal sense contains the names of the communicating subsystems.

Applications must follow the same permanence rules as ships. Just as there is no correct way to destroy, then recreate a ship, there is no correct way to remove, then reinstall an app. You can upgrade the app to a new codebase, whether from the original developer or a competitor; but the upgrade adapts the state of the old application, continues all its conversations, and remains responsible for its acknowledgments.

8.8 End-to-end acknowledgment

One problem that often produces unpredictable states in classical network stacks is multiple layers of error and acknowledgment. For instance, consider a simple REST API returning JSON. An error can be a TCP or TLS socket error, an HTTP result code, or an in-band JSON application message.

Urbit has two acknowledgment layers: messages and fragments. A message of arbitrary size is broken into MTU-size fragments. Each fragment states the sender, message and fragment number, but only whole messages are authenticated/decrypted.

E2E acknowledgments mean that (a) acknowledging all fragments in the message implies acknowledging the message, (b) acknowledging the message implies that the message was correctly processed. Every packet is a transaction (at the protocol level), but every message is also a transaction (at the application level). A single ack packet declares that the fragment was received, the message was decoded, and the recipient accepted it.

As a transactional system, Urbit supports both positive and negative acknowledgments. A positive acknowledgment carries no data. A negative acknowledgment carries an error trace. Since duplicate packets must generate duplicate acknowledgments, negative acks need to be saved forever; positive acks can be saved as a counter, with negative exceptions.

A negative ack always results from either (a) an application crash in the receiver or (b) a system error report. For example, a user interrupt or an event timer interrupt (any event not generated by the console, such as a packet, needs a timer, since there is no one to press `^C`) will send a negative ack with a stack trace.

One cost of this design: E2E acks complicate congestion control. The message computation time is in the roundtrip time for message-completing packets. To help the sender's heuristics, an ack includes the measured computation latency. We know of no algorithm which can use this data, but it can't hurt.

8.9 Packet rejection

Classical systems drop packets only because of buffer overflows. A buffer overflow is a great reason to drop a packet, but there are several others.

If a message fails authentication, decryption or validation, the packet that completes it must be silently dropped. There are two possible causes of a weird packet: (a) the sender is deranged or malicious; (b) the sender is emitting some new protocol that the receiver doesn't yet support.

In the case of (a), the drop is a no-brainer. The typical crypto attack involves a leakage channel in error responses, including but not limited to timing attacks. Nonresponse to crypto errors does not eliminate the need for constant-time algorithms – an attacker could find some other channel, such as another packet or even another ship, that could extract the timing signal – but there's certainly no reason to be helpful.

In the case of (b), the drop is also correct. It will cause the sender to back off exponentially until the receiver updates itself. A typical cause is a receiver that's been turned off while network updates were shipped. A period of unavailability is unavoidable, but no errors should propagate up to the user (as they would if we sent a negative acknowledgment). Dropping the packet says the wire is cut, as it is.

9 Ames protocol, part 2: bitstream

9.1 `jam` and `cue`: noun serialization

Most systems think of binary blobs as bitstreams. Urbit thinks of them as atoms (large unsigned integers). The difference: an atomic blob has no non-redundant length field. As a bitstream, its last bit (its high bit as an atom) is always 1. A bitstream can have trailing zeroes; a MIME octet-stream or Unix file invertibly represented as a noun is a `[length data]` cell.

Urbit in general and `%ames` in particular make much use of the `jam` and `cue` serialization system. `jam` maps any noun to an atom and `cue` maps the atom back to a noun, crashing if the encoding is invalid. `jam` and `cue` preserve DAG structure.

The definition of `cue`, the deserializer is given in Listing 8, with its dependency `rub`, the frame decoder in Listing 7.

This is the first layer of defense against barbaric bits. (The definition of `jam` is about the same size.) Its output may still be barbaric, but (a) it's at least a barbaric *noun*, and (b) Hoon is adept at civilizing barbaric nouns.

For obvious performance reasons, `cue` and `jam` are matched with C jets. This jet code (about 100 lines) is Urbit's most promising attack surface.

9.2 `rub`: prefix expansion

`rub` extracts a self-measuring atom from an atomic blob.

Listing 7: ++rub

```
++  rub
~/  %rub
|=  {a/@ b/@}
^-  {p/@ q/@}
=+  ^=  c
    =+  c=0
    |-  ^-  @
    ?.  =(0 (cut 0 [(add a c) 1] b))
      c
    $(c +(c))
?:  =(0 c)
  [1 0]
=.  a  (add a +(c))
=+  ^=  e
    %+  add
      (bex (dec c))
    (cut 0 [a (dec c)] b)
:-  (add (add c c) e)
(cut 0 [(add a (dec c)) e] b)]
```

Listing 8: ++cue

```
++  cue
~/  %cue
|=  b/@
^-  *
=+  a=0
=|  m/(map @ *)
=<  q
|-  ^-  {p/@ q/* r/(map @ *)}
?:  =(0 (cut 0 [a 1] b))
  =+  e=(rub +(a) b)
  [+(p.e) q.e (~(put by m) a q.e)]
=+  d  (add a 2)
?:  =(1 (cut 0 [+(a) 1] b))
  =+  e=(rub d b)
  [(add 2 p.e) (~(got by m) q.e) m]
=+  u=$(a d)
=+  v=$(a (add p.u +(a)), m r.u)
=+  w=[q.u q.v]
:+  (add 2 (add p.u p.v))
  w
(~(put by r.v) a w)
```

We suggest (line 1) our jet identity. We accept (line 2) a a cell `{a/@ b/@}`. `a` is a bit position which serves as a cursor into an atomic blob, `b`. We produce (3) a `{p/@ q/@}`; `p` is the number of bits to advance the cursor and `q` is the encoded atom.

We pin (4) `c`, a unary sequence of 1 bits followed by a 0 bit. If 10) `c` is 0, (11) `p` is 1 and `q` is 0. Otherwise, `c` is the number of bits needed to express the number of bits in `q`.

We advance (12) the cursor, `a`, to include `c` and the terminator bit.

We pin (13) `e`, the number of bits in `q`. This is encoded as a `c`-1 length sequence of bits following `a`, which is added to 2^{c-1}. `p` (number of bits consumed) is `c + c + e` (17). `q` (the packaged atom) is the `e`-length bitfield at `a + c + c` (18).

9.3 `cue`: noun expansion

`cue` decodes an arbitrary noun from an atomic blob.

We suggest (1) our jet identity. We accept (2) an atomic blob `b`. We produce (3) a general noun.

We pin (4) `a`, a cursor position running from low to high, at 0; and (5) an empty map `m`, from cursor to noun. We'll save all cursor-to-noun mappings in this cache `m`.

We then enter a loop (7) which produces a triple `{p q r}`, where `p` is the following cursor position, `q` is the noun, and `r` is the cache. Our product (6) is `q`.

If (8) the first bit at `a` is 0, the noun is a direct atom. We pin (9) `e`, the expansion of the second bit at `a`. `p` (10) is the cursor after `e`, `q` is the atom from `e`, `r` is an updated cache.

We pin (11) a data cursor `d`, the third bit at `a`. If (12) the second bit at `a` is 1, this noun is a saved reference. We rub (13) `d`, producing an atom expansion `e`, whose atom is a cursor. `p` (14) is the length of `e`, plus the two meta bits; `q` is the cursor's value in the cache, crashing if not found; `r` is the current cache `m`.

So the noun is a cell. We pin (15) `u`, decoding the noun at `d`, the head. We pin (16) `v`, decoding the cursor after `u`, using the cache produced by `u`. We pin (17) `w`, the cell of `q.u` and `q.v`. `p` (18) is the lengths of `u` and `v`, plus 2. `q` (19) is `w`. `r` (20) is the cache, with `w` inserted.

9.4 Packet structure

`jam` encoding is not actually used for the outermost layer of a packet. `jam` has the same relationship to custom bit stuffing that an FPGA has to custom silicon. The packet encoding of the `%ames` protocol is custom; but the first thing we make from a packet is a `jam` blob. It's just easier.

We can still express the structure we're producing as a noun.

We're decoding the packet as a `cake` (Listing 9), which is a triple of `sock` (a pair of sender and receiver ships), a `skin` (one of four encodings), and an encoded blob. The decoder is `bite` (Listing 10).

Listing 9: ++cake

```
++  cake  {p/sock q/skin r/@}
++  skin  ?(%none %open %fast %full)
++  skit  ?(%0 %1 %2 %3)
++  sock  {p/@p q/@p}
```

Listing 10: ++bite

```
++  bite
  |=  pac/@
  ^-  cake
  =+  [mag=(end 5 1 pac) bod=(rsh 5 1 pac)]
  =+  :*  vez=(end 0 3 mag)
          chk=(cut 0 [3 20] mag)
          wix=(bex +((cut 0 [23 2] mag)))
          vix=(bex +((cut 0 [25 2] mag)))
          tay=(cut 0 [27 2] mag)
      ==
  ?>  =(7 vez)
  ?>  =(chk (end 0 20 (mug bod)))
  :+  [(end 3 wix bod) (cut 3 [wix vix] bod)]
    (kins tay)
  (rsh 3 (add wix vix) bod)
::
++  kins
  |=  tay/@
  (snag tay `(list skin)`[%none %open %fast %full ~]))
```

The format: a 32-bit header word, then the sender ship, then the receiver ship, then the payload. The header: 3 bits for a protocol version (currently 7); a 20-bit checksum; 2 bits each for the length of sender and receiver ship, in bytes; 2 bits for a skin code; and the last 3 bits reserved.

The 2-bit skin code (`skit`) matches 0 through 3 with `%none` (no crypto), `%open` (signed but not encrypted), `%full` (public-key signed and encrypted), and `%fast` (symmetric authenticated encryption).

9.5 Meal decoding

Our next goal is to decode our packet data into a `meal` (Listing 11).

Listing 11: `++meal`

```
++  bone  @ud  ::  connection number
++  hork  (unit (pair term tang))  ::  error report
++  nose  (pair bone tick)  ::  message identity
++  tick  @ud  ::  message number
++  flap  @uv  ::  128-bit packet hash
++  lane  ::  packet address
  $%   {$if p/@ud q/@if}  ::  udp4: port and IPv4
       {$is p/@ud q/@is}  ::  udp6: port and IPv6
  ==
++  mile  ::  compound message
  $:  p/nose  ::  message identity
      q/skit  ::  skin of whole
      r/@ud   ::  number of fragments
  ==
++  frag  ::  fragment detail
  $:  p/@ud  ::  fragment number
      q/@    ::  fragment data
  ==
++  bark  ::  acknowledgment
  $:  p/bone          ::  connection number
      q/flap          ::  packet hash
      r/(unit hork)   ::  success/error
      s/@dr           ::  compute time
  ==
++  meal
  $%  {$back p/bark}              ::  acknowledgment
      {$bond p/nose q/path r/*}   ::  message
      {$part p/mile q/frag}       ::  fragment
      {$fore p/(unit lane) q/@}   ::  forwarded packet
  ==
```

If the encoding is `$none`, we just `cue` the packet data and mold it as a `meal`. `$none` is only valid for `$part` and `$fore` packets. (For a fragmented message, crypto is at the message level.)

If the encoding is `$fast`, the first 128 bits of the message are the hash of the symmetric key. If we hold this key, we use it to decrypt the rest of the packet and `cue` it as a meal. Otherwise we drop the packet. (Apparent connectivity loss makes the sender fall back to `$full` encoding.)

If the encoding is `$open`, the packet is signed but not encrypted (because the sender has no public or symmetric keys for the receiver). We `cue` the data as a cell `{p/will q/@}`, a certificate and a payload. The payload is signed with the key of the newest deed in will `p`. The verification function in the cryptosuite checks this signed payload `q` and spits back out another atom,

If the encoding is `$full`, the packet is signed and public-key encrypted. We `cue` the data as a triple `{p/life q/will r/@}`. This is the payload of `$open`, plus a `life` that tells us which deed in *our own will* the packet is encrypted to. (A sender may have an outdated will, which is still better than nothing).

A `$full` packet, if addressed to our current life, also contains a valid shared secret. Packets encoded with this secret can be sent as `$fast` to the sender.

Every ship tracks the set of other ships it has shared its will with; it sends with `$open` or `$full` when it is not sure the receiver has the latest will. We don't send the full will, just the suffix diff. If a will update fails, obviously, the packet is dropped.

If the sender has no receiver will, the general flow of a conversation is to send the first message with an `$open` encoding, ie, signed but not encrypted. This will trigger a `$full` ack, whose symmetric key the sender saves as a shared secret, establishing the secure session from initial sender to initial receiver. To finish the key exchange, the sender will respond with a dummy ack, also `$full`, also containing a shared secret.

This design eliminates crypto-specific roundtrips at the cost of a leak. The leak is motivated by the observation that your first words to a stranger are seldom a secret.

9.6 Meal processing

A `$bond` meal is a completed message ready to pass to the next level up.

The first field in a `$bond` is a `nose`, a message identity. This is a message sequence number or `tick`, relative to an outbound session identity or `bone`. To an application, a bone is best compared to a Unix socket fd; in Arvo terms, it's an opaque encoding of the `duct` (cause stack) of the sending event; in protocol terms it's a half-duplex stream selector (incoming bones are not related to outcoming bones).

The message content is a general noun `r`, routed to an internal path `q`. We have solved the problem of delivering a noun to a higher level of Arvo.

A `$part` meal is a fragment of a larger superpacket. All fragments must agree on the full details of the superpacket, which must expand to a `$bond`. Generally a part is sent with `$none` encoding (no encryption).

A `$back` meal is a positive or negative acknowledgment, depending on its `(unit hork)`. Acknowledgments are packet level and end-to-end; the packet that completes a `$bond` is not acknowledged until the message is fully processed,

and contains a negative ack if message processing failed. This message-level transaction acknowledgment falls back into Arvo on the sender's side.

For example, if the message pokes an application, and that application's poke arm crashed, the negative ack contains the crash report with stack trace. We don't know that the client user wants to see this, but we don't know that she doesn't.

Ack handling in general follows the first law of acknowledgments, which is: always ack a dup, never ack an ack. (We do sometimes ack an ack to move along a key exchange.) In particular, the set of all negative acks must be saved indefinitely; a duplicate packet must produce a duplicate ack.

A `$fore` meal is a forwarded packet. Along with the packet itself (with its end-to-end sender and destination unencrypted in the packet envelope), we have a `(unit lane)` which indicates the UDP address that the first forwarding server received it from.

Either the packet sender has full cone NAT, or not. If so, the UDP address reported by the forwarding service will work for any sender; if we respond directly to that UDP address, and the packet goes through, the sender will get our direct UDP address. In this case we've accomplished a STUN style holepunch.

But we don't know that this will work. When we have an indirect or inactive UDP address for a ship, we send duplicate packets: one direct, one through the forwarding hierarchy. Only when we receive a direct packet do we stop doing this. So until STUN works, we do TURN – at the cost of some wasted packets.

The proper way to forward a packet is to send it to the nearest child if you're an ancestor of the target, to your parent if not. The peer leg of the path is at the galaxy level by default, but stars or anyone can short-circuit.

9.7 Poke processing

Again, from a `$bond` we get a path and a noun. The prefix of the path defines an Arvo vane, eg, `%g` for the app system `%gall`. For a poke, the path is `[%g app %poke ~]`, where `app` is an application name (eg, the shell `dojo`).

The noun will be a cell `{p/mark q/data}`. A mark is a `term` (symbol) which is the Urbit equivalent of a MIME type, if MIME types were names of typed validation functions. The mark is mapped into a `%clay` search path, in the same desk (`%clay`'s equivalent of a branch) as the application.

This path leads to a Hoon source file containing a core that defines a variety of functions on nouns in the mark, such as diffs and format translations. The simplest such function is a mold for validation.

Like a MIME type, the mark is just a label. There is no way to guarantee that the sender and receiver agree on what this label means. A noun which doesn't normalize to itself is a packet drop. (We could use molds to implement the Postelian behavior of repairing corrupt input, but this is bad policy in a typed networking environment.)

Otherwise, we have a typed noun. For a mark `foo`, this becomes the sample to a `poke-foo` gate in the app core. There is no endpoint name separate from the mark name; in a sense, there is one poke endpoint overloaded by mark.

If this call doesn't crash, the message is acknowledged by default. But the product of the application's poke gate is a list of Arvo moves. One move the application can make is a `%wait` gift that tells `%ames` to delay the acknowledgment. A further gift reports final completion (or failure). An application can therefore delegate the transaction to another service, which is always dangerous but sometimes necessary. In the meantime, the end-to-end packet ack will be withheld and the sender will back off.

10 System status and roadmap

Urbit works and self-hosts. The full, running stack, with compiler, standard library, vanes, and distributed messaging and shell applications with both console and Web interfaces, is about 30,000 lines of code. This is probably too big, since Hoon (like most functional languages) is quite compact and expressive.

Some components remain alpha-grade, some are not quite up to spec. The boot sequence as implemented is not as defined; security masks are not implemented *at all*; etc. Most system updates at all levels (Hoon, Arvo, apps) are propagated over the network, but sometimes we still reboot the universe (declare a flag day, or "continuity breach") for a particularly gnarly one. The event log is not at all above suspicion. Documentation is particularly weak. Performance is usable, but hardly impressive. So Urbit is not yet production code, but nor is it absurdly far away.

The Urbit repository is at `www.github.com/urbit/urbit`. The website (hosted by Urbit) is at `www.urbit.org`.

11 Inadequate summary of related work

Many historical OSes and interpreters have approached the SSI ideal, but fail on persistence, determinism, or both. In the OS department, the classic single-level store is the IBM AS/400 [18]. NewtonOS [19] was a shipping product with language-level persistence. Many image-oriented interpreters (eg, Lisps [20] and Smalltalks) are also SSI-ish, but usually not transactional or deterministic. And of course, many databases are transactional and deterministic, but their lifecycle function is not a general-purpose interpreter.

12 Conclusion

Urbit is cool and you should check it out.

13 Acknowledgments

We thank our coworkers, Galen Wolfe-Pauly and Henry Ault, and everyone else who has committed to the Urbit repository.

14 Appendix

Sieve of Eratosthenes in keyword syntax

```
:gate  thru/atom
:cast  (list atom)
:var   field/(set atom)  (silt (gulf 2 thru))
:rap   abet:main
:core
++  abet
  (sort (~(tap in field) ~) lth)
::
++  main
  :var   factor/atom  2
  :loop  :like  ..main
  :if    (gth (mul factor factor) thru)
    ..main
  :moar(factor +(factor), ..main (reap factor))
::
++  reap
  :gate  factor/atom
  :var   count/atom  (mul 2 factor)
  :loop  :like  ..reap
  :if    (gth count thru)
    ..reap
  :moar
    count  (add count factor)
    field  (~(del in field) count)
  ==
--
```

Sieve of Eratosthenes in rune syntax

```
|=  top/@
^-  (list @)
=+  fed=(silt (gulf 2 top))
=<  abet:main
|%
++  abet  (sort (~(tap in fed)) lth)
++  main
  =+  fac=2
  |-  ^+  ..main
  ?:  (gth (mul fac fac) top)
    ..main
  $(fac +(fac), ..main (reap fac))
::
++  reap
  |=  fac/atom
  =+  cot=(mul 2 fac)
  |-  ^+  ..reap
  ?:  (gth cot top)
    ..reap
  $(cot (add cot fac), fed (~(del in fed) cot))
--
```

References

[1] Tahir Ahmad and Usman Younis. Randomness testing of non-cryptographic hash functions for real-time hash table based storage and look-up of urls. *Journal of Network and Computer Applications*, 41:197–205, 2014.

[2] J. N. Gray. *Operating Systems: An Advanced Course*, chapter Notes on data base operating systems, pages 393–481. Springer Berlin Heidelberg, Berlin, Heidelberg, 1978.

[3] Henry F Huang and Tao Jiang. Design and implementation of flash based nvdimm. In *Non-Volatile Memory Systems and Applications Symposium (NVMSA), 2014 IEEE*, pages 1–6. IEEE, 2014.

[4] Van Jacobson, Marc Mosko, D Smetters, and Jose Garcia-Luna-Aceves. Content-centric networking. *Whitepaper, Palo Alto Research Center*, pages 2–4, 2007.

[5] Ghassan O Karame, Elli Androulaki, and Srdjan Capkun. Two bitcoins at the price of one. *Double-Spending Attacks on Fast Payments in Bitcoin. IACR Report*, 248:1–17, 2012.

[6] Robert C Martin. The liskov substitution principle. *C++ Report*, 8(3):14, 1996.

[7] Nicholas D Matsakis and Felix S Klock II. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, 2014.

[8] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[9] Bertrand Meyer. *Object-oriented software construction*, volume 2. Prentice Hall New York, 1988.

[10] Ron Morrison, Richard Connor, Graham Kirby, David Munro, Malcolm P Atkinson, Quintin Cutts, Fred Brown, and Alan Dearie. The napier88 persistent programming language and environment. In *Fully Integrated Data Environments*, pages 98–154. Springer, 2000.

[11] Phillip Rogaway and Thomas Shrimpton. A provable-security treatment of the key-wrap problem. In *Advances in Cryptology-EUROCRYPT 2006*, pages 373–390. Springer, 2006.

[12] Jonathan S Shapiro and Jonathan Adams. Design evolution of the eros single-level store. In *USENIX Annual Technical Conference, General Track*, pages 59–72, 2002.

[13] Mark Tarver. *Functional programming in Qi*. Free University Press, 2008.

[14] Mark Tarver. *The Book of Shen*. FastPrint Pub., 2013.

[15] Ka-Ping Yee and Kragen Sitaker. Passpet: convenient password management and phishing protection. In *Proceedings of the second symposium on Usable privacy and security*, pages 32–43. ACM, 2006.

[16] Google Inc. Protocol Buffers. `https://developers.google.com/protocol-buffers/`

[17] Ted Neward. The Vietnam of Computer Science `http://blogs.tedneward.com/post/the-vietnam-of-computer-science/`

[18] Frank G. Soltis. Inside the AS/400 ISBN 1882419669, 1997

[19] Apple Computer. NewtonOS 2.1 Engineering Documents `http://newted.org/download/manuals/NewtonOS21EngDoc.pdf`

[20] Janet H. Walker, David A. Moon, Daniel L. Weinreb, and Mike McMahon "The Symbolics Genera Programming Environment", IEEE Software 4.6 (1987)

[21] Tyler Treat. You cannot have exactly-once message delivery `http://bravenewgeek.com/you-cannot-have-exactly-once-delivery/`

[22] C2 Wiki. "SufficientlySmartCompiler" `http://c2.com/cgi/wiki?SufficientlySmartCompiler`