half-title
Urbit Systems
Technical Journal

title/toc

First paperback edition April 2024.

urbitsystems.tech
urbit.org

Published by West Martian Limited Company, Cheyenne, Wyoming.

www.westmartian.com

Welcome to the inaugural issue of the *Urbit Systems Technical Journal: The Journal of Solid-State Computing*. The core developers of Urbit from 2013 onwards have done yeoman's work in specifying and producing a novel and inventive system, one that implements and explores dozens of new concepts in computer science. By and large, Urbit has functioned as an event horizon—when someone is convicted by the project's ambition, they are drawn in and their work becomes less visible to the broader software development community. USTJ will change that by showcasing our travails, challenges, and triumphs.

We've wanted to write about Urbit for years. It never felt like the timing was right for the effort of producing a classical reference or textbook: Urbit was too hot (in the kelvin sense) and some of the prodigious work of producing a volume would have to be repeated for every system release. There are deep and true things that can be said about the platonic Urbit, the diamond Urbit, but much of what we are working through now is a contingent Urbit, feeling our way towards zero kelvin. A technical journal is more forgiving in all the senses we want: it is time-resolved; it is episodic; and it allows deep rabbit holes that would never fit in a book. Rather than compromise for a textbook, we can instead find ways to start saying every important technical thing about Urbit and solid-state computing.

USTJ will follow a permissive dictum: "Therefore every scribe which is instructed unto the kingdom of heaven is like unto a man that is an householder, which bringeth forth out of his treasure things new and old" (Matthew *xiii* 52). In these pages you will read groundbreaking new work as well as well-considered expositions from Urbit's development. Note as well the plural in the title. Deterministic computing and secure computing are all larger fields than Urbit alone.

Many have contributed, not least the authors of the code and articles. I would particularly like to thank `~mopfel-winrux` for his early enthusiasm for the concept of USTJ; `~wolref-podlex` for paving the way; each contributor, reviewer, and designer; and Simon DeDeo, who first suggested to us the idea of a technical journal.

Computer science happens in the trenches. ⊠

The State of Urbit: Eight Years After the Whitepaper

Ted Blackman ~rovnys-ricfer
Urbit Foundation

Abstract

The Urbit whitepaper was released in 2016. In the intervening eight years, the project has become much more fleshed out: the sketch has started to take on color and detail. This article explores the major developments since the whitepaper, including changes to the runtime, the network, the kernel, and the userspace. It also discusses the current state of the project and intended refinements.

Contents

Urbit Systems Technical Journal I:1 (2024): 1–46.

Address author correspondence to ~rovnys-ricfer.

1 Introduction

The Urbit whitepaper (~sorreg-namtyv et al., 2016) publicly elaborated the technical prospectus for a solid-state operating function, an operating system built as a simple state machine defined as a pure function of its events. While much of the whitepaper was concerned with establishing the philosophical objectives of the project, there was also substantial exposition about key parts of the Arvo event handler, the language, and

the network. Given that the whitepaper was a snapshot of the project and its ambitions as it stood in 2016, how has that vision held up in practice, and how has the project fared?

While the whitepaper by its own admission presented itself as a "work-in-progress presentation" of a "semi-closed alpha test", the basic claims of the Urbit whitepaper are still true today. Most of the intervening work over the past eight years has gone into improving robustness and better support for applications. A capsule history of "late early" Urbit includes the following milestones:

- 2015–2017: Early vane explosion
- 2017–2019: Rewrites of Hoon, Ford, Eyre, and Vere refine the system
- 2018–2019: Rewrites of Azimuth and Ames refine the network stack
- 2018–2019: Nock bytecode interpreter
- 2020: Ford Fusion enables network continuity
- 2020: Security audit on Ames
- 2021: Azimuth Layer2
- 2021: Software distribution introduced
- 2022–2024: Software distribution maturation
- 2023–2024: Network scaling

This article will discuss the changes to Urbit since the whitepaper, focusing on the runtime, the network, the kernel, and the userspace. It will also discuss the current state of the project and intended refinements.

2 Changes to Urbit

Briefly, the lower layers of the system have deviated very little from their descriptions in the whitepaper. Nock (the machine

code), Hoon (the programming language), and Arvo (the operating system kernel) – which implies atomic event processing and orthogonal persistence – are very much the same system today as they were even a decade ago.

Since 2016, Nock has undergone a revision from kelvin version 5K down to 4K. This was a refinement rather than a deviation, mainly adding an opcode to manage tree mutation (see appendix for details). Hoon has undergone a number of changes but is certainly recognizable as the same language.

At the next layer up, the Arvo kernel, despite having been rewritten at least once, serves the same purpose and has a similar shape to the one that existed at the time of the whitepaper.

The system has primarily changed via addition, mostly growing upward in the stack: runtime and kernel infrastructure to support distributing, installing, running, and upgrading userspace applications. Changes have been made to peer-to-peer networking, HTTP serving, source code management in Clay, and Arvo kernel upgrades. Permanent network continuity was achieved at the end of 2020, largely by deleting the Ford build system vane, integrating the build system into the source control system in the Clay vane, and modifying the Arvo kernel's upgrade semantics.

Urbit's "scry" data namespace is also much more real and fleshed out than it was when the whitepaper was written. The concept of scrying predates Urbit itself (`~sorreg-namtyv` (2006); compare `~sorreg-namtyv` (2010)), but was not used heavily by applications.

The other big change is the public key infrastructure (PKI). The whitepaper's PKI, which was effectively live from 2013 to 2019, had no double-spend protection and was effectively centralized. To fix this, the PKI was moved to a set of Ethereum contracts. Going on-chain made the ownership ledger for Urbit IDs censorship-resistant … until a year later when high transaction fees on Ethereum brought network growth to a standstill and the core developers were forced to write a custom "naive rollup" Ethereum contract. This brought the price of spawning a new Urbit node down from hundreds of dollars to under a dollar, at the cost of on-chain interoperability.

Going on-chain also enabled "personal continuity

breaches", in which a ship (instance) owner can notify the network through the chain that their Urbit node was rebooted from scratch, due to data loss, unrecoverable error, or compromised security. This feature necessitated some rework of the kernel to ensure data consistency across vanes when receiving a notification that a peer had breached.

Finally, a ton of work has been done on Urbit's runtime. Lots of 2016 vaporware is corporeal in 2024.

2.1 Solid-Stateness

> 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. (~sorreg-namtyv et al., 2016)

Urbit is a solid-state interpreter but some of its higher layers are more solid-state than others. The scry namespace, hardly mentioned in the whitepaper, provides the primary means of preserving solid-stateness higher up the stack, especially into userspace.

A program is considered more solid-state if it has fewer moving parts, fewer complex asynchronous code paths, less transient state, more idempotence, more immutability, more referential transparency, clearer transaction boundaries, less ordering dependence, more declarativeness, and more determinism.

Much of Urbit's kernel modules and application model were originally written in ways that were less solid-state than they could have been. As the reliability requirements for Urbit have increased, more attention has been devoted toward making the system more solid-state, as the main pathway toward increasing simplicity and reliability. More attention has been paid to the scry namespace in particular, to increase its capabilities to allow it to be used more heavily.

2.2 Clay "Theory of a Desk"

A relatively early change that increased the solid-stateness of the system was called the "Theory of a Desk". This was implemented as part of the Ford Fusion project in early 2020. Ford, the build system, was moved from being a standalone vane (kernel module) into the Clay source management vane. When Ford stood alone, despite – or maybe exacerbated by – multiple rewrites, it was difficult to achieve full correctness of source builds during a kernel upgrade. It was common for post-upgrade files to be built erroneously using the old kernel. This behavior was effectively nondeterministic because it relied on the ordering of short hashes ("mug"s) that identified subscribers to Clay file-changed notifications. Sometimes it would work correctly by luck; other times it would fail in various hard-to-reason-about ways.

Moving Ford into Clay removed asynchrony from the system: instead of Ford requesting files one by one from Clay and building them, Ford became a subroutine in Clay that Clay could call synchronously. This "fusion" allowed the Ford build semantics to be modified to depend only on files in the current revision of the "desk" (Clay's version of a Git repository).

Previous versions of Clay only allowed files of a newly defined filetype ("mark") to be added *after* a commit that added a "mark file" that defined the semantics of that filetype. This meant that no Clay revision was ever truly self-contained: a deterministic understanding of how its files would be validated required knowledge of the previous commit as well.

The "theory of a desk" paradigm uses only the current snapshot of the desk to build any mark files needed to validate the files in the desk. This was not technically impossible while Ford was a separate vane, but the asynchronous communication back and forth made it exceedingly difficult. The new Ford shrank from 6,000 lines of code down to about 500, and became much easier to reason about and debug.

A year later, near the end of 2021, the "software distribution" project completed, allowing third-party developers to distribute userspace applications to their users over the Urbit network, storing and building the code in Clay. This necessitated

reproducible builds, which is a notoriously solid-state requirement. Ford Fusion builds were reproducible, but the system had to be extended quite a bit to practically support multiple userspace installations. These extensions caused a new set of problems.

The interaction between Gall (which runs applications) and Clay (which stores and builds their source code) was also complex, asynchronous, and bug-prone. A year after software distribution, in the second half of 2022, the "agents in Clay" project moved the locus of control over which userspace applications were supposed to be running from Gall itself to Clay. This allowed Clay to send a single "move" (effect) to Gall containing the full list of all agents that Gall should now be running.

Note that this latest iteration, with a single large effect, is more declarative, has less transient state, is more deterministic, has less ordering dependence, and is more idempotent – in other words, significantly more solid-state. As expected, this system has been much more reliable than the initial software-distribution system.

2.2.1 Remote Scry Protocol

A major part of making the namespace more usablae was to expose namespace queries over the network. Designed in 2021 and deployed in 2023, the "Fine"[1] remote scry network protocol was added to Urbit. The first use cases were Clay, which uses it for downloading desks, and the Gall `%grow` namespace, described in more detail in a later section.

This protocol lets one ship fetch data at a namespace path bound by another ship. The runtime includes a cache so later requests for the same data are typically served from the runtime cache rather than requiring Arvo's `+peek` arm to be called again. In addition to the scalability benefits this provided, it also represented a fairly large conceptual shift: instead of Urbit's networking only being able to express commands (writes) over the network, it could now also support reads.

[1] Pronounced *fee-NAY*, after French cartographer and mathematician Oronce Fine.

These reads are from the immutable namespace, so they're quite solid-state.

Two-Party Encrypted Remote Scry: `%chum` The remote scry protocol authenticates responses (scry bindings), but it does not have any concept of encryption built in – everything published over the network is public. Since most Urbit applications deal with private data, there needed to be a way to distribute data privately. The `%chum` namespace overlay was the first way to do this.

The `%chum` namespace overlay allows a ship to make an authenticated and encrypted scry request to another ship. Only the publisher ship can decrypt the request, which is symmetrically encrypted using the Diffie–Hellman of the two ships' Azimuth networking keys. The scry response is also encrypted symmetrically, so only the requesting ship can decrypt it. Crucially, this system uses the Arvo kernel's security mask (which was described in the whitepaper but not enforced until this protocol made use of it) to inform the publishing module about which ship is making the request.

The first version of this system uses the Fine protocol unmodified, although the Directed Messaging project's implementation has explicit support for symmetric encryption, which it also uses for pokes (commands).

Multi-Party Encrypted Remote Scry The symmetric encryption of the `%chum` namespace overlay has a problem in a multi-party context like a social group: no two requesting ships can share entries in the runtime cache, meaning each subscriber has to make a separate call to Arvo's `+peek` arm. To fix this, a multi-party form of encrypted remote scry was added.

To use this system, a publishing application must register a scry path prefix as a security domain, to which the kernel associates a (rotatable) symmetric key. Subscriber ships request access to this key. The publisher ship's kernel asks the publishing application whether that ship should be allowed to view the content at that path prefix. Once the subscribing ship has the key for a path prefix, it requests data under that prefix by

encrypting the request path and placing the key's identifier in cleartext so the publishing ship knows which key to use to try to decrypt the path.

Querying the publishing application dynamically like this allows the publishing application to avoid materializing the set of all ships who are allowed to see this content. Materializing that set in a complex application with layers of roles, whitelists, blacklists, and other criteria could be prohibitively slow, superlinear in the number of allowed subscribers. This arrangement should allow the application to keep the query time to roughly logarithmic in the number of subscribers, which is much more practical.

The next steps in increasing the utility of the scry namespace include `%pine` (a non-referentially-transparent "request-at-latest" network protocol) and "sticky scry", which would facilitate scry-based network subscriptions by allowing requests for unbound data to persist on a publisher until the data is published, at which point it will be pushed down to subscribers with low latency.

2.3 Gall `%grow` Namespace

Gall agents were not able to directly bind data into the scry namespace until the `%grow` namespace was added in early 2023 in the same release as the first remote scry protocol. A Gall agent can emit an effect to bind a path to a value in the `%grow` namespace.

This allows applications to publish data that can be read remotely using the remote scry protocol, making applications more legible and solid-state.

2.4 Persistent Nock Memoization

For around a decade, Nock interpreters have supported the `%memo` memoization hint, informing the interpreter that the enclosed Nock computation should be cached. Since Nock is purely functional, these caches will never need invalidation, an important property. However, until early 2024, runtime Nock

caches were ephemeral – they did not last past a single Arvo event, or were even more short-lived.

Persistent Nock caching allows for removal of most caching-related state from the kernel and applications, allowing code to focus on business logic. Making more logic stateless contributes strongly to solid-stateness. Persistent Nock caching also opens up new architectural possibilities, such as userspace build systems that share a runtime cache to avoid rerunning compilations or duplicating libraries, even if they have varying semantics.

2.5 Breadth-First Arvo Move Ordering

Arvo has traditionally used "depth-first" ordering when processing "moves", i. e. messages passed between parts of the system. Since each module in Arvo emits a list of moves whenever it runs, and then the moves are delivered one at a time to their destination modules for processing, at any given time there is a tree of unprocessed moves that have been emitted but not delivered.

Processing moves in depth-first order is planned on being changed. A version of Arvo that instead uses breadth-first move order has been written and is intended for deployment in Zuse 410 or 409. This remains somewhat controversial, though.

Depth-first move ordering has the advantage of a certain kind of consistency. Consider the case where as the result of a single activation, Gall agent `%foo` "pokes" (sends a command to) `%bar` and another to `%baz`. With depth-first move order, any moves emitted by `%bar` upon receiving `%foo`'s poke will be processed before `%baz` receives `%foo`'s poke. This applies recursively, so if upon receiving `%foo`'s poke, `%bar` poked `%qux` and `%qux` poked `%qux2`, all that still happens before `%baz` receives `%foo`'s poke.

This means `%foo` can rely on `%bar` completely handling the first poke before `%baz` receives the second poke, even if `%bar` has to issue further commands in order to do so – at least as long as all the pokes are, one, domestic (in the same ship) and two, processable synchronously, i. e. no operations need to be spread across multiple Arvo events, such as waiting for a timer

to elapse or waiting for a user to type input on the command line. If both these conditions are met, then the poke to `%bar` has one view of the system's state, and the poke to `%baz` has a newer, updated view, but both views are consistent in the sense that for either one of them, `%bar`'s move processing has either not started at all or completely finished.

A major issue with depth-first move order is that many common interactions, including anything social that involves peer nodes, cannot preserve both of those properties that let each module send commands to other modules without acknowlegment but still rest assured that those commands will complete in order and with each having a consistent view of the state. This means the system has needed to incorporate the concept of an acknowledgment for a poke, since a poke could be over the network rather than local. In the current formulation, local pokes are also acknowledged, losing some of the streamlining one would hope for with depth-first ordering.

Another problem with depth-first ordering is that it makes consistency among modules difficult to guarantee. Jael delivers a notification that a peer has breached to Ames and Gall. When Ames hears the notification, it deletes all state regarding the old pre-breach ship and updates its state to store a cache of the peer's new post-breach PKI state (public key, etc.). When Gall hears the notification, for each agent that had subscriptions to the old ship, it closes the subscriptions and notifies the agent that the subscription was closed.

The problem comes up when an agent wakes up with the notification that its subscription was closed and emits a move to resubscribe. Gall then sends this move to Ames. If Gall heard the breach notification before Ames, then the resubscribe request would be sent to the old pre-breach ship, since in depth-first move ordering, Ames hears Gall's requests before Jael's breach notification. Since the ship had breached, it ignores the resubscribe request to its old key.

This can be band-aided by baking the specifics of the interaction into Jael, so it knows to tell Ames about the breach before Gall. This approach works as long as the data dependencies among modules remains acyclic, but as soon as two modules could both send moves to each other upon being notified

– meaning they need to be notified "at the same time" – it becomes nontrivial to provide consistency. It could be achieved through a two-phase commit system where Jael delivers two moves for each notification, and the receiving modules would know to update their state in response to the first notification but only emit moves upon hearing the second notification.

The final problem with depth-first ordering is that it is empirically difficult to reason about. Moves are not processed chronologically – later moves can jump to the front of the queue. This has led to a number of subtle bugs due to later moves invalidating earlier moves, such as canceling a subscription that hadn't been established yet because the unsubscribe move jumped in front of the subscribe move.

An alternative has been built which uses breadth-first move ordering. This is strictly chronological, does not reorder moves, has a concept of simultaneity for maintaining consistency when delivering notifications, and works equally well locally and over the network. The major disadvantage of breadth-first ordering is that there is no automatically growing transaction: everything requires explicit acknowledgment.

The latest "shrubbery" (q. v.) experiment uses a hybrid move ordering: depth-first for local pokes within the same application, and breadth-first otherwise.

2.6 Shrubbery

"Shrubbery" refers to an architectural concept in which all userspace data is addressible in the scry namespace. This would include source code and application state in addition to inert file-like data.

A well-functioning shrubbery vane would replace Clay and Gall, unifying the storage and management of data and code into a single namespace-backed system.

A shrubbery model has the potential to make application programming significantly more solid-state. Each piece of data in a shrubbery system has a canonical name to which it is immutably bound. The state of pending I/O will also live in the namespace, facilitating inspection and recovery after a suspension or breach. Subscription and state synchronization become

declarative. Functional reactive programming features would allow for declarative composition of programs.

2.6.1 Move Ordering in Shrubbery

The latest "shrubbery" experiment attempts a hybrid solution to get the best of both modalities. A typical shrubbery "app" will be broken into a hierarchy of event handlers at various subpaths. When sending a move "downward" into a subpath, the system will handle that move depth-first, since an app is presumed to know the characteristics of its sub-apps. These moves will not need to be acknowledged, since they will be truly depth-first. When passing a move "upward" or "sideways", i. e. outside of the sub-tree, that move will be appended to one global queue, to be handled breadth-first.

This hybrid approach can be thought of as the lowest layers of the move tree being depth-first, with breadth-first at higher layers. Core developers expect the shrubbery experiment to be runnable within a month of this writing.

3 Continuity

Since the whitepaper was written in 2016, quite a bit of effort has gone into improving Urbit's continuity. Continuity in this context might best be defined in contrast to its opposite: a "continuity breach" refers to the act of restarting an Urbit from scratch, throwing away all its previous data and the states of its connections with other ships.

Until the end of 2020, core developers would routinely conduct network-wide breaches. Every ship on the network would need to be restarted from scratch in order to communicate with other ships.

For a user, the experience of discontinuity is one of loss of data and possibly even of friends. This document will use the term "continuity" in a broad sense.

3.1 Azimuth: On-Chain PKI (2019)

One major event that improved the level of continuity was moving Urbit's public key infrastructure (PKI) from an informal spreadsheet maintained by the Tlon Corporation to a set of smart contracts on the Ethereum blockchain (`~ravmel-ropdyl`, 2019).

This ensured Byzantine fault-tolerant ownership of an Urbit ship and permissionless entry – anyone can buy an Urbit address, without needing to communicate with any particular person or organization. This had always been the goal, but it wasn't feasible until programmable smart-contract blockchains became production-worthy.

In addition to moving the address ownership and sponsorship information on-chain, other features were added.

3.1.1 Azimuth: Individual Continuity Breaches

The capability to perform an "individual continuity breach" was added. Before going on-chain, if an Urbit suffered any major data loss, it would be permanently dead and could never communicate with peers again. An individual continuity breach allows a user to post a transaction to chain that increments the "rift" number for their address, signaling to the network that the ship has been restarted from scratch. Peer ships that were storing message sequence numbers for the breached ship reset their networking state for the breached ship.

This allows a user to continue to use the address they purchased even if their Urbit loses data. The newly written `%phoenix` app implements a system of encrypted backups, so that after a breach, a user can recover application data from the latest backup (`~midden-fabler`, 2024).

3.1.2 Azimuth: Constitution

The Urbit galaxies have always been considered to form a senate, acting as the Shelling point for network governance. Before going on chain, this was merely an idea – now it is a formalized reality. Galaxies can vote on-chain, using a smart contract that has control over the other smart contracts.

There are two ways galaxies can vote: on (the hash of) a document to signal support of that document, and on a new constitution contract, which will trigger the old constitution contract to cede control to the new constitution contract that replaces it. This means the galaxies can vote to upgrade the constitution as they see fit – an elegant example of social meta-circularity.

It should be noted that galactic governance is purely voluntary: because users have full control over their own personal servers, they retain the power to "vote with their feet" and decide to use a different PKI or a different governance structure.

It is incumbent upon the galaxies to make decisions that are prudent enough not to fork the network – or prudent enough to fork the network, if called for. Using a blockchain means a fork could be performed in an orderly manner without losing consistency. This added technical feasibility of forking puts virtuous pressure on the galaxies to govern reasonably.

3.1.3 Azimuth: Naive Rollup

In 2020, Ethereum transaction fees rose to levels high enough that Urbit network growth ground to a halt. The core developers, after evaluating then-nascent "zero-knowledge" and "optimistic" rollup technologies for reducing fees, determined they were not yet production-worthy and wrote a simpler rollup that they called a "naive rollup". This brought the price of spawning a planet down from roughly $100 to under $1.

The naive rollup treats Ethereum as an inert data store for storing a series of transaction inputs. Each Urbit ship then reads those transaction inputs from Ethereum and executes them itself, arriving at the latest state of the rollup. This can be thought of as a simplification of an optimistic rollup, where the chain does not have any representation of the latest state of the rollup, only the inputs that would lead to that state. This obviates the need for fraud proofs and bond slashing, yielding a substantially simpler system at the cost of some interoperability between Urbit addresses in the rollup and other Ethereum contracts, which do not have any cryptographic proof on chain regarding which Ethereum address owns those addresses.

Each ship's Azimuth state is stored either in the rollup or in the original contract. Galaxies cannot be moved into the rollup. For the time being, addresses cannot be moved back out of the rollup.

3.2 New Ames & New Jael (2019)

In order to take advantage of the new PKI, the Ames networking vane and Jael secret storage vane both had to be rewritten. This was done in 2019, with new Ames being deployed later, near the end of the year.

The "new" Ames and Jael – now in their fourth year of use, with Ames having had some refactoring and extension – support individual continuity breaches and on-chain sponsorship changes ("escapes").

3.3 Ford Fusion (2020)

As described earlier, the Ford Fusion project moved Urbit's build system from a standalone vane to a synchronously callable function inside Clay and rectified over-the-air Arvo kernel upgrades.

This was crucial for performing the last network breach at the end of 2020, establishing permanent continuity – since then, many dozens of over-the-air upgrades have been deployed. Without a reliable upgrade process, this continuity era would be infeasible.

3.4 Essential Desks (2023–24)

Having gained experience with managing deployments of both kernelspace and userspace desks, it became clear that the original way that kernel updates synchronized with app updates needed more flexibility. Any app could block a kernel upgrade if its developer had not also pushed out a new version that was compatible with the new kernel. A user could always suspend a stale app to allow the upgrade, but this had to be done manually.

A later update modified this behavior. A newly installed app is now marked as non-essential by default. The kernel will only block an upgrade if an essential app is stale. A stale non-essential app will be suspended automatically on upgrade. If a user wants to ensure an app will never be suspended, they can mark it as essential.

3.5 Future Work

The next major frontier in continuity will be "kelvin shims", i. e. a kernel that can keep stale apps designed to work with previous kernels running (through a "shim" for a previous kelvin version), not just apps designed to run at exactly the current version. This is a thorny problem that the core development team experimented with in early 2023 but abandoned.

The hard part of kelvin shimming turned out to be that the current version of the userspace/kernelspace API is complex. Taming that interface and localizing application-related state to make it easier to migrate will be critical for enabling kelvin shims. The shrubbery experiment is a step in this direction.

More sophisticated kernel support for migrating applications is also likely to be a fruitful line of work. This involves migrating application state from an old type to a new type, as well as supporting protocol negotiation to upgrade or downgrade commands and subscription updates to maintain compatibility between old and new versions of the same app on different ships.

4 Maturation

Urbit has matured along several dimensions since the whitepaper was written. This work can be roughly categorized into organizational changes, scaling improvements, reliability work, work on the runtime, and adding an explicit control plane.

4.1 Changes to Development Practices

Since 2016, a much larger number of developers are involved in core development, and the development practices have evolved accordingly.

4.1.1 Organizational Changes

In early 2023, the responsibility for core development transferred from the Tlon Corporation, which had managed it since 2013, to the newer Urbit Foundation, a nonprofit. The Urbit Foundation has become the custodian of the Urbit kernel and ecosystem. UF adopted and continued to modify Tlon's release workflow, including CI/CD (continuous integration/continuous deployment), multi-stage canary deployments, and repository management practices. This process has more recently been extended to include a liaison period for each kelvin update where organizations that use Urbit can use the prerelease software on their own infrastructure first and report any bugs to UF.

4.1.2 Urbit Improvement Proposals

The Urbit Improvement Proposal system was introduced in mid-2023. This is an RFC system based heavily on Bitcoin's BIP system and Ethereum's EIP system. On a biweekly basis, proposals are reviewed by core developers in group calls. Since its launch, over twenty UIPs have been written, of which thirteen have been ratified and built.

4.2 Scaling

In 2016, the Urbit network could barely handle a few hundred online ships. In 2024, at 411 kelvin, the network can easily handle many tens of thousands of online ships, if not more. Packet forwarding is now stateless and does not run Nock. Sponsor pinging now uses the STUN protocol, incurring no disk writes on sponsoring galaxies.

The Directed Messaging project will increase network throughput by roughly a factor of a thousand, fast enough to

saturate a gigabit ethernet link. The Ares project will increase the maximum persistent data storage by a factor of several thousand, up to 64 terabytes. Ares should also improve Nock execution speed dramatically, paving the way toward Nock being able to be compiled to machine code comparable to that emitted by a C compiler.

Heavy use of the scry namespace facilitates scalable data dissemination through the network, following a diffusion model, where intermediate relays can cache arbitrary data, minimizing duplicate packets across the network.

Future work will likely include multithreaded scry requests, shared-memory interprocess communication in the runtime, and more advanced Nock interpreter optimizations.

4.3 Reliability

All through 2017 and 2018, Urbit would crash for the typical user on a roughly weekly basis. As of 2024, that frequency seems to be anecdotally once every few months, usually from running out of disk space due to excessive event log buildup; this has already been addressed on NativePlanet's systems, which perform automatic event log rollover and truncation.

Kernel and app upgrades are much more reliable. The system runs out of memory much less often. The network maintains connectivity much better. Commonly used applications have fewer bugs. Browser page loads are much faster and more reliable. Individual continuity breaches cause fewer downstream bugs in the kernel and applications. Arvo error handling has fewer known flaws. Gall subscriptions underwent a major rearchitecture in 2022 to fix a queue desynchronization bug. The Behn timer module was rewritten multiple times to shake out several different bugs.

The list goes on. While Urbit still has flaws and occasional crashes, its level of robustness is strikingly higher than it was when the whitepaper was written. By early 2025, we expect Urbit to be reliable enough to pass a security audit that includes penetration testing. At that point, custodial cryptographic applications will be viable on the platform.

4.4 Runtime Work

The runtime has absorbed a large percentage of the work that has been done in the system in the last eight years. The naive tree-walking Nock interpreter was replaced by a custom bytecode interpreter. The system was made to recover from out-of-memory errors and automatically reclaim memory. A memory allocator trick called "pointer compression" was used to increase data storage from 2 GB to 8 GB while remaining a 32-bit interpreter. Demand paging was introduced, reducing the cost of running an Urbit ship from over $10 per month to $0.15 per month on some hosting platforms.

The runtime was broken into two processes, allowing I/O and Arvo event processing to run in parallel. The event log was moved out of a custom file into an LMDB database, then broken up into "epochs" allowing for event log rollover and truncation. The I/O drivers in the runtime have continued to be developed, including full rewrites of the HTTP, Ames, and terminal drivers.

The Ares project, close to initial deployment as of writing, includes a clean-slate rewrite of the Nock interpreter, using an all-new allocator, a novel static analysis technique called Subject Knowledge Analysis to achieve substantial Nock execution speed improvements, and a new persistence module called the Persistent Memory Arena, using a copy-on-write B+ tree to keep memory pages synchronized between disk and RAM.

4.5 Control Plane

Since Urbits are now commonly run in one of a few cloud hosting platforms, the Khan I/O driver and vane were added, giving Urbit an explicit external control plane. Khan can manage runtime lifecycle events and send arbitrary commands and queries to a running ship, all over a Unix domain socket. This has facilitated cloud orchestration of large fleets of Urbit ships. NativePlanet's GroundSeg user-facing ship management interface (NativePlanet, 2024) also uses Khan, in a self-hosted setting.

5 Extensibility

A large portion of development efforts since the whitepaper has gone into various improvements in making the system extensible. Mostly this has been in the direction of userspace applications, but some runtime extensibility has also been added.

5.1 Software Distribution

In 2021, a major incision was made into Clay, Gall, Kiln (the userspace agent that manages agent upgrades), and the Arvo kernel to enable application developers to publish bundles of source code that could be installed and run as applications.

Before that, a user would have to run a command-line merge in order to install any software other than the base distribution. This effectively meant forking the source code of your ship, and it commonly caused downstream issues when trying to stay up to date. Within a few months of launching this feature, several Urbit application companies formed to take advantage of the new capability.

This system has evolved some in the ensuing few years, but the basic concepts and components remain the same as the ones introduced in 2021.

5.2 Clay Static Linking and Deduplication

Once users started to have multiple desks with complex applications in them, the Ford build system started to have quite a bit of duplication, where multiple desks would build the exact same library or filetype definition, leading to several copies of common libraries in memory.

Having many copies of the same library in memory is a well-known problem with static linking, which is logically what Ford does to link Hoon programs together. The linking step is a trivial dynamically typed ("vase-mode") 'cons' operation, since Hoon's variable lookup simply traverses the "subject" (environment) type left-to-right. Importing a library means prepending that library to the environment.

Fortunately, because Urbit supports structural sharing and Ford is a purely functional build system, a referentially transparent cache was added to Ford. This cache does not know which desk a build is being run in, so it can cache and deduplicate builds across desks seamlessly without losing the logical static linking (see `~rovnys-ricfer` and `~wicdev-wisryt` (2024), pp. 75–82 in this issue).

Urbit thus gets the best of both worlds: the sanity of static linking and the memory deduplication of dynamic linking.

5.3 Clay Tombstoning

Besides noun deduplication, Clay has also introduced a tombstoning system. Files in all revisions before the current one may be "tombstoned", meaning that the old data has been deleted and just the reference remains. Tombstoning policy can be configured for files, directories, etc, so that large files aren't duplicated while full revision history can be retained for other things where it's appropriate.

Tombstoning is not at odds with the referential transparency of scries, since a scry attempt may return `[~ ~]`, a permanent scry failure (that resource will never be available).

5.4 Userspace Permissions

As users install more applications, it becomes increasingly important for the system to be able to protect itself and other apps from a malicious app. A basic measure was introduced in 2023 so that a Gall agent can tell which other local agent sent it a command. This "poke provenance" was enough to prevent certain kinds of confused deputy attacks involving ships delegating some functionality to their moons.

For full protection, however, a system of "userspace permissions" is required, where the user must approve an agent's access to kernel features and other applications with some granularity. Such a system was prototyped in late 2022 and is scheduled for further development and deployment later in 2024.

6 Conclusion

Urbit development has begun to realize many of the promises of the 2016 whitepaper. The system is more reliable, more scalable, more extensible, and more secure than it was then. The network has grown to include thousands of users, and the system is now capable of supporting a much larger userbase. Indeed, this article omits many quality-of-life improvements, such as scrying over HTTP, EAuth, `%bout` timing hints, bootstrap pill management, reliability and usability of Tlon's social media application, color printing, ordered map data structures, compile-time evaluation in Hoon, the Bridge contract operations tool for Azimuth, and details of the Urbit HD wallet. More detail on past and planned accomplishments are provided at the published roadmap, `https://roadmap.urbit.org/`.

While more work remains to be done, the kernel of Urbit is sound and the contours of the kelvin-cooled system have begun to assume more resolution. ⊠

Appendix

This appendix addresses developments from content in the whitepaper by section. While this is somewhat narratively unsatisfactory, it provides an easy reference. In cases where no significant changes have been made, the section has been omitted.

Accessing Urbit's namespace as a FUSE filesystem still has not been implemented. Some of the whitepaper was written in the "prophetic present tense", and while a lot of those prophecies have come true, that one remains aspirational.

Web 2.0 interoperability (as the whitepaper discusses, Urbit as a "browser for the server side") has been partially superseded by Web3 interoperability. In 2016 Web3 barely existed, but by 2024 it has become a relatively established industry. Over the last few years, acceptance of Urbit's complementary nature to blockchains has broadened.

Web3 is based on the idea of individual sovereignty: open entry and permissionless extensibility. Blockchains strive to

provide these properties for public, global state. For individual sovereignty over personal and private social computing, something like Urbit is needed. Web3 has an Urbit-shaped hole.

Urbit is still "neither secure nor reliable", but it is much more reliable than back then. We estimate that about two more "nines" have been added to its uptime percentage. Error handling has improved dramatically. Userspace apps can be managed by a user. The runtime and kernel both clean up after themselves better in several different ways, so crashes due to resource exhaustion are much less common.

All of the commands that users now lean on for managing resource usage were written after the whitepaper:

- `|trim` cache pruning
- `|pack` defragmentation
- `|meld` deduplication
- `chop` event log truncation
- `roll` event log rollover

Nock's changes going from 5K to 4K resulted from `~fodwyt-ragful`'s bytecode interpreter in 2018. Three changes were made:

1. The Nock 5 equality check opcode was tweaked to enforce taking two subexpressions, rather than optionally taking a single expression that could return a cell (pair), whose head and tail would then be compared for equality. This optionality turned out to be annoying when writing an interpreter, and it provided little utility.

2. Nock version 4 added a new Nock 10 opcode to replace part of a tree with a new value. Creating mutant trees like this is a basic operation that Hoon code performs regularly, using the `%=` rune and its variants (`%*`, `%_`, `=.`, and `=:`). Before the addition of the Nock 10 opcode, the Hoon compiler would use the type of the trees involved in a mutation expression to codegen a tuple of

Nock "auto-cons" expressions (cells of expressions that return cells of their results) that would reconstruct the mutant tree out of the new subtree cons'd together with the remaining pieces of the original tree.

This had a few problems. It allocated more Nock cells at runtime than needed. It prevented the interpreter from optimizing in-place mutation of a noun with a reference count of one. If no one else is looking, creating a mutant copy can be done by overwriting the noun in-place and nobody can tell on you. APL interpreters, which have a similar memory model to Nock nouns, usually include this optimization, and Urbit's current runtime does as well.

3. A tertiary benefit to adding Nock 10 was that the calling convention for "slamming a gate" (providing an argument to an anonymous function and running the function's body) could now be expressed in statically generated Nock code without needing to invoke the Hoon compiler. This allowed a bit of additional expressivity in some Hoon standard library routines that call into untyped Nock.

Parsing and compiling are still slow, unfortunately, but there have been massive performance improvements to Hoon compilation, especially in memory usage. In 2017 compiling kernelspace would use around a gigabyte of RAM; now it uses on the order of a hundred megabytes. Reducing this memory pressure alleviated a lot of crashes from out-of-memory errors (`bail:meme` errors) – it was a major contributor to the increase in overall reliability.

The Udon and Sail domain-specific languages were added to Hoon, for Markdown-like rich text and XML/HTML, respectively. There is a tentative consensus among core developers that these DSLs should be kicked up a layer out of the base Hoon language, but making Hoon syntax recursively extensible likely requires splitting out the definition of the Hoon AST into a recursion scheme, which is nontrivial.

Hoon's recent changes have reduced the prevalence of the

"TMI problem". Since Hoon's type system supports subtyping, it is common for code to fail to compile due to having "too much information" (TMI) – sometimes a more specific type cannot be used in place of a more general one, namely when that type is in "contravariant position". For example, if the type of some field in the Hoon subject was a list but was specialized in a conditional branch to be a non-null list, attempting to "overwrite" (really, create a mutant copy of) that field in the subject with a normal list, the compiler will throw a type error because it cannot guarantee that the normal list is not null. If the compiler did not know the subject's list field was non-null, then overwriting it with some other list would "just work". Almost always, the eventual result of the expression admits a null list, so the type error is an annoyance rather than an assurance.

Hoon type system changes have also included multiple attempts at resolving another difficulty with subtyping: "wetness". A "wet core" is a Hoon core (pair of code and data, where the code is run using the whole core as the subject) with a "sample" (argument slot – the head of the "data" tail by convention in the type system) whose type can vary. This is very closely related to the mathematical idea of parametric polymorphism. Like some other languages (including Haskell), Hoon does not strictly guarantee "parametricity", but wetness is used to implement parametrically polymorphic functions and even type constructors (called "mold builders" in Hoon).

Wetness has an issue where the "faces" (names for particular slots within the tree) in the sample type in the definition of the wet core can differ from the faces in the sample type at any given call site. Deciding which of those faces to use in the body of the code is a subtle, thorny problem. The Hoon at the time of the whitepaper employed one solution; another was written afterward, around 2017; a third was deployed in 2019. This last revision, called "repainting", is generally thought to be the right strategy, but its implementation remains unfinished in 2024 – it has an issue where the precedence of call-site vs. definition-site face names is not preserved under recursion. This issue will need to be fixed before we can say the right answer to this issue has been achieved.

Wetness in general is an interesting and novel approach to

both polymorphic and recursive types, both of which are directly represented as lazily evaluated types (Urbit core developers often call the "evaluation" step "type inference" informally, but it corresponds to the "type synthesis" direction of a bidirectional typechecker in type theory terminology). This has pros and cons.

On one hand, this representation allows for more expressivity than many comparable type systems at around the same spot on Barendregt's lambda cube (λω: types and values can both depend on types). For example, the `$_` rune allows the programmer to declare the type of one expression in terms of the return type of any other expression, without that expression ever needing to be run, only "type-inferred" (synthesized). This is very convenient for programs that return modified versions of themselves, which is ubiquitous in Urbit.

On the other hand, wetness can be difficult to reason about – it is the hardest part of Hoon to understand fully. In addition, it likely slows down the Hoon compiler considerably, since every call-site needs to run a separate full type synthesis step, and in chains of wet calls, that synthesis becomes recursive. It is not difficult to imagine running up against exponential asymptotics in a system like this.

There is some interest among core developers in experimenting with a more constrained type scheme that could be compiled to a fully evaluated data structure with de Bruijn indices. Such a quantified type scheme with binders could make Hoon's type system easier to learn and reason about.

The formal Arvo boot sequence as described in the whitepaper was not implemented at the time it was written. It was deployed in 2019. Around the same time a lot of other runtime changes were made. The runtime was split into an I/O process and an Arvo worker process. The event log was switched from a custom binary file to the LMDB database.

Meanwhile, in userspace, the `%spider` agent was added, to run a new kind of userspace program: a `%thread`. This is not a separate processor thread of execution, but rather a system of monadic I/O based heavily on Haskell's I/O monad. Unlike agents, which are designed to allow a programmer to implement multiple long-lived asynchronous operations concur-

rently using non-blocking I/O, a thread is designed for building a linear sequence of asynchronous operations, more akin to a coroutine.

Arvo's "duct" as a reified vane call stack is still very much in use. The patterns of use have evolved somewhat. It is now much more common for vanes to use "synthetic ducts" as the origins of certain operations to disconnect them from their cause.

For example, Ames uses a synthetic duct for each "flow" (Ames's version of a socket connection) to set packet re-send timers for that flow and pass requests to other vanes. This disconnects the flow's duct from the duct that the runtime used to inject the event into Arvo, preventing Unix process restarts from causing desynchronization. As another example, Gall now uses a per-agent synthetic duct for the moves coming out of that agent.[2]

The `.^` dotket rune no longer blocks until complete. Because the agent model expects non-blocking I/O, a single dotket could make an entire agent stuck if it can't be completed. While a lot of work has gone into fleshing out the scry namespace, core developers are now entertaining the notion that the expectation of synchronous access to the namespace might not be worth it for agents. An Urbit Improvement Proposal has been submitted that would remove the `.^` dotket rune in favor of a move-based interface to the namespace (see `~rovnys-ricfer`, `~fodwyt-ragful`, and `~mastyr-bottec` (2024), pp. 47–58 in this issue).

This shift reflects a subtler attitude shift toward being more wary about asynchronicity. Earlier Urbit designs were bolder about introducing asynchronous operations; more modern ones treat asynchronous operations as complex and bug-prone, to be avoided if possible. This shift has come from experience – it is surprising to find race conditions in a single-threaded, purely functional operating system, but we have found and fixed plenty over the years.

[2]Note that this approach currently has a known flaw, wherein if an agent sets a timer in response to hearing a request from another ship, it needs to remember that ship in order to obtain the same duct to cancel the timer, cf. issue #6884.

The whitepaper states that "most data is now in Clay". This is no longer true. Now most data lives in Gall as agents and their state. Clay has proven difficult to use for storing application data for a number of reasons, and Gall has become more featureful. There is consensus among the core developers that a better userspace model should be devised to unify Clay and Gall. One such model, called "shrubbery", is now being investigated.

Urbit is still a non-preemptive OS, but it could be made preemptive by letting the kernel emit a Nock hint to the runtime to enforce a time limit on the enclosed computation. If it failed to complete within the given time limit, the runtime would abort the entire Arvo event. This would be considered a nondeterministic error (`bail:fail`), like a user-initiated event interruption (Ctrl-C). This could potentially even be used to turn Urbit into a real-time OS.

3.1 Uniform Persistence

Significant work has gone into improving the runtime's implementation of uniform persistence:

- demand paging to allow storage of more data on disk than fits in RAM;
- a guard page to efficiently prevent stack smashing;
- a snapshot migration framework to facilitate upgrades;
- separating ephemeral and persistent state to avoid extraneous disk writes;
- using the LMDB database for the event log, instead of a custom binary file;
- the Ares project's Persistent Memory Arena (PMA), a general-purpose single-level store that can efficiently manage the persistence and paging for 64 terabytes of data, using a copy-on-write B+ tree;
- the "epoch system", which broke the event log into epochs, allowing event log rollover and truncation;

- persistent Nock caches, allowing arbitrary computations to be memoized in the runtime in a way that persists across Arvo events and Unix process restarts.

3.2 Source-Independent Networking Ames is still sequential and idempotent for commands. Despite having been described as "content-centric" and "source-independent" in the whitepaper, those two properties are only now being realized, in the "directed messaging" project that is currently underway. Directed messaging refactors Ames to be much more like the Named Data Networking project (Zhang et al., 2014): packets and messages are both constrained to be request/response, with routing being achieved using a Pending Interest Table just like in NDN.

Directed messaging supports network reads in the form of namespace reads. A ship can send a network request to read from another ship's namespace. The requesting ship sends request packets containing a scry path, and the publishing ship responds with an authenticated scry binding: a pair of the path and the data bound to that path.

Network writes (commands) are factored in terms of reads: a write request is sent as a read request for an acknowledgment that the command has been performed by the receiving ship. When the receiving ship hears this request, it sends a network read request to download the command datum from the sending ship, who has published it in its scry namespace. Once the command finishes downloading, the receiving ship performs the command and makes the acknowledgment available in its own namespace.

As an optimization, the requesting ship places the first fragment of the command datum in the first request packet it sends. In the common case of a 1 KB command datum, the entire command+ack interaction consists of a single packet roundtrip.

This is only the latest in a series of changes that have been made to Ames since the whitepaper:

- it is now backed by Azimuth, on the blockchain
- it supports personal continuity breaches

- "stateless forwarding" brought orders of magnitude more scalability
- the `%clog` backpressure system was added to prevent unbounded packet queueing on publisher ships; this was later tuned for better performance
- flows can now be closed, so their memory resources can be reclaimed
- "dead flow consolidation": outbound requests to unresponsive ships now share one global timer every two minutes, dramatically reducing the number of Arvo events
- the `%flub` system was added to close a DoS vulnerability: the system can drop incoming requests until it is ready to process them, without any queueing
- the `|snub` and `|ruin` commands were added to allow manual blacklisting of peers
- network reads have been implemented
  - the Fine remote scry protocol
  - the `%chum` one-to-one encrypted remote scry protocol
  - the one-to-many encrypted remote scry protocol

3.4 Interruption, etc. There are still no worker threads, but a potential strategy has been devised for them, and the Ares PMA should support parallel execution. There are still no event timeouts, but we do not expect them to be difficult to implement.

3.5 Solid-State Interpreter While there is not yet a unikernel, a specification has been written for one. There is no Raft cluster but the `%phoenix` app now uses encrypted remote scry to back up application state. There have also been proposals to use FoundationDB to replicate event log storage.

4.2 Larval State This section is still applicable. Arvo still has a larval stage, and it does more or less the same thing.

The "bring your own boot system" work, which has been built but not deployed, will allow a user to assemble a boot sequence out of kernel source and a list of userspace desks (app source packages). This could also be used to facilitate "quick-boot". Nock caches could be injected into the runtime along with kernel or app source. These caches could contain all invocations of the Hoon parser and compiler, reducing boot time to well under a second on common machines.

5.1 Nouns The Murmur3 system was aspirational but has been subsequently realized. A few years ago, a horrendous bug was found and fixed in the "mug" short hash for cells: the algorithm was symmetric, so `[1 0]` hashed to the same value as `[0 1]`. This led to far more hash collisions than there should have been, degrading the asymptotics of a number of data structures.

6.3 Nock Optimization The "jet dashboard" (the module in the runtime responsible for registering, matching, and running jets) has been rewritten multiple times. `~fodwyt-ragful` has proposed a jet dashboard that would match Nock formulas on the basis of noun equality rather than matching Nock cores (a code/data pair used as the unit of code organization) while ignoring mutable slots, the way the previous ones have worked. This would have the advantage of simplicity, but it could be more difficult to implement more complex pieces of code, such as closures.

The Ares project has another new approach to jet matching, which is to resolve the matching statically during a transpilation step from Nock to a more optimized representation. This is facilitated by a novel static analysis technique for Nock, called Subject Knowledge Analysis (SKA), the brainchild of `~ritpub-sipsyl`. SKA analyzes the data dependencies within a Nock formula, allowing inference about the tree shape of the "subject" (environment, or scope) at different points of the computation. SKA should allow for more efficient registerization of intermediate values, and will also provide enough information

to statically compute whether a jet will match. The goal is for Ares to convert a hot loop in Nock into similar machine code that would be emitted by a C compiler, including direct jumps back to the start of the loop.

The Rust interpreter project mentioned in the whitepaper is dead, but in the meantime several other interpreters have been written:

1. Jaque, written using the Graal/Truffle JIT framework;
2. NockJS, an interpreter in JavaScript that uses trampolining;
3. `cl-urbit`, a Common Lisp interpreter used as a jet dashboard testbed; and
4. Ares, written in Rust and C, intended as the next mainline interpreter version.

In 2019, a custom Nock bytecode was written and deployed. This was so much faster than the old tree-walking interpreter that we were able to delete tens of thousands of lines of C code that jetted the Hoon compiler. Until then, the Hoon compiler was almost entirely jetted, since the Nock interpretation overhead was so bad. Since then, the allocator and Nock function calls have been slow, but the speed of Nock noun manipulation has been comparable to manual manipulation in C.

An interesting optimization that was added to Vere more recently is called "tail-call modulo cons" (Leijen and Lorenzen, 2023). This approach allows recursive loops that are almost tail-recursive to be optimized as if they are tail-recursive, as long as the only operation that happens after each recursive call is a "cons" that prepends or appends another value to the result of the recursion.

Another optimization, introduced with the bytecode interpreter and refined recently, was to optimize the Nock 10 opcode that overwrites a slot in a tree. In the case that there is only one reference to the tree being modified, the mutation can be performed in-place, by overwriting memory, rather than by allocating a new mutant tree and then decrementing the refcount on the old one.

6.5 Transition Function The whitepaper mentions off-handedly that Nock could not be replaced. While this author does not expect Nock to be changed, much less replaced, this could be done if needed. Such implementation is left as an exercise for the reader.

7.1 Hoon Semantics Hoon has grown, but the semantics have largely remained intact. `$span` and `$twig` have been renamed, but this may be reverted at some point. (Most core developers prefer the older names.) Core types have "chapters" added to them, mostly for documentation purposes.

Head recursion still requires a cast in Hoon. The compiler can go into an infinite loop if given malformed input. Interestingly, of all the challenges experienced by Hoon developers, this has turned out not to be a severe issue in practice. (Future language designers take note: needing to hit Ctrl-C if your compilation takes a lot longer than normal is not a show stopper.)

Hoon keywords, employed in the whitepaper appendices, was removed due to disuse and disinterest. Keyword Hoon may be re-introduced for pedagogical purposes, but for all essential purposes, runes are the most expressive and intuitive way to think about Hoon as a mature developer.

7.2 Type Definition All of the whitepaper's high-level claims about Hoon are still true, but a few changes to mold (type) declaration and the scope of the language's keyword equivalent (runes) has been expanded.[3] There have been two or three revisions of syntax and type system. The latest change added "doccords", a form of docstring, to the syntax.

The syntax for declaring molds has changed, and as of the last major revision to the mold system in 2019, it is now possible to add a predicate gate to a mold definition that will be run whenever the derived mold function ("gate") is run to coerce a raw noun into a noun of that type. This is useful for ingesting data that has constraints other than tree shape, such as the

[3] We consider the growth of Hoon to be somewhat unfortunate, and envision that it eventually slims back down somewhat as it is refined.

ordering of keys in a map.

A number of runes have been added to Hoon:

1. `|$` barbuc (produce mold builder wet gate)
2. `|@` barpat (produce wet core)
3. `$|` bucbar (produce structure to satisfy validator)
4. `$<` bucgal (filter a mold, exclude)
5. `$>` bucgar (filter a mold, match)
6. `$:` bucmic (enter manual value mode)
7. `$~` bucsig (define default custom type value)
8. `#/` haxfas (produce typed path)
9. `+$` lusbuc (type constructor arm)
10. `+*` lustar (door alias)
11. `;<` micgal (monadic do notation)
12. `^~` tissig (compose many expressions)
13. `?#` wuthax (match type, experimental)
14. `!<` zapgal (extract vase to mold)

The canonical pronunciations for syllables was changed in 2019, then reverted due to effectively unanimous demand. The one modification that persisted was the change of `;` from 'sem' to 'mic', since 'sem' sounded too close to 'cen' (`%`).

The Hoon type system used to use an idempotence check when coercing a raw noun into a typed noun. As of the 2019 type system revision, that check has been removed, and the mold gate will crash (`!!` zapzap) if the noun fails to validate.

8.1 Arvo Kernel Interface Arvo's `+peek` arm is employed much more than it used to be, since we make heavier use of the namespace, and because as Urbit matures, the runtime needs to do more inspection of the Arvo state.

The "wires" (request identifier paths) that Vere uses were cleaned up to share the same conventions and match the name of the target vane.

8.2 Arvo Kernel Modules Vane routing has long used only the first letter of the vane; this is now specialized to the standard vanes, and new vanes can be added that have an arbitrary number of letters, although four-letter vane names remain standard. This allows for an arbitrary number of vanes, not 26 maximum.

The vanes have changed quite a bit since the whitepaper was written. No vane has remained untouched.

1. Ames (p2p networking) was rewritten twice.

2. Behn (timers) was added, rewritten, and then heavily modified.

3. Clay (revision control) has undergone several major transformations (discussed elsewhere throughout this article).

4. Dill (terminal interaction) has been rewritten.

5. Eyre (HTTP) was rewritten and its HTTP client functionality was split into a new vane, Iris, leaving Eyre as just the server side.

6. Ford (build system for Hoon) was written, rewritten, and finally moved into Clay.

7. Gall (application runner) has been rewritten and has had many major changes made to it.

8. Jael (PKI) was added for secret management, rewritten to track the PKI, then modified to work with the PKI rollup.

9. Khan (control plane) was added to allow scripting and lifecycle management from Unix.

10. Lick (Unix inter-process communication) was added recently, so Urbit can communicate using nouns with other local Unix processes.

8.3 Structured Events Depth-first move ordering is being revised, as described in more detail in an earlier section.

The whitepaper mentions it is "not conventional" in Urbit to use promises. `%spider` threads are now a major exception to this. Threads are all killed on upgrade to prevent leaking the old kernel inside the closures.

8.4 Security Mask "Arvo is single-homed" per the whitepaper. This property was subsequently changed to multi-homed to support onboarding as a comet and upgrading to a planet, then removed again when Azimuth was put on-chain.

The security mask is just now starting to be enforced by the vanes. This was necessitated by the `%chum` one-to-one encrypted remote scry protocol. Agent provenance was added to pokes, so an agent now knows which local agent poked it.

This represents a shift in thinking among core developers. Rather than a more monolithic, static, trusted userspace in earlier designs, userspace is now thought of as more multicentric, dynamic, and untrusted. If apps can all be trusted, there is no need to protect one app against the others. In practice, however, there is a compelling need to prevent a single malicious (or, equivalently, badly written) app from breaking the security properties of any other app. Allowing an app to see which other local app is poking it provides enough information to resolve a major class of confused deputy problems without implementing a full-fledged capabilities system.

The security mask model is not expressive enough to encode the idea of a private group with a dynamically calculated set of participants. This was needed for the one-to-many encrypted remote scry protocol, since otherwise an app would need to fully materialize the set of allowed peers whenever it published a piece of data. The computational cost of this would

have superlinear asymptotics, which was deemed unacceptable at the base layer.

Instead, the encrypted remote system has its own separate concept of a "security domain" as a path prefix, and the kernel performs a dynamic query when a peer asks for access to private data with that prefix. While the performance is not a direct comparison with the security mask solution, the dynamic query can typically be implemented in logarithmic time by the application, e. g. a binary search through a Hoon `$map` data structure (a treap). For more complex permissions models, this approach should scale much better.

9.2 Cryptosuite The cryptosuite is still not upgradeable through the initially intended mechanism, where an interface core would be sent to Ames and stored there until a kernel update, when it would need to be deleted and re-sent to avoid leaking the old kernel that would be in the closure of the crypto interface core. The "new" `crub` cryptosuite core has been in use for several years now, though, with standard cryptographic algorithms jetted by reference implementations. The Ames cryptosuite currently in use was audited by a security firm in 2020 (`~rovyns-ricfer` and `~poldec-tonteg`, 2020).

In practice, the Ames protocol has changed a few times since the whitepaper, but the cryptosuite has not. Allowing the cryptosuite to continue to upgrade after kelvin zero will probably still need to be implemented, but we have not yet done so.

For simplicity, when going on chain, the Ames protocol key handshake was changed to a stateless Diffie-Hellman key exchange between the public keys listed on chain. This has the advantage that two ships can start sending encrypted, authenticated packets to each other without any handshake at all. The downside is that forward secrecy is limited to on-chain key updates, which have a cost, so in practice there is very little forward secrecy. This will likely be revised in a later protocol version to support better forward secrecy.

With the Directed Messaging rewrite, a new packet authentication protocol has been added, called LockStep. This

protocol is based on the BAO streaming authentication protocol (O'Connor, 2024). Like BAO, it uses the Blake3 tree hash to establish fingerprints for both individual packets and the full message datum. It differs from BAO in two main ways: it is packet-oriented rather than byte-oriented, and it guarantees that as long as packets are received in order, each packet can be authenticated immediately without waiting for any other packets.

The encrypted remote scry protocol also uses the `+crub` cryptosuite with deterministic encryption, but it does have forward secrecy, tunable by each application.

9.3 PKI Azimuth, the set of Ethereum contracts, replaced the whitepaper's original PKI. About two years later, the "Layer 2" naive rollup was added to reduce the cost of spawning planets.

"Wills" and "deeds", as described in this section, are no longer part of the system. 8-bit galaxies, 16-bit stars, and 32-bit planets all have their public keys listed on Ethereum. A 64-bit moon is assigned its public key by the planet that spawned and owns it, and other ships find the moon's public key by asking the planet. A 128-bit comet signs its address with its own key, and other ships validate the comet's key by checking that it hashes to the comet's address. This prevents comets from breaching, but it allows them to self-authenticate without being part of the on-chain PKI.

9.4 Update, Routing, and Parental Trust Urbit's federated routing system can still be analogized to a combination of STUN and TURN approaches to NAT traversal.

The STUN protocol is now used explicitly: a sponsee ship sends a STUN request to its sponsor every twenty-five seconds. It only sends an Ames command to its sponsor to tell the sponsor it has moved to a new transport address (IPv4 address and port) when the STUN response changes, indicating that from the perspective of the sponsor, the sponsee is at a new transport address. A caveat "from the perspective of the sponsor" is due to the policies of some routers to expose a different source address for each outgoing destination ("source NAT") –

if the sponsee is behind a source NAT, each ship on the network will see a different source address on packets coming from the sponsee.

With directed messaging, routing is now "directed": if ship A sends a request to ship C through C's sponsor B, then the request takes the path $A \rightarrow B \rightarrow C$ through the network, and the response takes the reversed path $C \rightarrow B \rightarrow A$. This is in contrast to all previous versions of Ames, in which the response path will ignore any relays in the request's path and instead get relayed through A's sponsor, in so-called "criss-cross routing".

This directedness promises to reduce routing complexity and handle more NAT configurations, including source NAT. It has already proven to reduce complexity in several other layers of the networking stack.

As for parental trust, the trend has been toward minimizing the amount a parent needs to be trusted. Code signing could be done by a consortium of galaxies or other nodes designated with such a task, such as a set of core developer ships. Urbit's hierarchical deterministic wallet system could require a signature by an upstream key – associated with an Urbit ID but kept "cold" by not being used for any other purpose – to authenticate sensitive updates, such as kernel updates. That way, even if a vulnerability is found that compromises a planet's sponsoring galaxy or star, the planet would validate the signatures by the upstream keys, so that attack on its own would not be enough to load malicious source code onto the planet.

There is support at both the CLI and in Urbit's standard user interface, Tlon's "landscape" app, for turning kernel updates on and off and switching the download source of kernel source code to any ship the user likes.

Userspace apps can be downloaded from any ship on the network, not just the ship's sponsor. This is another step away from obtaining all software from the parent. Userspace apps could use the same kind of code signing mechanism described above – which to be clear, is still vaporware as of the time of this writing.

9.5 Deed Distribution This section is now obsolete except for comets, which as described earlier, are self-signed. It is worth noting that the double-spend problem inherent in the whitepaper's PKI design has now been solved, through using a blockchain.

9.6 Permanent Networking Urbit's networking is still permanent, but edging toward more flexibility and recoverability.

Ames flows (Ames's version of socket connections) can now be killed, allowing both participating ships to reclaim memory resources used for storing the sequence numbers and other metadata about that flow. This is a good example of Urbit doing more to clean up after itself than it did when the whitepaper was written.

The whitepaper reads "brain damage is fatal by default", and "event backup makes no sense", but these claims are no longer considered true. Various forms of recoverability have been added. Personal breaches allow a ship to reset its communications with other ships after losing information. The `%phoenix` app lets app state be backed up and restored – if the restoration occurs after a breach, it is heuristic and could be lossy, but apps can be written defensively to maintain as much state as possible across a breach. Future work will likely push harder on recoverability.

Moving more network activity from writes to reads, through the scry namespace, has been shifting our conception of exactly-once delivery. It now seems likely that by a year or two from now, Urbit's traditional notion of exactly-once delivery will be opt-in, built on top of the more general substrate of publishing scry bindings.

This requires another vaporware warning, but one could imagine a userspace application sending three poke commands to `~zod` by binding the commands to three paths like `/poke/~zod/foo/1`, `/poke/~zod/foo/2`, and `/poke/~zod/foo/3`, into a subtree of its own scry namespace. The kernel would ensure `~zod` is notified of those bindings. `~zod` would download the values at those paths and notify its `%foo` application. The app would interpret the values as a se-

quence of poke commands, to be applied once and in order. Once it processes each poke, it would bind an acknowledgment value to a predetermined path in its own namespace, such as `/ack/~nec/1`.

9.7 Permanent Applications Applications are generally no longer considered permanent. The current Gall model lets a user suspend an application and revive it later. Unfortunately the current model is known to be unsound: "signs", i. e. responses to outgoing commands or queries, to a Gall agent are dropped while the agent is suspended, leading to an inconsistent state.

Future designs for Urbit userspace will likely include a suspension notification for agents. The kernel also needs to intercept and store all pending I/O in and out of agents, so it can cancel the I/O and let the agent know it was canceled.

This goal is at least somewhat at odds with exactly-once network messaging, since in order for the invariants of exactly-once messaging to hold, the sending agent needs to be delivered the acknowledgment reliably. If the ack is dropped while the agent is suspended, that could lead to an inconsistent state between the requesting and receiving ships.

Something namespace-oriented, like the example with poking `~zod` by binding paths in our local namespace, could be more amenable to maintaining exactly-once delivery across suspension and revival without the kernel needing to maintain an unbounded queue of acks to deliver to the agent. It could also help with recoverability after a breach, since storing userspace data with absolute references (i. e. scry paths) allows for more seamless social recovery – if any peer still has the scry binding, the breached ship could download and reinstate it. The importance of immutable absolute references for data is one of the key insights leading to the "shrubbery" noun-maximalist way of thinking.

One way in which applications are more persistent now than at the time of the whitepaper is that upgrades are now quite real. The kernel and userspace apps can both upgrade "over the air" (OTA) by receiving new source code over the Ur-

bit network. The kernel's upgrade system was too buggy for production use until 2020, when the Ford Fusion project moved Ford into Clay – ridding Ford of all its buggy asynchronicities – and enforced a bottom-up upgrade order: Hoon, Arvo, vanes, then userspace.

A year later, near the end of 2021, the ability for Urbit to distribute and install userspace apps was added, enabled by major internal changes to Gall, Clay, and the Arvo kernel to ensure each Clay desk (unit of source distribution) is independently buildable, bootstrapped from its own source code without any external dependencies.

App publishers can push down source code to their users for the next version of their app, intended to be built and run using the next kelvin version of the kernel. When the user's ship receives the kernel's source code for the new kelvin, it upgrades the apps and the kernel atomically with each other, ensuring all apps (soon: only apps marked as "essential") update succesfully – if an app fails to upgrade to the new kelvin version, the Arvo event is aborted, rolling back the kernel upgrade and any other app upgrades that had happened during the event.

Eventually, the kernel should be able to keep agents designed for older kernels running. We call this idea a "kelvin shim". A project along these lines was attempted at the start of 2023, but the current userspace model turns out to be to be too tightly coupled to the kernel for shimming to be feasible. This experience has informed our next generation of ideas for the kernelspace/userspace interface.

9.8 Acknowledgment Acknowledgments have changed in a number of ways. Packet-level acks are being (mostly) removed. Authentication for both packets and messages is being redone in the Directed Messaging project. Negative acknowledgments are no longer a space leak and are cleaned up without loss of correctness after a certain number of later messages. Finally, congestion control is being moved into the runtime.

Some things have stayed the same. A message is still an application-level transaction. Ignoring the computation time

to process a message is still a useful optimization for congestion control. Negative acknowledgments are still considered to be string-like data describing an error, rather than a programmatically manipulable exception data structure.

9.9 Packet Rejection Cryptography failures (failure to decrypt or authenticate a packet) still cause a silent packet drop. To streamline the implementation of this, the `bail:evil` error tag was added to the system so the runtime can know not to inject an error notification event into Arvo in the case of cryptographic failure.

The `%flub` move is a new reason to drop a packet. When Ames passes a message to a local vane, the vane can now respond with a `%flub` move, indicating it is not yet ready to process the message. If a message sent to a Gall agent that is not running, for example, then Gall will respond with a `%flub`. The sending ship will keep retrying, slowly, on a timer, until the message goes through, but the receiving ship does not have to update any state in response – previous versions of the system required the receiving ship to maintain an unbounded queue of incoming pokes to agents that were not running, which was a denial-of-service vector.

10 Ames Part 2: Bitstream The "jam" serialization format for nouns remains unchanged. Its implementation in Vere is now much more flexible and optimized. There are now also implementations in Rust, JavaScript, Haskell, and Common Lisp.

There are more use cases for jammed nouns now. Portable Arvo snapshots are jammed nouns. Interprocess communication between the two Vere processes uses jammed nouns for its messages. The Khan and Lick vanes also use jammed nouns for IPC, and Eyre, the HTTP vane, now supports HTTP subscriptions using jammed nouns in addition to JSON.

10.4 Packet Structure Modern Ames packets are still jammed nouns with a 32-bit header but other than that almost everything is different. For one thing, their internal fields are now

byte-aligned, making them more human-readable in Wireshark or other Terran packet inspection programs.

A bigger shift is that as of Directed Messaging, packets are actually source-independent for the first time.

10.5 Message Decoding This has been totally rewritten multiple times. There is no more unencrypted first packet. Routing will now be directed.

10.7 Poke Processing Pokes are still typed and transactional. Mark lookup is now performed on a per-desk basis, rather than globally, since each app can define its own marks.

Deferred acknowledgments remain vaporware.

11 System Status and Roadmap The whitepaper's 30,000 source lines of code has expanded to roughly 40,000. This number should go down a bit over time. Other aspects of reliability and performance have been extensively discussed above.

References

Barendregt, Henk (1991). "Introduction to generalized type systems." In: *Journal of Functional Programming* 1.2, pp. 125–154. DOI: 10.1017/s0956796800020025.

Leijen, Daan and Anton Lorenzen (2023). "Tail Recursion Modulo Context: An Equational Approach." In: *Proceedings of the ACM on Programming Languages* 7.POPL 40, pp. 1–30. DOI: 10.1145/3571233.

~midden-fabler, Scott Wilson (2024) "%phoenix". URL: https://github.com/urbit/phoenix (visited on ~2024.4.24).

NativePlanet (2024) "GroundSeg". URL: https://github.com/Native-Planet/GroundSeg (visited on ~2024.4.8).

O'Connor, Jack (2024) "Bao". URL: https://github.com/oconnor663/bao (visited on ~2024.4.8).

~ravmel-ropdyl, Galen Wolfe-Pauly (2019) "Azimuth is On Chain". URL: https://urbit.org/blog/azimuth-is-on-chain (visited on ~2024.4.24).

~rovnys-ricfer, Ted Blackman, Paul Driver ~fodwyt-ragful, and Matthew Levan ~mastyr-bottec (2024). "The Stakes of Scrying: .^ Dotket and the Seer Proposal." In: *Urbit Systems Technical Journal* 1.1, pp. 47–58.

~rovnys-ricfer, Ted Blackman and Philip C. Monk ~wicdev-wisryt (2024). "A Solution to Static vs. Dynamic Linking." In: *Urbit Systems Technical Journal* 1.1, pp. 75–82.

~rovyns-ricfer, Ted Blackman and Anthony Arroyo ~poldec-tonteg (2020) "Ames Security Audit and the Future of the Protocol". URL: https://urbit.org/blog/security-audit (visited on ~2024.4.8).

~sorreg-namtyv, Curtis Yarvin (2006) "U, a small model". URL: http://lambda-the-ultimate.org/node/1269 (visited on ~2024.2.20).

— (2010) "Urbit: functional programming from scratch". URL: http://moronlab.blogspot.com/2010/01/urbit-functional-programming-from.html (visited on ~2024.1.25).

~sorreg-namtyv, Curtis Yarvin et al. (2016). *Urbit: A Solid-State Interpreter*. Whitepaper. Tlon Corporation. URL: https://media.urbit.org/whitepaper.pdf (visited on ~2024.1.25).

Zhang, Lixia et al. (2014). "Named Data Networking." In: *ACM SIGCOMM Computer Communication Review* 44.3, pp. 66–73. DOI: 10.1145/2656877.2656887.

The Stakes of Scrying: . ^ dotket and the Seer Proposal

Ted Blackman ~rovnys-ricfer
Paul Driver ~fodwyt-ragful
Matthew Levan ~mastyr-bottec
Urbit Foundation

Abstract

The expedient of Nock 12 and the . ^ dotket rune break the referential transparency of Nock, with consequences for the purity of Urbit's functional programming model. We consider the implications of this and propose a move-based replacement for the . ^ dotket rune that restores referential transparency.

Contents

Urbit Systems Technical Journal I:1 (2024): 47–58.

Address author correspondence to ~rovnys-ricfer.

1 Introduction

While Nock formally has twelve opcodes, numbered 0–11, it is convenient to expose in userspace a fake Nock 12 opcode to expose the scry namespace to userspace developers without an asynchronous retrievel step. This opcode is trivially invoked by using the Hoon `.^` dotket rune.[1] However, Nock 12 breaks the referential transparency of Nock, and this has consequences for the purity of Urbit's functional programming model.

2 How `.^` dotket Works

The Nock 12 opcode and the `.^` dotket rune are used by userspace code to ask the kernel for data in the scry namespace.[2] When Urbit runs userspace code, that code is not raw Nock. It is instead called Mock, which is Nock with two additions: deterministic errors are caught and returned as values to the kernel; and there is an extra Nock opcode, opcode 12. A simple Nock 12 formula looks like `[12 [1 /foo/bar/baz]]`, where `/foo/bar/baz` is a path literal `[%foo %bar %baz ~]`, representing a path in the scry namespace. The result of running this opcode will be the result of performing the scry request, i. e. dereferencing the path.

Catching deterministic errors is uncontroversial, but the Nock 12 opcode poses some difficulties and has been under scrutiny for years now. The most tangible problem with Nock 12 is that it prevents userspace code from persistently memoizing Nock computations; at the moment, only kernelspace code can hold onto Nock memoization caches across multiple Arvo events. The lifetime of a userspace memoization cache is limited to a single metacircular evaluation context, i. e. a sin-

[1] The idea of using a pattern like scrying to introduce a value into a referentially transparent namespace predates Urbit itself; cf. `~sorreg-namtyv` (2006).

[2] The scry namespace, in brief, is a collection of immutable values "bound" by convention at fixed endpoints. The scry namespace is defined by a unique path for each resource. A representative scry path would have the entries `/host/rift/life/vane/request-type/revision-number/desk /file-path/mark`.

gle call to `++mink` or an equivalent gate that runs Mock (virtual Nock with the extra opcode).

Userspace code cannot memoize Nock persistently because it is not feasible for the runtime to prevent a jet mismatch in the metacircular evaluator gate. A gate like `++mink` takes in a Mock expression (a cell of `[subject formula]`) and a "scry handler gate" (which is described shortly), and returns a value which is one of:

- `[0 computation-result]` for a successful evaluation;
- `[1 block]` in the case of a failed scry request; or
- `[2 error]` in the case of a deterministic error.

While error handling is not relevant to the current question, a brief description is included here for thoroughness. The `error` in `++mink`'s result `[2 error]` is a deterministic error, meaning that every attempt to evaluate that `[subject formula]` pair will always result in that same error value—deterministic errors correspond to the lines in the Nock spec where `*a` evaluates to `*a`, i. e. a trivial infinite loop. Nondeterministic errors, such as running out of usable memory or getting interrupted by the user, result in the function failing to complete, and the Arvo event will be aborted and rolled back to its state before trying to run this new event.

As mentioned, the `[subject formula]` pair is not the only input to `++mink`. There is also an argument of the form `scry=$-(^ (unit (unit)))`, called the scry handler gate. When `++mink` encounters a Nock 12 opcode while evaluating an expression `[subject formula]`, it evaluates the sub-formula to obtain a path to look up in the scry namespace. Once it has this path, it calls the scry handler gate on the path, which returns one of:

- `[~ ~ value]` positive scry response;
- `[~ ~]` permanent scry failure; or
- ~ "block".

The positive response case means that the scry was performed successfully and yielded a value, in which case `++mink` injects the value as the result of the Nock `12` opcode and continues to evaluate the Nock. The permanent failure means the scry namespace knows there will never be a value at that path, so `++mink` can immediately return a deterministic error.

A scry block means the system refused to answer the question, i. e. the request was "blocked" for some reason—this could be due to lack of permission, or because the request asked about the future, or the request asked for data from another ship that isn't synchronously available within this Arvo event. If the scry blocks, `++mink` returns a `[1 block]`, where `block` is the scry request path whose value could not be resolved.

The Nock `12` opcode is problematic because it renders the result of the computation dependent on external data which are not in the `[subject formula]` that is being evaluated. The result is no longer a pure function of the `subject` and `formula`: depending on what the scry handler gate is, the same `[subject formula]` could yield different results. Consider `[subject formula]` of `[0 [12 1 /foo/bar]]`. This returns the result of looking up the `/foo/bar` scry request path from the scry gate. Userspace code (which is untrusted) could call `(mink [[0 12 1 /foo/bar]] |=(^ [~ ~ 33]))`, acquire a result `33`, then call `(mink [[0 12 1 /foo/bar]] |=(^ [~ ~ 44]))` to yield `44`. The `subject`s and `formula`s were identical, but because they were evaluated with different scry gates, they yielded different results.

Given that Urbit is supposed to be a pure-functional system, how can this be acceptable? The answer is that Mock, virtual Nock, *is* purely functional (mathematically, a "partial function" like Nock itself) as long as it's only ever run using the real scry namespace, which is referentially transparent. That is to say, all data in the scry namespace can be thought of as something like axiomatic data that everyone agrees on and never changes: all virtual Nock computations will get the same answers to all Nock `12` opcodes that complete, because everyone agrees on which value is bound to each path in the namespace.

Another way to think of the scry namespace is as comprising an extra implicit argument passed to every Mock computa-

tion along with the subject and formula. This extra argument is discovered dynamically over time as more ships bind more values. Since any path can only be bound once, it becomes immutable; all Mock expressions that dereference some scry path will either fail to complete or complete with identical results, hence pure functional.

Note that there's no requirement that all the scry gates be noun-equivalent (intensionally equal), or even contain the same set of namespace paths and values (extensionally equal). The requirement is that no two scry gates ever return different values for any given scry request path.[3]

The Arvo kernel's scry handler gate, as described in the original Moron Lab blog post about Urbit (`~sorreg-namtyv`, 2010), is a "static functional namespace"; even though that gate changes frequently, it maintains the invariant that every scry path's binding to a value is immutable.[4]

As long as a correct Arvo kernel is supplying the scry handler, Mock retains its purely functional nature. But consider again the case where userspace code runs the same `[subject formula]` twice, but with two different scry gates that give different answers to some scry path. From the perspective of a simple Nock interpreter, it's still deterministic: `++mink` is just some Nock function that runs some other computation (Mock) on some inputs. Because it's just some Nock function, it can't break determinism of the Arvo event; replaying the Arvo event will still yield the same result. The problem only arises when trying to get clever about how the runtime evaluates Nock, in particular by caching the results of Nock computations.

Development versions of Vere support persistent Nock memoization. This means a programmer can emit a Nock hint to the interpreter asking the interpreter to cache the result of the computation, so that if it gets run again, the result will be retrieved from the runtime's cache instead of rerun step by step. The cache key is a `[subject formula]` pair. When the interpreter sees a `%memo` hint, it looks at the `[subject formula]` that the hint surrounds. If the memoization cache

[3] A scry gate that always blocks is trivially legitimate, since it never gives any answers, much less answers that differ from those of another scry gate.

[4] Up to some known concerns that are addressed by Arvo Ticks UIP-0116.

has a value for that key, the runtime uses that value as the result of the computation. Otherwise the runtime runs the computation, inserts the result into the cache, and returns the result.

For the moment, only the kernel can place items into the cache. Userspace code runs Mock, so if it were allowed to place items into the cache, there would be three options:

1. Let userspace code place arbitrary values into the cache. This poses a severe security problem: event replay would be nondeterministic, influenced by the contents of the memoization cache, allowing malicious userspace agents to perform a kind of denial of service attack by preventing event replay, or possibly even worse, by injecting cache lines that other applications would use, causing those other applications to malfunction in a way the malicious code wants. The worst case would be causing a system app such as `%hood` to fail to enforce a security check, allowing the malicious code to escalate its own privileges and permitting it to ask the system app to do something like reinstall the kernel.

2. Place the scry gate into the cache key for the memoization cache, so a cache key would be `[[subject formula] scry-gate]` instead of just `[subject formula]`. This would close the security hole, but since Arvo's scry gate undergoes small changes on between every agent activation, it would make these cache lines useless outside of the immediate execution context within a single `++mink` call. This is exactly the situation userspace code is already in, and thus not helpful at all.

3. Have the interpreter record the set of scry `[path value]` pairs looked up by the computation. This would allow userspace code to perform scry requests safely, but the cost is a nontrivial amount of machinery in the runtime, along with a computational cost of looking up a cache line that grows linearly with the number of scry paths that were requested. Of the three options listed here, this is the most reasonable, but it is still highly

nontrivial and the requirement to compare all requested paths could be onerous.

Instead of implementing any of these options, Urbit has currently simple disabled the ability for userspace code to modify the Nock memoization cache. This is enforced by the jet for `++mink`; the jet is simply a call back into the normal Nock interpreter, which is really a Mock interpreter, whose state includes a list of nouns representing the stack of scry handler gates for the current execution context. If this list is null ~, then we must be in the kernel and the code is allowed to modify the memoization cache; otherwise, we are in some kind of Mock context, and the code is not allowed to modify the cache.

The concerns with . ^ dotket go deeper than memoization however, despite that being the only known major concrete issue.[5] Conceptually, the fact that a Mock computation has implicit external dependencies makes the pure-functional nature of the system more fragile—the memoization issue is downstream of this fragility.

Morally and aesthetically, the fact that Nock makes no assumptions about any of its contents is part of the magic of Urbit. Nock is a true functional programming language with no implicit external dependencies at all, and no constraints on the nouns it operates on. Anything that pollutes this vision is potentially harmful to the environment Urbit creates for programs and their authors. Mock's implicit dependency on the namespace makes Nock context-dependent instead of context-independent, as it should be. A related issue is that userspace code does not have access to raw Nock, only Mock. The Hoon compiler willingly outputs Nock 12 instructions when handed a . ^ dotket rune, and there is no means for userspace code to signal that it will refrain from using Nock 12.

[5] Some complexity is introduced in the runtime when jetting the metacircular interpreter.

3 Recommendations

The removal of Nock 12 is fraught with difficulty for userspace and kernel development. The core development team has considered two proposals which will allow the kernel to gracefully deprecate Nock 12. The two concrete lines of action are:

1. Add a new metacircular evaluator gate that does not run Nock 12 expressions (the Seer proposal).

2. Add a move-based interface for scrying to the Gall interface.

This approach will allow core developers and userspace application developers to experiment with code that does not use `.^` dotket. If that resulting code and associated design patterns work well, we will deprecate the `.^` dotket rune and the Nock 12 opcode in favor of move-based scrying.

3.1 The Seer Monad

Seer is a monadic scry interface which allows application developers to read from the namespace without using `.^` dotket or its virtual Nock operator 12. Userspace code run inside of this gate can reestablish the guarantee that it is purely functional, allowing it to modify the memoization cache.[6]

Seer is designed as a replacement for `.^` dotket, and thus attempts to be as ergonomic as the `.^` dotket pattern. Consider a Gall agent that scries using the current `.^` dotket pattern:

```
++  on-poke
  |=  [m=mark v=vase]
  ^-  (quip card _this)
  =/  foo-result  .^(@da /cx/(scot %da now)/foo)
  ::  do something with foo-result...
  [~ this]
```

[6]Note that it's still acceptable for that gate to be called from Mock code, or even nested Mock code, that had previously performed scry requests that were not referentially transparent: raw Nock is context-free, so no matter how the `[subject formula]` cell was created, the result of its evaluation is still pure.

A corresponding snippet of a Gall agent that scries using `++seer` in the continuation style would look like this:

```
++  gear          ::  gall seer
  |*  a=mold
  (seer vase a)
++  on-poke
  |=  [m=mark v=vase]
  ^-  (gear (quip card _this))
  ::  (seer vase (quip card _this))
  :+  %scry  /cx/(now)/foo  |=  foo=vase
  =/  foo-result  !<(@ foo)
  ::  do something with foo-result...
  [%done ~ this]
```

`++seer` values can be composed using a monadic bind:

```
++  foo  ^-  (seer vase @)  !!  ::  /foo
++  bar  ^-  (seer vase @)  !!  ::  /bar
++  on-poke
  |=  [m=mark v=vase]
  =*  r  (quip card _this)
  ^-  (gear r)
  =/  bind  (rapt vase r)
  ;<  foo=@  bind  foo
  ;<  bar=@  bind  bar
  [%done ~ this(some-state (add foo bar))]
```

(In this example, use is made of `++rapt`. `++rapt` is a monadic scry binding gate (~mastyr-bottec, 2023c).)

Here, a `++seer` is the type of programs that scry:

```
++  seer
  |*  [r=mold a=mold]
  $~  [%done *a]
  $%  [%done p=a]
      [%scry p=path k=$-(r (seer r a))]
  ==
```

The `++seer` scry system is intentionally incompatible with the current . ˆ dotket scry system. We believe that `++seer` degrades ergonomics only slightly. In the initial introduction, we

shall provide a wrapper library which accepts an agent written in the `++seer` style and produces an agent that scries in the `.^` dotket style.

The current reference implementation of Seer is available at `urbit/urbit` `~mastyr-bottec` (2023a). A more complete example is available at `~mastyr-bottec` (2023b).

3.2 Move-Based Scry Interface

A Gall agent will be able to emit a new kind of move, in addition to all the moves it can emit now, that asks Gall to perform a scry request on the agent's behalf. When Gall processes this move, it will perform the scry request and convert the result of the scry into a response move that will be delivered back into the agent in a later activation of the Gall vane, in accordance with the normal move processing order that Gall and Arvo use. Thus an agent could emit a list of moves such as `[poke-1 scry-A poke-2 scry-B poke-3 ~]` and Gall will handle all of those moves in the same order as if they were all pokes, i. e. the kernel will not introduce any exceptions to its normal move processing order for scry request moves.

This proposal is less developed than the Seer proposal, and thus no code examples can be provided, and no reference implementation exists at the time of writing.

4 Conclusion

There is a general consensus among Urbit core developers that a new move-based scry interface is better suited as the lowest layer of the scry system than the Seer monadic interface. The fact that Seer or `.^` dotket[7] could be implemented on top of moves (but not necessarily the other way around) militates in favor of moves at the lowest layer. Another benefit of this arrangement is that by making all scries asynchronous, the same interface can be used for both local and remote scries (since remote scries are always asynchronous). Such an abstraction

[7] Or at least an opt-in form of Mock, rather than it being the global default as it is now.

over local vs. remote scry calls removes what would otherwise be an unavoidable branch point in userspace I/O code. It also matches the rest of Gall's interprocess communication model, which does not distinguish between local and remote. A strong argument can be made that this is also the most "grug-brained" option, which is not to be undervalued.

There remain concerns about application development patterns if . ^ dotket is removed. Losing synchronous read access to other parts of the system potentially reduces the ergonomic wins of a naturally growing transaction across applications. This can be implemented in a way that is almost synchronous for local scries, and in particular it is still possible to perform multiple scry requests at the same system snapshot, meaning the requesting agent will see a consistent view of the system without any tearing.

Gall will also need to run the `++on-peek` arm of a Gall agent in the Seer monad, i. e. that arm will return either `[%done scry-result]` to indicate it's done or `[%scry scry-request-path callback]` to ask Gall to perform a scry request, in which case Gall performs the scry, injects the result into the callback gate, and repeats, until a `%done` is returned. This should not be a direct problem, but betokens complexity creeping into the Gall interface.

Because of these concerns with the proposed move-based replacement for . ^ dotket, we will first add it as a new feature alongside . ^ dotket before committing to removing . ^ dotket. In this way, if the application patterns end up suffering inordinately, . ^ dotket may be retained. That being said, the current primary plan is to deprecate . ^ dotket once move-based scrying has been verified as providing a sufficiently good application model. This is a conservative, incremental approach that will not break any existing code and lets us try out the new style in production applications before committing to the removal of Nock 12. ⊠

References

~mastyr-bottec, Matthew Levan (2023a) "/lib/seer". URL: https://github.com/urbit/urbit/blob/uip/seer/pkg/arvo/lib/seer.hoon (visited on ~2024.1.11).

— (2023b) "Seer: A Monadic Scry Interface #6842". URL: https://github.com/urbit/urbit/pull/6842 (visited on ~2024.1.11).

— (2023c) "Seer: A Monadic Scry Interface Examples". URL: https://gist.github.com/matthew-levan/8772f204749748e7d72cc5d298eeeb03 (visited on ~2024.1.11).

~sorreg-namtyv, Curtis Yarvin (2006) "U, a small model". URL: http://lambda-the-ultimate.org/node/1269 (visited on ~2024.3.8).

— (2010) "Urbit: Functional Programming from Scratch". URL: https://moronlab.blogspot.com/2010/01/urbit-functional-programming-from.html (visited on ~2024.1.11).

~wicdev-wisryt, Philip C. Monk, Ted Blackman ~rovnys-ricfer, and Scott Wilson ~midden-fabler (2023) "UIP-0116: Arvo Ticks. Pin Arvo's scry handler to a move tick". URL: https://github.com/urbit/UIPs/blob/main/UIPS/UIP-0116.md (visited on ~2024.1.25).

A Philosophy of (Inductive) Testing

Jack Fox ~nomryg-nilref
FoxyLabs

Abstract

Testing, including unit testing, forms a critical step in software development. This article explores the philosophy and practice of testing in the context of Hoon and Urbit development, particularly as motivated by the development of urQL and Obelisk. Inductive testing provides an efficient yet complete basis for verifying code behavior. Principles and best practices are suggested, as well as a practical example of testing in Hoon.

Contents

Urbit Systems Technical Journal I:1 (2024): 59–73.

Address author correspondence to ~nomryg-nilref.

1 Introduction

What should I test? The short answer is, "Everything". If you don't test it, it doesn't work.

However, *everything* raises more questions than it answers. Such questions fall into two categories, what and how.

No one writes non-trivial software without testing, even if you come to believe that "if it builds, it works". The first time you ran your software was a test. Chances are, you ran individual components to see if those worked. Why not take the extra few minutes to record those tests and results? With practice you will get smart about efficiently recording your throwaway tests. And after you read this I hope you will have a new appreciation of why it is worth the extra time to record your tests.

2 Testing Frameworks

Use what is useful and don't get hung-up on a particular framework or methodology. Record tests as you develop. You may think this slows down your development. That's not necessarily a bad thing. You will move forward with confidence that the last thing you worked on really works and you will find yourself thinking through your specification at a deeper level. This is important in a world of mostly incomplete and nebulous specs. What I am driving at is a cousin to test-driven development.[1] Think of it as real-time test development.

Your test suite has two important non-obvious functions, as regression tests (OK, this one should be obvious) and as part of your specification. If an existing test starts failing, carefully consider Chesterton's fence (Wikipedia (2024a)) before taking the easy road and removing the test.

Regression tests are important not just to prevent unexpectedly breaking functionality, but to allow you to refactor with confidence. Great software is the result of constant refactoring.

[1] Beck (2023) summarizes the principles of TDD.

Listing 1: Agent-level test

```
::
::  Build an example bowl manually
++  bowl
  |=  [run=@ud now=@da]
  ^-  bowl:gall
  :*  :: (our src dap sap)
      [~zod ~zod %obelisk path(limo path/test-agent)]
      :: (wex sup sky)
      [~ ~ ~]
      :: (act eny now byk)
      [run @uvJ(shax run) now [~zod %base ud+run]]
  ==
```

3 "Unit" Testing

Descending from the hypothetical to the practical, every practical programming language comes with a unit testing environment. Hoon supplies `/lib/test` on the `%base` desk, paired with the `/ted/test` thread; the `/lib` file should be copied to your development desk.[2] The rest of this article assumes a working knowledge of `/lib/test`.

Don't get hung up on unit testing dogma. A unit is whatever you want it to be. As a subject-oriented programming language, Hoon does not support private properties and functions: every arm and core in your software is accessible for testing. Generally a unit is some self-contained level of functionality, either from the programmer's or the user's point of view.

Listing 1 depicts a more thorough unit test for working with a Gall agent. (Some would prefer to call this an *integration test*.) Given that level of granularity, we can proceed to test the evolution of a database over time (Listing 2). The test is a bit more complex, but the principle holds constant: test a unit of functionality.

Under the hood, the Obelisk engine calls out to the urQL

[2] `~lagrev-nocfep` (2022) discusses this procedure in detail.

Listing 2: Testing database evolution over time

```
++  test-time-insert-gt-schema
  =|  run=@ud
  =^  mov1  agent
    %:  ~(on-poke agent (bowl [run ~2000.1.1]))
        %obelisk-action
        !>([%tape-create-db "CREATE DATABASE db1"])
    ==
  =.  run  +(run)
  =^  mov2  agent
    %:  ~(on-poke agent (bowl [run ~2000.1.2]))
        %obelisk-action
        !>  :+    %tape
                %db1
              "CREATE TABLE db1..my-table (col1 @t)
    ↪ PRIMARY KEY (col1) ".
            "AS OF ~2023.7.9..22.35.35..7e90"
    ==
  =.  run  +(run)
  =^  mov3  agent
    %:  ~(on-poke agent (bowl [run
    ↪ ~2023.7.9..22.35.35..7e90]))
        %obelisk-action
        !>  :+    %tape
                %db1
              "INSERT INTO db1..my-table (col1) VALUES
    ↪  ('cord') ".
            "AS OF ~2023.7.9..22.35.36..7e90"
    ==
  =+  !<(=state on-save:agent)
  ;:  weld
  %+  expect-eq
    !>  :-  %results
          :~  [%result-da 'data time'
    ↪ ~2023.7.9..22.35.36..7e90]
              [%result-ud 'row count' 1]
        ==
    !>  ->+>+>.mov3
  %+  expect-eq
    !>  db-time-insert-tbl
    !>  databases.state
  ==  :: weld
```

parser to create AST commands from the user-created scripts. The `AS OF` clause overrides the time passed in the bowl. Finally we weld together two results and check them against what we expect. The first check, `++expect-eq`, is of the metadata returned by the last command. The second check is the final database state after the evolution.

Unit testing purism insists on specifically tailoring each test to one specific case of potential failure. This approach tends to be pedantic and frequently we pragmatically sneak in multiple independent tests in one test bundle, as in Listing 3. `++test-alter-index-1` is not an atomic unit test but tests several qualities together. Values like whitespace and mixed-case labels are tedious to exhaustively test and well-suited to property based testing software like `%quiz` (see below).

4 Proof by Induction

We were intentionally dismissive of testing methodologies above because they are mere conventions. There is however some actual theory we can leverage in figuring out what and how to test.[3]

Peano arithmetic is a better lay programmer's introduction to proof by induction. There exists (or perhaps *does not exist*) a special concept, zero. Playing fast and loose with classical logic, "ex nihil sequitur quodlibet", from nothing (more commonly a falsehood, or a contradiction) follows everything (i.e. anything).[4] Zero is not only an integer; rather, in terms of Nock-based programming, ~ is not only the beginning of

[3] Wikipedia (2024b) provides a modestly technical exposition of mathematical induction. You can skip it because we will explain it non-rigorously for lay programmers. We also quibble about the inclusion of zero as a natural number. While old school maths started the natural numbers with "1", but computer science has since infected maths. (Have you ever seen zero of anything? No—it is not natural at all.) Zero is a very, very special number. It's an abstraction that does not map to anything in the physical world.

[4] The "principle of explosion" rigorously follows from use of disjunctive syllogisms. Here, we jocosely indicate that by proceeding from zero inductively we can demonstrate desired properties of testing.

Listing 3: Multiple tests bundled together in one testing arm

```
:: common things to test
:: 1) basic command works producing AST object
:: 2) multiple ASTs
:: 3) all keywords are case ambivalent
:: 4) all names follow rules for faces
:: 5) all qualifier combinations work
::
:: alter index
::
:: tests 1, 2, 3, 5
::       extra whitespace characters
::       multiple command script:
::         alter index... db.ns.index db.ns.table
::           columns action %disable
::         alter index db..index db..table one column
::         action %rebuild
++  test-alter-index-1
  =/  expected1
    :*  %alter-index
        :*  %qualified-object
            ship=~ database='db'
            namespace='ns' name='my-index'
        ==
        :*  %qualified-object
            ship=~ database='db'
            namespace='ns' name='table'
        ==
        :~  :*  %ordered-column
                name='col1'
                is-ascending=%.y
            ==
            :*  %ordered-column
                name='col2'
                is-ascending=%.n
            ==
            :*  %ordered-column
                name='col3'
                is-ascending=%.y
        ==  ==
        %disable
        ==
```

Listing 3 continued

```
=/  expected2
  :*  %alter-index
      :*  %qualified-object
          ship=~ database='db'
          namespace='dbo' name='my-index'
      ==
      :*  %qualified-object
          ship=~ database='db'
          namespace='dbo' name='table'
      ==
      :~  :*  %ordered-column
              name='col1'
              is-ascending=%.y
      ==  ==
      %rebuild
      ==
%+  expect-eq
  !>  ~[expected1 expected2]
  !>  %-  parse:parse(default-database 'db1')
    "aLter \0d INdEX\09db.ns.my-index On
  ↪ db.ns.table  ".
    "( col1  asc , col2\0a desc  , col3) \0a
  ↪ dIsable \0a;\0a aLter \0d ".
    "INdEX\09db..my-index On db..table  ( col1  asc
  ↪ ) \0a \0a rEBuild "
```

counting,[5] but the nothing of every inductive type, most importantly trees and lists. In speaking to programmers, Peano tells us there is a universal nothing ~ and there is a function called successor Succ, which produces some next thing from a previous thing (or lack of thing). Applying Succ to nothing, Succ(0), gives us the first thing. Applying Succ to the first thing, Succ(Succ(0)), gives us the second thing, and so on.

Induction appears in unexpected places—think zero-length strings, which do not appear to involve ~ at all. For instance, the Hoon type `unit` is also an inductive type: it is literally either nothing or something. Knowing now what to look for, see where you can find induction in your own code.

What does induction have to do with testing, however? For inductive types—and simple functions over inductive types—the software author only has to prove, or test, two cases: the case for zero and the case for the successor value after zero. For example, Listing 4 depicts tests for a gate over an inductive type (`list`) in `/lib/seq` (`jackfoxy/sequent`, currently distributed via `%yard` (`urbit/yard`)).

"Hey", you say—"that's more than two tests." That's right: two is the bare minimum of required tests, and only applies to the simplest units of inductive testing. In this case the first two tests suffice, but minimal tests frequently make for unhelpful documentation examples. We heartily recommend providing interesting examples in your documentation and including those examples in your test suite. You don't want users to struggle with examples that don't work, or worse, don't even build. More tests don't hurt anything; the computer doesn't get tired.

Another reason for additional tests is taking a page out of "white box" or "gray box" testing. If you know that there is special logic for the first successor case, you need to test the first case independently as well as a subsequent successor. If you are the programmer and the tester you should approach all of your testing from this perspective. You might even see how to make your program simpler.

[5] Yarvin opted to invert `true` (as 0) and `false` (as 1) (`~sorreg-namtyv`, 2006), meaning that Nock loobeans do not align inductively with other inductive types.

Listing 4: Sequent tests

```
::
::  +contains
++  test-contains-00
  %+  expect-eq
    !>  %.n
    !>  (contains `(list)`~ "yep")
++  test-contains-01
  %+  expect-eq
    !>  %.y
    !>  (contains `(list @)`~[1] 1)
++  test-contains-example-00
  %+  expect-eq
    !>  %.y
    !>  (contains `(list tape)`~["nope" "yep"] "yep")
```

4.1 The First Cartesian Explosion

Through the mathematics of currying programmatic functions, we can have input arguments of multiple inductive types.[6] This results in the minimum number of tests being the number of inductive input elements squared, starting with all elements set to ~ and so forth. Listing 5 depicts a series of tests for the `++append` gate subject to this n^2 explosion.

In this situation, the required number of tests may grow exponentially but in most practical cases remains a relatively small finite number.

4.2 The Second Cartesian Explosion

Input argument interactions are not the only source of combinatorial explosion in testing. Imagine that your function (gate) is a black box. You start submitting random input to figure out what the underlying algorithm is. However, it turns out that you discover that there is not one consistent algorithm

[6]While most programming languages handle multiple function inputs via currying, Hoon gates always accept a single noun, which can be a cell. So in this case the currying is not even theoretical.

Listing 5: Multiple single tests

```
::
::  +append
++  test-append-00
  %+  expect-eq
    !>  ~
    !>  (append ~ ~)
++  test-append-01
  %+  expect-eq
    !>  ~[1]
    !>  (append ~[1] `(list)`~)
++  test-append-02
  %+  expect-eq
    !>  ~[1]
    !>  (append ~ ~[1])
++  test-append-03
  %+  expect-eq
    !>  ~[1 2]
    !>  (append ~[1] ~[2])
```

for all inputs. Some values or ranges of values behave differently from others. (Unexpected UTF-8 whitespace characters are notorious for revealing head-scratching bugs—so imagine all possible inputs over complex XML.) You could model this behavior as multiple inductive types making for an even bigger cartesian explosion of inputs to test. It is no longer practical to construct all the tests required by our theory. The number is still finite, but impractically large. What is to be done?

There is no general solution. Complete code coverage with tests is a start, inductively testing over each clause in a unit. This is time-consuming and requires thinking deeply about the code and its structure.

Another approach is to favor so-called edge case testing, in which you test the boundaries of the input space. This is a good idea, but it is not a complete solution. It is not always clear what the edge is, and it is not always clear that the edge is the same for all inductive types. In the case of testing a string function, should an edge be the empty string, a string with one

character, or a string with two characters?

An automated testing solution called property-based testing may be applicable in these cases. The idea of property-based testing is to develop and instantiate invariant properties of the code and let software generate random inputs. The software runs the random inputs and tests the outputs against the invariant properties. An architecture for this approach was originally developed for Haskell and has since ported to many other languages, including Hoon. `%quiz` is a well-documented Hoon implementation (`hjorthjort/quiz`). Once again, this solution requires some deep thinking about the code and its specification.

From our experience with the F♯ implementation of property-based testing, we expect that you will need to boost the number of random input permutations beyond the default of 100 to get the kind of coverage that is "reasonably" exhaustive. We found 10,000 to be frequently adequate. (Since the inputs are randomly generated each run, however, a property test of production code may fail when nothing has changed. Then think about how to explain to your boss that your tests are not deterministic.)

Lastly, whenever you fix a bona fide production bug (or one that a framework like `%quiz` discovered), add a test case to address that circumstance. This not only provides the standard for when the bug has been fixed, but protects against regressing to the prior behavior (thus, a "regression test"). Congratulations, you have just refined your specification.

5 Testing Failure

In our experience failure modes are the most overlooked part of sofware development. It starts with passing insufficient, or no information from the programmed points of failure. So even before testing failure modes make sure you distinguish (i.e. make unique) each message from every point of failure and include any and all relevant information available for debugging (cf. Listing 6).

Listing 6: Error message for duplicate key

```
~|("cannot add duplicate key: {<row-key>}" !!)
```

The standard `/lib/test` library, as of this writing, can test for failure but not for an expected message. A modified testing arm (Listing 7) can be included in a testing thread to facilitate this kind of testing. Listing 8 shows a test for an expected error message in the Obelisk database engine which uses this functionality to verify that the correct error message is raised on crash.

6 Conclusion

Keep the following principles in mind when producing and evaluating code as a software developer.

1. Strive to make the collection of units of testing exhaustive, both primitive units of code and units of work from the user perspective.
2. Test inductively wherever possible.
3. Test from a white or gray box perspective.
4. Test the failure modes.
5. Test all the examples in your documentation.
6. In a world lacking documentation tests may be the only real specification.[7]
7. Regression tests are the key to refactoring with confidence. Beautiful code comes from refactoring. ⊠

[7] Cf. the definition by Feathers, p. xvi that "legacy code is simply code without tests." The entire text may thus be commended as a thorough guide to testing despite its name.

Listing 7: Testing for expected error message

```
::
::  +expect-fail-message
++  expect-fail-message
  |=  [msg=@t a=(trap)]
  ^-  tang
  =/  b  (mule a)
  ?-  -.b
    %|  |^
        =/  =tang  (flatten +.b)
        ?:  ?=(^ (find (trip msg) tang))
          ~
        ['expected error message - not found' ~]
        ++  flatten
          |=  tang=(list tank)
          =|  res=tape
          |-  ^-  tape
          ?~  tang  res
          %=  $
            tang  t.tang
            res   (weld ~(ram re i.tang) res)
          ==
        --
    %&  ['expected failure - succeeded' ~]
  ==
```

Listing 8: Obelisk database engine tests

```
::
:: fail on dup rows
++  test-fail-insert-dup-rows
  =|  run=@ud
  =/  my-insert
    "INSERT INTO db1..my-table (col1, col2, col3)  ".
    "VALUES ('cord',~zod,20) ('Default',Default, 0)"
  %+  expect-fail-message
    'cannot add duplicate key:'
  |.  %-  process-cmds
      :+  gen3-dbs  :: <- key 'cord' already exists
        (bowl [run ~2031.1.1])
      (parse:parse(default-database 'db1') my-insert)
```

References

Beck, Kent (2023) "Canon TDD". URL: https://tidyfirst.substack.com/p/canon-tdd (visited on ~2024.2.20).

~bithex-topnym, Rikard Hjort (2023) "Quiz: A randomized property testing library for Urbit". URL: https://github.com/hjorthjort/quiz (visited on ~2024.2.7).

Feathers, Michael C. (2005). *Working Effectively with Legacy Code*. Prentice Hall. ISBN: 978-0-13-117705-5.

~lagrev-nocfep, N. E. Davis (2022) "Writing Robust Hoon: A Guide to Urbit Unit Testing". URL: https://medium.com/dcspark/writing-robust-hoon-a-guide-to-urbit-unit-testing-82b2631fe20a (visited on ~2024.2.20).

~nomryg-nilref, Jack Fox (2023) "Sequent: A library of Hoon list functions for mortal developers". URL: https://github.com/jackfoxy/sequent (visited on ~2024.2.7).

~sorreg-namtyv, Curtis Yarvin (2006) “U, a small model”. URL: http://urbit.sourceforge.net/u.txt (visited on ~2024.2.20).

Wikipedia (2024a) “G. K. Chesterton, Chesterton’s Fence”. URL: https://en.wikipedia.org/wiki/G._K._Chesterton (visited on ~2024.2.7).

— (2024b) “Mathematical Induction”. URL: https://en.wikipedia.org/wiki/Mathematical_induction (visited on ~2024.2.20).

“Yard: A Developer Commons” (2023). URL: https://github.com/urbit/yard (visited on ~2024.2.7).

A Solution to Static vs. Dynamic Linking

Ted Blackman ~rovnys-ricfer
Urbit Foundation
Philip C. Monk ~wicdev-wisryt
Tlon Corporation

Abstract

Computing systems that utilize library code must either link to a self-provided version (static linking) or a system-supplied version (dynamic linking). This can lead to memory duplication or dependency hell. Urbit's `++ford` build system elegantly solves the linking problem by promoting structural sharing of objects (nouns) in memory and by utilizing a referentially transparent build cache.

Contents

Urbit Systems Technical Journal I:1 (2024): 75–82.

Address author correspondence to ~rovnys-ricfer.

1 Introduction

A compiled computer program is conventionally built by parsing and compiling the source code into an object file and then linking that object file to library code, yielding an executable file. That linking can be accomplished in two ways: either directly including all of the library code in the program file, or supplying the library code in the operating system as a service. Programmers have to balance flexibility, portability, and dependency management when deciding how to link a program, but both approaches can still lead to practical difficulties.

2 Static vs. Dynamic Linking

"Static linking" includes all library code in the program executable file. Static linking is the naïve way to combine source files into a program: compound the files together into one object and then compile that into a program. The semantics tend to be clean and simple, and many programs are linked this way. Statically-linked programs can also run faster since they do not need to resolve library references during execution.

The problem with static linking is that if the same library is used by more than one application, there is more than one copy of it in memory as a result. This memory duplication can be demanding on RAM utilization and reduce cache locality, degrading overall system performance.

Dynamic linking was invented to address this problem. Instead of linking a library into a program at build time, one links it at runtime. The OS keeps a single copy of the shared library in RAM that multiple programs can use. This reduces memory usage and improves performance, but can lead to version mismatches and dependency issues. Locklin explains,

> The shared object concept itself is a towering failure. This is little appreciated but undeniably true. The idea of the shared object is simple enough: if you have a computer running lots of code, some of the code used will be the same. Why not just

> load it to memory once and share that memory at runtime? … When people invented shared objects back in the 1960s, the computer was a giant, rare thing ministered to by a priesthood: there was no such thing as multiple versions of a shared object. You used what the mainframe vendor sold you.
>
> It's now such an enormous problem there are multiple billion-dollar startups for technologies for dealing with this complexity by adding further complexity. Docker, Kubernetes, various Amazon atrocities for dealing with Docker and Kubernetes and their competitors, Flatpak, Conda, AppImages, MacPorts, `brew`, RPM, NixOS, `dpkg/apt`, VirtualBox, `pacman`, Yum, SnapCraft…. As the number of packages grows, this breaks down, and even the OS maintainers are giving up and turning to flatpak, AppImages and Snap files.
>
> These are extremely complicated and incredibly wasteful *(of memory)* ways of literally packaging up a bunch of needed shared libraries with your application and presenting it to you as a crappy simulacrum of a statically compiled binary. (Locklin, 2022)

Locklin's picturesque exposition highlights the "dependency hell" or "DLL hell" that mires modern software development (cf. Grimes (2003)). From Urbit's perspective as a solid-state computer, another problem with dynamic linking is that it is not deterministic. Dynamic linking was something of a pact with the devil, permitting efficient memory usage at the price of legibility.

Programmers have to balance flexibility, portability, and dependency management when deciding how to link a program.

3 Building Code Deterministically

Determinism has long been a desired characteristic of any given build system. The source code should be a function of build environment and build instructions in a straightforward way. Even accessing the linked libraries in a different order can alter the resulting binary, however, meaning that true reproducibility is elusive. Declarative package managers like Nix (NixOS, 2024) and Guix (GNU Guix, 2024) use a functional package management approach to achieve reproducibility, marking packages using cryptographic hashes to track dependencies uniquely and repeatably.

In Urbit, compilation means converting Hoon source code (as text) into Nock code (as a binary tree). This process is handled by `++rash`, `++mint:ut`, and other components of the Urbit build system.

Linking, in the Urbit sense, derives from supplying nouns to nouns at compile time.[1] In the Urbit build arm `++ford`, a pair of builds becomes the build of a pair. The subject (environment) used to build a file is the tuple `[import_n … import_2 import_1 stdlib]`. Since Hoon symbol lookup is left-to-right, this nests scopes seamlessly and predictably. The linking technique is essentially trivial.

Ford compiles a Hoon source file into a data structure called a "vase", a pair `[type noun]` where `noun` is a member of the set of nouns described by `type`. Linking is thence just calling the `++slop` funciton, a one-liner from a pair of vases to a vase of the pair.

For example, the (trivial) Hoon source file with text `'3'` compiles to the vase `[[%atom %ud ~] 3]`. `3` is the value. Its type is an atom (number), tagged as an unsigned decimal (`%ud`) for printing.

As another example, the Hoon source file with text `'[3 0x4]'` compiles to

```
[[%cell [%atom %ud ~] [%atom %ux ~]] [3 0x4]]
```

[1] This is perhaps the biggest distinction from the conventional scenario for linking, which refers to RAM words. *Ceteris paribus*, this article's discussion can inform the traditional dialogue.

This is a vase of a `%cell` (pair) of unsigned decimal number (`%ud`) and unsigned hexadecimal number (`%ux`). If `/foo/hoon` is `'3'`, `/bar/hoon` is `'0x4'`, then importing both of those files changes the build subject to `[foo=3 bar=0x4 <stdlib>]` (or really, a vase of that value).

This is a form of static linking. The linking is performed at build time, not at runtime, and the resulting program contains its imports. "Relocation pointers" in C correspond to adjusting tree slots in Hoon, which the Hoon compiler does for the developer. Because Urbit uses static linking, it long had the same problem static linking has always had: memory duplication. If two different apps imported the same library, that library would be built twice and two copies of it would exist in memory.

In Urbit, everything is a "noun" (a binary tree with arbitrarily-sized integers at leaves). If you "copy" a noun `foo`, like `[foo foo]`, the runtime just copies a pointer to it. They are immutable, so everything shares structure. Nouns are "persistent" data structures, like Clojure's collections. If one copies a library, the copy is merely a pointer to the library—the library is a *noun*. If an imported library can be looked up from a build cache, the builder can copy it into a new app's build subject without duplicating it in memory.

But how does one know whether the cached library is valid? To achieve global (cross-application) deduplication, one needs a referentially transparent build cache, i. e. a build factored as a full description of an input to the build system. Since the build system is deterministic, if one sees the same input, one knows that it will produce the same output.

With one referentially transparent cache for all builds in the whole system, no invalidation is necessary. Cache eviction can take place efficiently due to reference counting, since which revisions of which apps refer to which builds is known.

As of #5745, Urbit supplies such a referentially transparent build cache. What this means in practice is that Urbit can have the memory deduplication benefits of dynamic linking while still using static linking. Since every filesystem snapshot lives at a unique, immutable, authenticated path within Urbit's scry namespace, reproducible builds are possible on every node, a

crucial feature for software supply chain security and reliable app distribution.

4 The Modern `++ford` Build Cache

The former Ford cache was per-desk,[2] keyed on the name of the build (e. g. a file at a certain path). It was impossible to share such a cache between desks because the name may refer to different things on different desks or at different revisions.

Since the caches weren't shared, they commonly held exactly the same data but generated independently. For instance, the same library used on different desks would be built repeatedly and the memory was not shared unless the user manually ran the `|meld` command to manually deduplicate nouns. For users with many apps installed, this added significant memory pressure.

A new cache was designed which is keyed on the name of the build plus its dependencies. This is all the input to a build except the standard library. Thus when one matches a key it does not matter which desk the value was on; the cache can be shared across all desks. When the standard library changes, all caches are cleared automatically.

Reclaiming space from this cache becomes important. Since this is a "true" cache, it's never incorrect to keep data in it. One could adopt a heuristic such as to clear hourly or so you could use a heuristic such as "clear the whole thing every hour" or a least-recently-used (LRU) policy. However, `++ford` has a long history of trying to use such heuristics and still using exorbitant amounts of memory. The primary innovation of the former `++ford` cache was that its size is deterministic, and it stored no builds unless they could be generated from the head of its desk (`%home` then `%base`).

These properties are extended to the global cache by counting references and maintaining a per-desk set of references to builds which are still relevant to the head of that desk. On every commit, the per-desk set of references is inspected to de-

[2] Clay desks in Urbit are like Git repositories, describing particular filesystem continuities.

termine which have been invalidated. Invalidated references are freed in the usual manner—their refcount is decremented, and if it's now zero, then it is deleted from the cache and all of its immediate dependencies are freed.

An alternative would be to garbage collect, which could be done on every commit to maintain determinism. However, this scales with the size of the cache (thus, with the number of desks installed), whereas refcounting scales only with the desk in question. Additionally, the cache is acyclic, and no manual refcounting is required—there is precisely one place where references are gained and one where references are lost.

As a result, the current cache has a "least upper bound" property: first, it minimizes the number of rebuilds required; given that, it minimizes the amount of memory required. In other words, cache entries are thrown away only when they become irrelevant. An alternate approach would be a "greatest lower bound"—throw away any cache entries that the system is not certain to need. This uses a bit less memory but results in more rebuilds. (It is also a little more complex to implement, since it requires clients to "register" the builds they want to keep warm in the cache, even if those builds didn't become invalid.)

Generating a cache key for this cache can be slow—hundreds of milliseconds is not uncommon, and it scales with the number of transitive dependencies. To mitigate this, a per-desk cache is included isomorphic to the former cache system. This has sub-millisecond lookup speed, and remains well-understood.

The current procedure to perform a build is thus:

1. Check if the build is in the per-desk cache; if so, return it generate its dependencies.
2. Check if the build, with these dependencies, is in the global cache; if so, return it.
3. Else, build it.

In each case, the new build is added to either cache if it was not already present; and if it was not in the per-desk set of references, it is added there as well.

5 Conclusion

Developers have long sought to balance the flexibility and portability of static linking with the better system demands of dynamic linking. Despite care, the balancing act has led from well-managed mainframe code into the current linker spaghetti situation. Urbit's `++ford` build system elegantly solves for the linking problem by promoting structural sharing of objects (nouns) in memory and by utilizing a referentially transparent build cache across desks. This balances efficiency in code compilation and building with the reliability of solid-state computing. ⊠

References

GNU Guix (2024) "GNU Guix transactional package manager". URL: https://guix.gnu.org/ (visited on ~2024.1.23).

Grimes, Richard (2003) ".NET and DLL Hell". URL: https://drdobbs.com/windows/net-and-dll-hell/184416837 (visited on ~2003.6.4).

Locklin, Scott (2022). *Managerial Failings: Complification.* Locklin on Science Blog. URL: https://scottlocklin.wordpress.com/2022/02/19/managerial-failings-complification/ (visited on ~2022.2.19).

NixOS (2024) "Nix and NixOS reproducible builds and deployments". URL: https://nixos.org/ (visited on ~2024.1.23).

~wicdev-wisryt, Philip C. Monk (2022) "`ford`: rewrite cache to share more builds #5745". URL: https://github.com/urbit/urbit/pull/5745 (visited on ~2022.5.3).

What Agreement Hath Mars With Earth? An Exploration of Urbit's IPC Interfaces

~mopfel-winrux
NativePlanet

Abstract

The idealistic Maxwellian principles of a system built on Nock are somewhat at odds with the evolved and tangled nature of legacy software ecosystems. In order to facilitate practical computing, Urbit presents several interfaces for communication with external ("Earth") systems. Through a comprehensive examination of Urbit's Arvo kernel and Vere runtime, alongside its high-level and low-level IPC interfaces (`%khan` and `%lick`, respectively), we elucidate the architectural mechanisms that facilitate communication and control between these disparate computing realms. This exposition aims to demystify Urbit's architecture and underscore its potential to bridge the gap between the familiar digital environments of Earth and the untapped possibilities of Martian computing.

Contents

Urbit Systems Technical Journal I:1 (2024): 83–92.

Address author correspondence to `~mopfel-winrux`.

1 Introduction

In Urbit's conception of software, two worlds exist in stark contrast: Earth, the familiar domain of established software paradigms, bustling with complex and varied digital ecosystems; and Mars, the enigmatic realm of Urbit, a system as pristine and archaic as the Martian landscape itself. This article will explore how Earth software establishes communication with Martian code.

Unlike Earth, where code is a tapestry of evolution and patchwork, Mars aims to present a regime of functional-only idealism built on Nock, the "Maxwell's laws of software". Urbit envisions a world in which software exists as a Maxwellian construct defined by a small number of axioms (`~sorreg-namtyv`, 2013). Urbit code is purely functional-as-in-language, and should be untouched and unaltered by the non-Maxwellian intricacies that characterize Earth's digital environments. Underlying this sequestration lies a profound awareness by the "Martians" that there exist diverse forms of computing with which their system must nevertheless engage. Mars accommodates these external entities by allowing them to interact in the Martian *lingua franca* of Nock nouns.

In the original public explanation of Urbit, `~sorreg-namtyv` explicitly charges,

> The general structure of cross-planet computation is that Earth always calls Mars; Mars never calls Earth. The latter would be quite impossible, since Earth code is non-Maxwellian. There is only one way for a Maxwellian computer to run non-Maxwellian code: in a Maxwellian emulator.

However, this pristine discipline has never been strictly adhered to for any real Urbit implementation. Urbit has long provided two kernel-level vanes (which one may think of as microkernel modules) for interfacing with Earth's HTTP Web services, `%eyre` (the server) and `%iris` (the client). One can even make a reasonably strong argument that Urbit running as Nock on a virtual machine specifically breaks the convention to some extent.

This article briefly explains Urbit's architecture, focusing on Arvo (Urbit's kernel) and Vere (Urbit's runtime VM). Following this, we explain three core parts of the Urbit OS interface: `%khan`, `conn.c` and `%lick`. These serve as distinct conduits between the terrestrial and the Martian, facilitating not just interaction but a nuanced form of command and control of the entire system. This exposition aims to demystify the architecture of Urbit and bridge the gap between Earth's established software paradigms and the Martian vision of computation.

2 Urbit's Unique Landscape

2.1 Arvo: The Heart of Martian Computing

The Arvo kernel (also referred to as "Urbit OS") represents a paradigm shift in operating system design. Unlike traditional operating systems that operate on preemptive multitasking and complex event networks, Arvo is a purely functional operating system, characterized by its deterministic nature and compact size. The entire Urbit stack is about 80,000 lines of code, with Arvo itself comprising around 2,000 lines. This small codebase is intentional, reflecting a philosophy that system administration complexity is directly proportional to code size.

Arvo's unique architecture avoids what is referred to as "event spaghetti" by maintaining a clear causal chain for every computation (`~sorreg-namtyv` et al., 2016). Each chain begins with a Unix I/O event and progresses through a series of steps until the computation completes successfully or fails. This deterministic nature allows for a high degree of predictability, a stark contrast to most Earth-based operating systems.

The kernel's design as a "purely functional operating system"—or, more accurately if idiosyncratically, an "operating function"—underscores its uniqueness. Arvo operates on the principle that the current state is a pure function of its event log, a record of every action ever performed. This determinism is a significant deviation from the normal situation that obtains on Earth, wherein operating systems allow for arbitrary programmatic alteration of global variables affecting other programs.

Arvo innovatively handles non-determinism by refusing to log non-completed events. The system, in essence, behaves like a stateful packet transceiver—events are processed, but there is of course no guarantee of any particular event successfully completing. This pattern mirrors the behavior of packet dropping in networking. Arvo's approach to handling non-determinism is unique and aligns with Urbit's overall philosophy of simplicity and determinism. Additionally, because Arvo runs on a VM (Vere), it can obtain non-deterministic information, such as stack traces from infinite loops, through the interpreter beneath it injecting events.

2.2 Vere: The Substrate of Martian Technology

Vere, the runtime of Urbit, plays a critical role in actualizing the Martian computing model on Earth-based hardware. As the Nock interpreter written in C, Vere is intricately optimized to run Arvo, ensuring seamless translation of its deterministic operations into practical execution on conventional systems.

Vere is architecturally split into two distinct parts: the `king` (the Earth interface, more or less) and the `serf` (the Mars substrate for Nock and Arvo). The `king` component is responsible for handling vane I/O and managing the effects on and from Unix. The `serf` provisions the Arvo virtual machine (VM) and related services. This partition allows for a clear delineation of tasks within the Vere runtime, enhancing its efficiency and robustness.[1]

[1]The Ares project, in development, seeks to cleanly replace the `serf` process, a contingency enabled by this division of responsibility.

Communication between the `king` and `serf` is achieved via an inter-process communication (IPC) port. This IPC mechanism facilitates a dialogue between the two components, ensuring that the `king` can effectively manage external interactions while the `serf` maintains the integrity of the Arvo VM.

Formally, Nock (and thus Urbit) is functional-as-in-language and free of side effects. In practice, the `%hint` opcode 11 in Nock is used to pass some noun to the interpreter. The interpreter can then choose a higher order action to take based on this value, whether I/O, memory management, or other side effects. This allows for the `king` to pass hints to the `serf` which in turn produces effects on Earth (sends a network packet, retrieves keyboard input, etc.). This bridging of the Earth/Mars gap allows for vanes to interact outside of Mars.

Event processing is key to Vere's functionality. Events from the Unix environment are passed to Vere, which then translates and injects them into Arvo. This mechanism demonstrates a profound integration between Vere and Arvo, where Vere possesses direct knowledge of Arvo's state. Notably, Vere can "scry" or query Arvo's state without altering it, retrieving information as needed. This capability allows Vere to maintain the integrity of Arvo's deterministic model while enabling dynamic interaction with the outside world.

Urbit's runtime operation is effected by using Unix as its BIOS. By leveraging the robust, widely-used Unix system, Vere ensures that Urbit can run on a broad range of hardware platforms, effectively making Urbit's Martian technology accessible and operable in Earth's diverse computing environments. This strategic use of Unix not only provides the necessary I/O and optimizations for Arvo but also ensures that Urbit's advanced and unique software architecture can function in tandem with the existing, established hardware and software ecosystems of Earth.

Moreover, Vere's design allows it to inject events into Arvo, a feature crucial for maintaining the fluid communication and interaction between the Urbit system and its underlying hardware and software infrastructure. This bidirectional communication channel ensures that Arvo can respond to external stimuli while remaining true to its functional and determin-

istic nature. Such events, from Arvo's perspective, are simply incidental injections to the event loop.

Thus, Vere serves as more than just a bridge between Martian technology and Earthly hardware; it is a finely tuned conduit, optimized to uphold and facilitate the unique computational paradigm that Arvo embodies.

3 %khan: The High-Level Thread Interface

The %khan vane functions as a high-level thread interface for managing both internal and external communications within the Urbit ecosystem. %khan is designed to efficiently handle complex I/O operations, employing an I/O monad coupled with an exception monad. This approach allows %khan to manage complex I/O tasks while robustly handling potential failures.

%khan's core design revolves around the use of 'threads', which are monadic functions designed to handle arguments and produce results. Threads are optimized for I/O operations, leveraging the I/O monad for structured and efficient task handling. The exception monad further ensures robust management of potential failures and unexpected events. An Arvo thread is impermanent and may fail unexpectedly; it usually relies on some process external to Arvo with unknown existence or reliability. In most of its intermediate states, it expects only a small number of events (usually one), so if the thread receives anything it does not expect, it fails. Running threads are also not upgradable and fail across code upgrades. Threads are fragile but well-suited for their use case in managing such tenuous I/O operations.

The %khan vane was conceived as a way to control Urbit ships from the exterior using threads.[2] %khan's conception evolved a fair bit from proposal to implementation. In practice, %khan is essentially an interface wrapper for %spider-based threads,[3] which produces the topsy-turvy situation in which a vane relies on a piece of userspace infrastructure to function

[2] %khan was originally called the "control plane" after the network routing distinction between the control plane and the data plane.

[3] %spider is a %gall agent to manage transient thread-level operations.

correctly. `%khan` allows for pre-written threads that allow for easy hosting and maintenance to be bundled and distributed, helping both hosting companies and self-hosters to manage Urbit instances efficiently.

4 `conn.c`

The `conn.c` driver in Vere[4] forms part of the `king` process. It exposes a Unix domain socket at `/path/to/pier/.urb/conn.sock` for sending and receiving data from external processes.

`conn.c`'s functionality, particularly its ability to dispatch messages and handle various types of requests, is critical for the command of Urbit's control plane. It serves as a way for Unix processes to receive insights about what is happening on Urbit. `conn.c` also allows for `%khan` and Vere generally to inject raw kernel moves (events, if successful) into the system.

From a technical perspective, `conn.c` accepts newt-encoded `++jam`med nouns (defined below) which take the form

```
[request-id command arguments]
```

Newt encoding is a way of communicating nouns as single data objects in an unambiguous format.

```
V.BBBB.JJJJ.JJJJ...
```

where `V` is the version (currently 0); `B` is the size of the `++jam`med noun in bytes (little-endian unary); and `J` is the `++jam`med noun (little-endian). This structure allows for a variety of commands to be executed, including:

1. `%ovum`, the injection of raw kernel moves;
2. `%fyrd`, a direct shortcut to `%khan` commands;
3. `%urth`, runtime subcommands like `%pack` or `%meld`;
4. `%peek`, namespace scry requests into Arvo; and
5. `%peel`, emulated namespace scry requests into Vere.

[4] `vere/io/conn.c`

A valid `conn.c` command produces a newt-encoded `++jammed` noun with type `[request-id output]`, where:

- `request-id` matches the input `request-id`; and
- `output` depends on the `command`.

An invalid `conn.c` command produces a newt-encoded `++jammed` noun with type `[0 %bail error-code error-string]`.

5 `%lick`: The Low-Level IPC Interface

Although also dealing with interprocess communication, the `%lick` vane (`~mopfel-winrux`, 2023) was designed for a very different use-case than `%khan`: to allow external processes, in particular hardware drivers, to intercommunicate with Urbit. `%lick` focuses on creating a generic noun interface over Unix domain sockets.

`%lick` manages IPC ports and the communication between Urbit applications and POSIX applications via these ports. Other vanes and applications request `%lick` to open an IPC port, notify it when something is connected or disconnected, and transfer data between itself and the Unix application. `%lick` works by opening a Unix socket for a particular process, which allows serialized IPC communications. These involve a `++jammed` noun so the receiving process needs to know how to communicate in nouns. The IPC ports texttt%lick creates are Unix domain sockets (`AF_UNIX` address family) of type `SOCK_STREAM`. Connections are made via files in a directory under the pier, `.urb/dev`.

The process on the host OS must therefore strip the first 5 bytes, `++cue` (unjam) the `++jammed` noun, check the mark, and (most likely) convert the noun into a native data structure.

To understand what `%lick` is doing, we need to briefly examine Unix's IPC model. IPC ("interprocess communication") describes any way that two processes in an operating system's shared context have to communicate with each other. `%lick` focuses on Unix domain sockets, which are merely file-like communication endpoints.

Listing 1: Usage of `%lick` cards.

```
++  init  [[%pass / %arvo %l %spin /control]~ this]
++  on-arvo
  |=  [=wire =sign-arvo]
  ?+    sign-arvo  (on-arvo:def wire sign-arvo)
      [%lick %soak *]
    ?+    mark.sign-arvo  [~ this]
        %connect
      :_  this
      :~  :*  %pass  /spit  %arvo  %l
              %spit  /control  %init  area.state
      ==  ==
  ==  ==
++  send-state
  |=  =state
  ^-  card:agent:gall
  :*  %pass  /spit  %arvo  %l
      %spit  /control  %state
      [slick:state face.state food.state live.state]
  ==
```

For instance, a valid use of `%lick` would use cards to communicate (Listing 1). Outbound moves are specified as `%spit`.

`%lick`'s vane definition is even simpler than `%khan`'s: it has no `++abet` nested-core pattern and primarily communicates to a `unix-duct` provided in its state. The `+$owner` is a `+$duct` to handle the return `%soak`.

Arvo will send three types of `%soaks` to an open `%lick` port. These `%soaks` may be one of:

1. `%connect` when the first connection is established on the Earth-side IPC port;
2. `%disconnect` when the last connection is broken on the Earth-side IPC port; or
3. `%error` when an error occurs.

`%lick` also will send a `%disconnect %soak` to every agent when Vere is started.

Gall needs to wrap `%soak` and `%spit` to route properly (see e. g. `++ap-generic-take`). This permits multiple agents to share sockets with the same name while still organizing an agent's sockets by a unique path.

6 Conclusion

This exposition of Urbit's IPC interfaces, in particular `%khan`, `conn.c`, and `%lick`, has sought to cast light upon architectural innovations that distinguish Urbit from traditional computing paradigms. These mechanisms permit Vere to balance Arvo's demand for pristine determinism against the host Unix OS's many competing software interfaces and tools. While Urbit itself obviously relies deeply on these, following this pattern in other systems could lead to a redefinition of how digital ecosystems interact. ⊠

References

~mopfel-winrux (2023) "UIP-0101: %lick. An IPC Vane". URL: https://github.com/urbit/UIPs/blob/main/UIPS/UIP-0101.md (visited on ~2024.1.25).

~sorreg-namtyv, Curtis Yarvin (2010) "Urbit: functional programming from scratch". URL: http://moronlab.blogspot.com/2010/01/urbit-functional-programming-from.html (visited on ~2024.1.25).

— (2013) "Nock 4K". URL: https://docs.urbit.org/language/nock/reference/definition (visited on ~2024.2.20).

~sorreg-namtyv, Curtis Yarvin et al. (2016). *Urbit: A Solid-State Interpreter*. Whitepaper. Tlon Corporation. URL: https://media.urbit.org/whitepaper.pdf (visited on ~2024.1.25).

The Desert of the Reals: Floating-Point Arithmetic on Deterministic Systems

N. E. Davis ~lagrev-nocfep
Urbit Foundation

Abstract

Floating-point calculations are critical to a number of special domains in modern computing, including machine learning, graphics, and scientific computing. Numerical calculations are particularly susceptible to opaque and system-local optimizations, which can break certain guarantees for deterministic computers. We consider the background and implementation of IEEE 754 floating-point arithmetic and options for implementing mathematics compatibly with fully reproducible and portable computing. We consider hardware-based and software-based proposals.

Contents

Urbit Systems Technical Journal I:1 (2024): 93–131.

Address author correspondence to ~lagrev-nocfep.

1 Introduction

> Floating-point operations are a technically complex subject and the extent to which developers or source code alter or test this information will depend on many factors. Apart from the general exhortation to developers to be careful and to make sure they know what they are doing, there is little of practical use that can be recommended. (Jones (2008), p. 197)

Modern digital computers deal, at their root, in binary representation, entirely zeros and ones.[1] These are often formally considered to be whole numbers in a number base of two. However, numerical calculations very frequently require the use of numbers with a fractional part to adequately represent the elements of a computation.

[1] Analog computers may operate on a continuum of value. Computer logic systems may be architected on other numeric bases for their logic, such as the 1837 Analytical Engine's decimal system and the 1958 Setun's (Сетунь) ternary system.

Early numeric computing tended to focus on problems of interest to military and national security applications, such as the solution of differential equations and numerical optimization. Such calculations typically involve arrays, and linear algebra was elaborated hand-in-hand with digital computing techniques in software and hardware. Numerics assumed prominence for a wider audience with the rise of gaming on personal computers, although these algorithms emphasized speed over exactness.[2] To this point in the history of computing, most software either ran on a single platform for its lifetime (as with supercomputing) or did not require portably deterministic algorithms (as with gaming).[3]

On the other hand, deterministic computing describes the ability for a given computation be reproducible exactly. Such reproducibility permits referential transparency and more powerful reasoning about a program's results and dependencies. This includes, for Urbit as a state machine, that the event log replay be portable across platforms to yield the same result. Conceptual guarantees must be backstopped by actual implementation guarantees for determinism to hold.

2 A Derivation of the Real Numbers

Binary computer values are at root easily represented as nonnegative integers. However, it is frequently convenient when working with human applications to either use other numeric bases (notably decimal and hexadecimal) or to permit noninteger mathematics.

In the historical development of mathematics, logical problems in each set of numbers drove the discovery and elaboration of more elaborate algebras. For instance, in the field of natural numbers $\mathbb{N}$, the operation of addition $+$ or multiplication $\times$ produces a value within the set; however, permitting

[2] This is reflected in algorithms such as the "fast inverse square root", which permits a degree of inaccuracy in exchange for a substantial speedup.

[3] To be clear, 3D gaming algorithms are deterministic (assuming no random sources are used), but exact error values are often not reproducible across platforms, nor was such portability a design criterion. The reasons for this are discussed below.

subtraction $-$ of a larger number from a smaller number can result in a value inexpressible in $\mathbb{N}$. This motivated the introduction of the integers $\mathbb{Z}$, augmenting the numbers from zero to (positive) infinity with the negative numbers. Division $/$ similarly produced a crisis when applied to values which did not have a whole-number ratio between them, a situation resolved by the Pythagorean[4] innovation of the rational numbers or fractions as a class $\mathbb{Q}$. Ultimately, the common reference set for engineering mathematics (and the human understanding of the continuum such as measurement) is the set of real numbers. The set of real numbers, denoted by $\mathbb{R}$, is characterized by its continuity, implying that for any two distinct values within this set, there exists a difference, no matter how small.

Since the operations and conventions of $\mathbb{R}$ have been found to be so useful, it is desirable to extend the semantics to computer programming. However, digital computers, by virtue of their binary representation, effectively use natural numbers $\mathbb{N}$ to represent numbers (to the limit of memory rather than positive infinity $+\infty$). Several schemes permit a computer integer to be interpreted as if it were a number with a fractional part, including a scaling factor, fixed-point representation,[5] and pairs as rational numbers.

The basic concept of floating point arithmetic is that it permits the representation of a discrete subset of $\mathbb{R}$ by composing a significand, a base, and an exponent. The significand is the set of significant digits, possibly including the sign; the base is the understood number base (typically 2); and the exponent is the power to which that base is put before multiplying by the significand to yield the result.[6] The most ubiquitous floating-point

[4] A legendary attribution, alas, predicated on the Pythagorean discovery of the irrational numbers as a separate class thereby inducing a crisis (Huffman, 2024).

[5] Hand-in-hand with the development of linear algebra, machines such as ENIAC and MANIAC were employed in the 1940s for solving thermonuclear reaction calculations and neutron diffusion equations. Under the direction of John von Neumann, it appears that some calculations did experimentally involve a floating-point scheme, although this was later rejected definitively in favor of fixed-point arithmetic. See Kahan (1997), p. 3, on floating-point representations.

[6] Compare so-called "engineering" notation, such as `1e5` for 10,000, which

format today is defined by the IEEE 754 standard, but certain hardware platforms such as GPUs utilize alternative floating-point arithmetic representations.[7]

To summarize, given an abstract description of a floating point system, there are several practical implementations that can be derived. We need to specify at least four quantities: sign,[8] significand, base, and exponent.[9] The base is presumably fixed by the protocol, leaving three free values for the implementation to economically encode.

2.1 IEEE 754 Basics

Early computer systems with floating-point units chose bespoke but incompatible representations, ultimately leading to the IEEE 754 (primarily architected by William Kahan). IEEE 754 reconciled considerations from many floating-point implementations across hardware manufacturers into an internally consistent set of fixed-width representations.[10] For instance, the 32-bit "single precision" C `float`/Fortran `REAL*4` specification denotes particular bit positions as meaningful,

```
SEEE.EEEE.EFFF.FFFF.FFFF.FFFF.FFFF.FFFF
```

where `S` is the sign bit, `0` for positive ($+$) and `1` for negative ($-$); `E` is the exponent in base-2 (8 bits); and `F` is the significand (23 bits). The exponent is actually calculated at an offset bias of 127 (2^7) so that a more expressive range of orders of magnitude can be covered. The significand has an implied leading `1` bit unless all are zero. To wit,

$$(-1)^S \times 2^{E-127} \times 1.F$$

compactly represents the significand 1 and the exponent 5 with an understood base of 10 indicated by `e` and a sign, implicit for positive.

[7] We cite `bfloat16` (Wang and Kanwar, 2019) and TensorFloat-32 (Kharva, 2020), among others.

[8] We could omit the sign by introducing an offset or only allowing positive values.

[9] The exponent has a bias so that it in turn does not need a sign.

[10] We ignore the decimal representations introduced in IEEE 754-2008, which do not materially change our argument.

IEEE 754 specifies operations between numbers, including of different magnitudes. The standard dictates behavior and provides outlines for arithmetic, but leaves algorithmic details to the implementation. Numbers are normalized by adjusting the exponent of the smaller operand and aligning the significands, then the operations are carried out. In practice, extended precision values are used in the intermediate steps of many algorithms, leading to greater accuracy than would otherwise be expected.[11]

Since the IEEE 754 floating-point format packs values of different kind together bitwise, conventional integer operations such as left shift $<<$ and addition $+$ do not trivially apply.[12]

Floating-point addition (`add`) proceeds per the following algorithm:

1. Compare exponents of the two numbers. Shift the smaller number rightwards until its exponent matches the larger exponent.
2. Add the significands together.
3. Normalize the sum by either shifting right and incrementing the exponent, or shifting left and decrementing the exponent.
4. If an overflow or an underflow occurs, yield an exception.
5. Round the significand to the appropriate number of bits.
6. Renormalize as necessary (back to step 3).

[11] "Each of the computational operations that return a numeric result specified by this standard shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that intermediate result, if necessary, to fit in the destination's format." (IEEE, 2008) Note that, per Risse, "there is no indication whether or not a computation with IEEE 754 is exact even if all arguments are." The ISO/IEC 9899 C standard confesses its own fallibility: "The floating-point model is intended to clarify the description of each floating-point characteristic and does not require the floating-point arithmetic of the implementation to be identical" (ISO/IEC (2018), fn. 21).

[12] For instance, left-shifting a floating-point value does not double it; we leave the mathematics of why as an exercise to the reader.

IEEE 754 floating-point arithmetic and its predecessors have some significant mathematical compromises even in its formal specification. For instance, as a result of the discrete nature of the bitwise representation in `E` and `F`, floating-point mathematics are actually a subset of discrete mathematics masquerading as real mathematics. This has non-trivial consequences for certain aspects of calculations, including error accrual. In particular, three facts dominate the resolution:

1. The distance between two adjacent values changes based on the magnitude of the exponent and the distance from zero. (The significand resolution stays the same but the exponent changes.)

2. There is a relative approximation error for a given bitwidth in IEEE 754, called the *machine epsilon*.[13]

3. Operations between numbers of different magnitudes are particularly affected by their relative numerical horizon.

Variable precision and truncation error. For most values of the exponent `E`, the difference between two discrete values is determined by the absolute magnitude of the significand `S`. The difference between serial values is

$$\begin{aligned}\Delta S &= 000.0000.0000.0000.0000.0001_2 \\ &= 1.0000001192092895507812 5 - 1.0 \\ &= 0.0000001192092895507812 5 \\ &= 2^{-23}. \end{aligned} \tag{1}$$

This is multipled by the the result of the exponent `E` and the bias, meaning that for each exponent value the difference between subsequent values changes. (Figure 1 represents this schematically.)

[13] Note that this is different from the smallest representable value for a given bit width; e. g., for 32-bit single-precision `float` the smallest representable value is $0000.0000.0000.0000.0000.0000.0000.0001 = 1 \times 10^{-45}$.

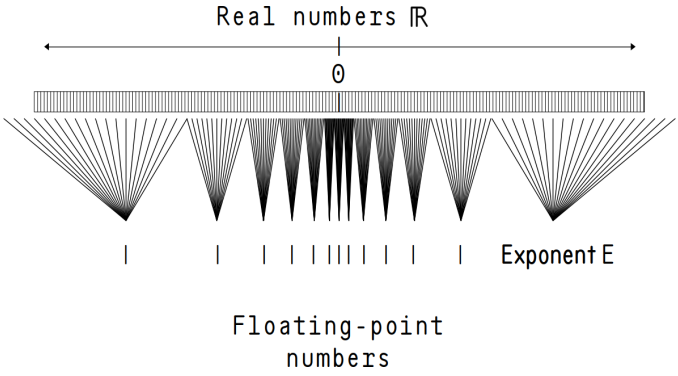


Figure 1: Schematic representation of granularity and variable precision of floating-point values and their relationship to the (continuous) set of real numbers.

However, for normalized numbers, or numbers that are left-shifted or right-shifted in order to carry out a calculation, values are determined by the *relative shift* in exponent ΔE. For $E = 2$, for instance, the difference between serial values is 2^{-21}. This variable precision means that the precision of floating-point values varies across the range of representable numbers when operations take place. Operations between two numbers of fairly different precisions are particularly vulnerable to accuracy loss, although some numerical techniques can be employed to mitigate.

Truncation error results from terminating repeating "binaries" (by analogy with "decimals"). Just as $\frac{1}{3} = 0.\bar{3} = 0.3333\ldots$ has a finite precision when written in base-10, numbers that are not precise powers of two result in repeating fractions. These necessarily terminate at the resolution of the significand. The number and nature of truncation and rounding can significantly affect the accuracy of floating-point arithmetic and algorithms (Izquierdo and Polhill, 2006).

Machine epsilon. The machine epsilon, or smallest value discernable from 1.0, is determined by the precision of the

floating-point representation. The machine epsilon for a particular bit width is determined by setting two to the negative power of the number of bits used for the magnitude of the mantissa and accounting for the leading implicit bit 1; for 32-bit single-precision `float` this is $2^{-23-1} = 2^{-22}$. Differences from 1.0 smaller than this cannot be represented in this bit width.

Sequence ordering. In situations in which floating-point operations may occur in different orders, even the basic guarantee of communitativity breaks. For instance, in 64-bit floating point arithmetic, the following holds true (example in Python):

```
In [1]: (1.1-0.3)-0.8
Out[1]: 0.0

In [2]: (1.1-0.8)-0.3
Out[2]: 5.551115123125783e-17
```

This occurs since operations of different magnitude can affect the resulting significand, a sort of horizon of resolution leading to differences in the outcome. Sequence order can be changed (and thus commutativity broken) as a result of many common programmer design patterns, including compiler optimizations, race conditions, and parallelization.

Another problem in numerical analysis, error accrual is likewise due to the horizon of resolution. The accrual of error due to summing sequences of numbers (whether in parallel or serially) occurs in the summation of sequences of numbers since the error term can grow as n. Kahan–Babuška compensated summation can be used to track a separate error term (*de facto* extending the precision during the operations) and adding it back in to the sum before yielding the final result (Kahan (1965), Babuška (1969)).

Formally neither associative nor commutative for the above reasons, floating-point arithmetic can break our mathematical intuitions in interesting ways. However, this is a consistent and well-understood phenomenon. For our purposes as designers of deterministic computers, the most damning indictment has to do not with IEEE 754 itself but with manufac-

turer deviation in hardware implementation. In 1997, William Kahan himself complained (justly) about the compromises inherent in the standard for compiler implementers:

> Most computer linguists find floating-point arithmetic too disruptive [due to] [t]heir predilection for "referential transparency" .... Computer linguists also dislike functions with side-effects and functions affected by implicit variables not explicit in argument lists. But floating-point operations can raise IEEE 754 exception flags as side-effects, and operations are affected implicitly by exception-handling and rounding modes eligible at run-time according to IEEE 754. Alas, that standard omitted to bind flags and modes to locutions in standard programming languages, and this omission grants computer linguists a licence for inaction. (Kahan, 1997)

There are several sources of trouble for even single-threaded deterministic computation using hardware IEEE 754 floating-point units (FPUs):[14]

1. Optional, discretionary, or advisory aspects.
2. Gaps or omissions in the specification.
3. Failure to implement the specification exactly.
4. Out-of-sequence computations.

Optional aspects. Several aspects of IEEE 754 are optional or advisory, including:

1. Exception handling means that the hardware may specify rounding via an overflow flag.

[14]We do not lay blame at the feet of any particular party; the facts are the facts. Indeed, a more recent revision of IEEE 754 leads with a call for portability: "This standard provides a discipline for performing floating-point computation that yields results independent of whether the processing is done in hardware, software, or a combination of the two" (IEEE, 2008).

2. Extended precisions formats are not a huge deal to leave out, but extended precision arithmetic (used for intermediate results) can materially change results.

3. Subnormals are optional;[15] some platforms may flush them to zero or (worse) allow subnormal support to be disabled in certain cases.[16]

Omissions. Whether something is a gap or optional is something of a philosophical question for us, but some parts are underspecified in a way that makes portability impossible. E. g., mixed-precision operations can yield unpredictable results depending on the compiler and hardware. This is a function of rounding modes, precision loss, precision of intermediate results, and the presence or absence of dedicated hardware support for certain precision combinations.

Inexact implementation. Failure to implement IEEE 754 correctly may happen inadvertently, as with the Pentium FDIV bug in the 1990s (Edelman, 1997). Alternatively, chipset designers may deviate from the specification for reasons of performance or limitations in the architecture.

For instance, IEEE 754 defines a range of numbers as "not-a-number" values, or NaNs. Per the specification, a NaN can be a signalling NaN, meaning that it intends to flag and possibly disrupt a problematic computation;[17] or a quiet NaN, which does not raise such an exception and merely yields a result with the

[15] Subnormals are a convention that allows values smaller than the "normal" IEEE 754 smallest non-zero value. They permit a graceflow underflow behavior, and can prevent an unintentional division by zero.

[16] "Some processors do not support subnormal numbers in hardware" (Jones (2008), p. 338). (The risk is that this permits inadvertent division by zero.) Various chipsets solve this exceptional behavior differently.

[17] "C support for signaling NaNs, or for auxiliary information that could be encoded in NaNs, is problematic. Trap handling varies widely among implementations. Implementation mechanisms may trigger signaling NaNs, or fail to, in mysterious ways. The IEC 60559 floating-point standard recommends that NaNs propagate; but it does not require this and not all implementations do." (Jones (2008), p. 339)

NaN propagated to the final result.[18] Not all processors implement this part of IEEE 754 correctly: "The Motorola DSP563CCC does not support NaNs or infinities. Floating-point arithmetic operations do not overflow to infinity; they saturate at the maximum representable value" (Jones (2008), p. 338).

As a further example, fused multiply-add (FMA) ($a \times b + c$) is implemented on certain hardware to favor `double` operations and not quadruple-precision operations (Kahan (1997), p. 5).

Out-of-sequence computations. A modern compiler using optimization flags or even modest parallelism can easily cause a floating-point calculation to rely on operands that were produced in an order different than that specified in the code. This is largely opaque to the programmer, aside from some simple heuristics, and makes it difficult to reproduce or reason about the fine details of computations.

As demonstrated above, out-of-sequence or resequenced computations can affect results due to rounding behavior and the "numerical horizon" which results between values. These can happen due to multithreaded computation or an optimizing compiler.

Rounding mode. IEEE 754 floating-point operations take place using one of several rounding modes, for instance,

1. Round to nearest, ties to even. Set ties to the last bit as zero (even). The default.

2. Round to zero. Truncate, effectively rounding positive numbers down and negative numbers up.

3. Round away from zero. Truncate, effectively rounding positive numbers up and negative numbers down.

4. Round toward positive infinity. Up regardless of sign.

[18] In Urbit, the Vere runtime unifies NaNs, meaning that any bitwise information which may be encoded in the significand field—the "NaN payload"—is thrown away. This is called the "canonical NaN".

5. Round toward negative infinity. Down regardless of sign.

The rounding mode can affect the result of computations, and if other processes are changing the mode (which can even be set per-thread), results may not be reliably reproducible.

"Obtaining the correctly rounded result of an addition or subtraction operation requires an additional bit in the significand (as provided by the IEC 60559 guard bit) to hold the intermediate result" (Jones (2008), p. 65).

3 Urbit's Implementation of IEEE 754

Urbit implements a subset of IEEE 754 functionality in `/sys/hoon`, the Hoon language specification. The Nock operations formally take place on integers. In practice, we could imagine several ways of implementing such operations: bitmasking the integers or breaking them apart into three components, for instance. We take Urbit's implementation of `@rs` (single-precision `float`) as representative.[19]

`++rs` is a wrapper core to instrument arithmetic arms like `++add` using the `++ff` floating-point functionality core. Ultimately this resolves to breaking out the components (sign, exponent, and significand) into separate numbers for the actual operation.[20]

The `++fn` core offers a generalized interface for a superset of IEEE 754-style floating-point implementations, permitting bit width, precision, bias, and rounding mode to be freely specified.[21] The actual implementation on `+$fn`-typed values is rather dense and features numerous rounding and overflow checks:

[19]The only significant variation in the other real types in Hoon arises for quadruple-precision floating-point values `@rq` which are represented in the runtime by a pair of `uint64_t`s.

[20]The decimal output is produced using the traditional Steele–White Dragon4 algorithm (Steele and White, 1991). It is worth considering upgrading Hoon from Dragon4 to Errol (Andryso, Jhala, and Lerner, 2016) for speed and accuracy.

[21]In practice, of course, Urbit hews to recognized types, but the temptation to design new floating-point layouts is intriguing.

```
++  add
  |=  [a=[e=@s a=@u] b=[e=@s a=@u] e=?]  ^-  fn
  =+  q=(dif:si e.a e.b)
  |-  ?.  (syn:si q)  $(b a, a b, q +(q))
  ?:  e
    [%f & e.b (^add (lsh [0 (abs:si q)] a.a) a.b)]
  =+  [ma=(met 0 a.a) mb=(met 0 a.b)]
  =+  ^=  w  %+  dif:si  e.a  %-  sun:si
    ?:  (gth prc ma)  (^sub prc ma)  0
  =+  ^=  x  %+  sum:si  e.b  (sun:si mb)
  ?:  =((cmp:si w x) --1)
    ?-  r
      %z  (lug %fl a &)  %d  (lug %fl a &)
      %a  (lug %lg a &)  %u  (lug %lg a &)
      %n  (lug %na a &)
    ==
  (rou [e.b (^add (lsh [0 (abs:si q)] a.a) a.b)])
```

There is, of course, a feint in the foregoing discussion. Nock is a virtual machine specification, and in practice operations that would benefit from more direct expression in C are *jetted*.[22] Thus the actual call in this case will correspond to some C code using the SoftFloat library:[23]

```
u3_noun u3qet_add(u3_atom a, u3_atom b, u3_atom r) {
  union sing c, d, e;
  // set IEEE 754 rounding mode
  _set_rounding(r);
  // unwrap nouns into C-typed values
  c.c = u3r_word(0, a);
  d.c = u3r_word(0, b);
  // perform addition and unify NaN
  e.s = _nan_unify(f32_add(c.s, d.s));

  // wrap C value back into noun
  return u3i_words(1, &e.c);
}
```

[22] A shorthand for *jet-accelerated code*.

[23] `u3` functions are Urbit noun library functions. The `sing` union is a union of `uint32_t` and SoftFloat `float32_t` types.

Why SoftFloat? Enter, stage left, the problem of platform-portable determinism.

4 Deterministic Computation with a Fractional Part

Non-real arithmetic is less significant for many of the core operations of Urbit as a personal server platform. However, gaming, machine learning, graphics, and other applications rely on floating-point calculations—preferably as fast as possible. In fact, not only applications-oriented processes rely on determinism: guarantees in cryptography and contractual correctness for web3; verification and validation; accounting and legal compliance; and code correctness analysis all require reproducible determinism.[24]

Why can't we just allow different results in the last binary places of the significand? Philosophically, Urbit holds to the following statements (`~wicdev-wisryt`, 2020):

1. A10. Correctness is more important than performance.
2. A12. Correctness is more important than optimality.
3. A14. Deterministic beats heuristic.
4. F1. If it's not deterministic, it isn't real.

Urbit makes much of avoiding the "ball of mud" "standard software architecture" (Foote and Yoder, 1999). In this design antipattern, a lack of guarantees and predictable behavior leads *inevitably* to haphazard and illegible software bloat. We can thus understand why Urbit as a platform considers even deviations in the last bit of a significand to be threads fraying the edge of sanity (`~wicdev-wisryt`, 2020):

[24] Support for IEEE 754 is similar for support for the Markdown markup language. Many platforms support a subset of Markdown coupled with platform-specific extensions. (See also SQL.) Internal references, HTML, inline LaTeX math mode, code block language specification, and other features see varying levels of support with GitHub, Pandoc, Obsidian, and other editors and converters.

> If you do the same thing twice, your computer should react the same way. This is comforting. This is also what makes it easy to reason about and use effectively. If you're not sure what your computer will do, you'll be afraid of it and act defensively toward it. This inevitably leads to a big ball of mud.

For most purposes in the broader software world, tightly reproducible precision has not been a high priority. Precision having already been sacrificed, the gist of the calculation is more important than the fourth decimal place (e. g. in realtime 3D graphics). This leads to the phrase "implements the IEEE 754 standard" being interpreted erroneously to imply full reproducibility (Figueroa del Cid, 2000).

For example, consider the expression $(a \times b) + c$. If a compiler permits the two operations to be evaluated sequentially (a multiplication followed by an addition), then rounding occurs twice. If a compiler optimizes the operation into an FMA, or fused multiply-add, then a single rounding occurs. Peters presents a pathological case for 32-bit single-precision floating-point values: $a = 1.00000011920929$, $b = 53400708$, and $c = -b$. In this case, the two-stage operation wipes out the 0.00000011920929 component of a, yielding a as an integer. Then c is added and the result is 8. With FMA as a single-step operation, the (correct) answer 6.365860462 is obtained. The optimization is more correct than the naïve route in this case.

However, in another example due to Dawson, FMA yields incorrect results: for $a \times b + c \times d$ with $a = c$ and $b = -d$, the answer should be zero, and calculated in two steps will typically be zero. With a fused multiply-add, however, the code becomes `fmadd(a, b, c*d)`, rounding the multiplication of c and d but not that of a and b; the answer will likely not be zero.

The situation grows more ambiguous across architectures. Jones (2008), p. 346, presents the pathological case of a compliant platform that may use extended precision bits in the calculation of `a + b`:

```
#include <stdio.h>

extern double a, b;

void f(void) {
    double x;
    x = a + b;
    if (x != a + b)
        printf("x != a + b\n");
}
```

In this hypothetical case, "any extended precision bits will be lost in the first calculation of `a+b` when it is assigned to `x`. The result of the second calculation of `a+b` may be held in a working register at the extended precision and potentially contain additional value bits not held in `x`, the result of the equality test then being false." Higham (2002) provides further examples of pathological cases.

K&R C permitted the compiler to re-order floating-point expressions by associativity, which could run afoul of our limitations. ANSI C (C89), recognizing the issue introduced by this innocuous change, forbade such re-ordering (MacDonald, 1991). Compiler optimizations (e. g. GCC's `-O3`) can bypass this restriction, once again breaking determinism;[25] for instance, floating-point operations can be pipelined, leading to out-of-order execution.

The fly in the ointment for Urbit's deterministic computing is that jet-accelerated Nock equivalents must reliably produce the same results (both to each other and to Nock) regardless of the runtime on which it is being evaluated. Thus even small irregularities in floating-point implementations have macroscopic ramifications for deterministic computing. Any guarantee broken breaks them all, just as it would for a formal correctness proof.[26]

The challenge of the lack of determinacy for certain crit-

[25] This can be mitigated in turn by the use of the `volatile` designation, but this is sufficient to illustrate the problem.

[26] I was once asked by a retired computer science professor if such guarantees would make things easier. Well, at the end developer level!

ical applications has been acknowledged before, such as by James Demmel and the ReproBLAS team (Demmel et al. (2017), Ahrens, Nguyen, and Demmel (2018)) and by Dawson. Dawson makes much of the effect of rounding modes and the option to disable subnormals, both of which would have major effects on computational reproducibility. The situation is worse for transcendental functions, because there is necessarily truncation and/or rounding error (Dawson, 2013). Even conversion between bases for output and input is not necessarily reproducible, as Dawson continues: "Doing perfect conversions efficiently was an unsolved problem when the original IEEE standard came out, and while it has since been solved, this doesn't mean that everybody does correctly rounded printing." Koenig (2018) expend much effort on the problem of computing a dot product exactly in hardware, given the contingencies of multicore processors.[27]

The field of debate for possible solutions for implementing floating-point arithmetic which is portable across platforms includes:

1. Hardware-supported floating-point arithmetic.
2. Software-defined floating-point library.
3. Opaque calculations.
4. Stored results.
5. Proscribing IEEE 754.

We consider each in turn, with its ramifications for a deterministic computing platform and in particular its prospects for adoption in Nock-based systems.

[27]The problem of exactness is not exactly the same as reproducibility, but it is related. Exactness means that the result is the same as if the calculation were carried out to infinite precision and then rounded to the appropriate number of bits. Reproducibility means that the result is the same across different platforms. Reproducibility requires identical outcomes from the same inputs, while exactness requires the correct answer regardless of algorithmic path.

4.1 Hardware-supported floating-point arithmetic

As outlined above, execution of code-equivalent floating-point computations produced from source by different compilers on different hardware architectures may lead to small differences in outcome, non-negligible for a deterministic computer. Thus, for this and a constellation of related reasons, hardware-supported floating-point arithmetic seems to be *prima facie* unviable for deterministic computing.[28]

We do not know the field of possible future hardware architectures which Nock as a deterministic computing platform may be called upon to execute.[29] Jet-accelerated code should be intelligently robust about its the hardware, but Hoon and Nock code should be completely agnostic to the hardware.

That's the problem. What are some possible hardware-targeted solutions?

1. Control the compiler and runtime stack top to bottom.

2. Store a hardware and compiler tag and simulate when not on that platform.

3. Support only a single hardware for the lifetime of a ship.

4. Dock floating-point results.

5. Check consistency of results.

[28] Although not of grave consequence, the C language (as of C23) does not implement at least two types specified by IEEE 754-2019 and recent predecessors: `binary128` quadruple precision and `binary256` octuple precision. While neither are significant losses, we also note that Urbit does not currently support a C-style `long double` type. C's `long double` is 80 bits wide on some common consumer hardware, such as the x86-64 architecture, but is 128 bits wide on the 64-bit ARM architecture. (The situation is worse for Python, whose `numpy.float128` type eponymously advertises itself as quadruple precision but is in fact a regular 80-bit `long double`.) Some compilers and libraries do support quadruple-precision floating-point mathematics, such as GCC's `__float128` type. We note that IEEE 754 80-bit extended-precision could be implemented using the `++fn` core should demand arise.

[29] Indeed, we do not know the future specifications which may be implemented to provide approximations of real values in either hardware or software.

4.1.1 Control the stack

If you controlled the compiler and runtime execution stack to a sufficient degree, could you yield deterministic floating-point arithmetic from the hardware? "A translator that generates very high performance code is of no use if the final behavior is incorrect" (Jones (2008), p. 189); that is, optimizations often come at the cost of correctness.

To start off, what must be considered part of the stack in this sense? At a minimum, the compiler and linker toolchain (including flags and options) and the actual runtime must be included. (This explicitly introduces a dependence between Martian software and Earthling software, repugnant to the Urbit ethos.)

We also must decide what the target is. Do we aim for the most portable configuration (as determined by number of consumer or enterprise users)? Do we aim for the "closest" to IEEE 754 adherence? Do we aim for simplicity, or compilation speed, or any of a half-dozen other optimizable variables?

For instance, suppose that one intended to use the C keyword `volatile` to block certain common optimizations on a floating-point value.[30] The runtime at the level of Nock does not know if a value is considered floating-point or not. At the level of a jet, the use of `volatile` can correctly bar certain hardware optimizations, but these need to be carefully enumerated and understood in the light of the other toolchain concerns enumerated in this section. Strictly speaking, `volatile` only seeks to guarantee that stale calculations are not inadvertently

[30] In any case, this assumes a legible and enumerable set of behaviors for `volatile` which is, alas, not the case. "`volatile` is a hint to the implementation to avoid aggressive optimization involving the object because the value of the object might be changed by means undetectable by an implementation" (Jones (2008), p. 472). "Actions on objects so declared shall not be 'optimized out' by an implementation or reordered except as permitted by the rules for evaluating expressions" (*ibid*, p. 1500). "The `volatile` qualifier only indicates that the value of an object may change in ways unknown to the translator (therefore the quality of generated machine code may be degraded because a translator cannot make use of previous accesses to optimize the current access)" (*ibid*, p. 963). The same author provides examples of C code that is ambiguous in `volatile`'s semantics, pp. 1290–1291; and undefined in `volatile`'s semantics, pp. 1482–1483.

reused due to optimization. Without hardware optimization, the utility of an [FPU] for fast floating-point computations is questionable. The risk of a jet mismatch remains high, as does a nonportable jet.[31]

Can the C-defined floating-point environment (as supplied by `fenv.h`) answer to this need? This affords the ability to specify not only rounding modes and access floating-point exception status flags, but it is not clear whether this environmental control portably spans the entire output of floating-point computations.[32]

Finally, "[a]n implementation is not required to provide a facility for altering the modes for translation-time arithmetic, or for making exception flags from the translation available to the executing program" (Jones (2008), p. 200). The information we purport to gain by controlling the stack in the manner above outlined is possibly not even available to the compiler and the runtime executable.

We suggest that deterministically correct stack control in the sense we have described here is impossible for an arbitrary configuration of the modern hardware stack.[33]

4.1.2 Simulate the hardware

If you knew what the compiler and execution stack behavior looked like when a calculation was performed, could you reproduce it in software at need on a different platform?

[31] "What constitutes an access to an object that has volatile-qualified type is implementation-defined" (*ibid*, p. 1488). "Volatile-qualified objects can also be affected by translator optimizations" (*ibid*, p. 1490). The C novice may at this point wonder what the intended utility of `volatile` in fact is: "[a] `volatile` declaration may be used to describe an object corresponding to a memory-mapped input/output port or an object accessed by an asynchronously interrupting function" (*ibid*, p. 1499).

[32] "The floating-point environment access and modification is only meaningful when `#pragma STDC FENV_ACCESS` is set to `ON`. ... In practice, few current compilers, such as HP aCC, Oracle Studio, and IBM XL, support the `#pragma` explicitly, but most compilers allow meaningful access to the floating-point environment anyway." (C++ Reference, 2023)

[33] The possibility of circumscribing the set of permissible IEEE 754 operations, which may afford a different approach to this problem but seems similarly susceptible of shipwreck, is explored in a subsequent section, *q. v.*

Hardware simulation faces some difficulties in the same vein as controlling the stack. The proposal yields a combinatorial explosion when considering the combinations of hardware chips, compilers, and compiler flags. Nor is it clear that hardware documentation can be accrued in sufficient quantity and detail to guarantee the success of such a project.

The Urbit runtime provides an epoch system, meaning that the event log is separated into snapshots and subsequent events (`~mastyr-bottec`, 2023). This is currently used to monitor the use of old binaries which could potentially have a jet mismatch. It would be moderately straightforward to extend this functionality to record the compilation flags and architecture of that Vere binary, which could be useful in event playback. However, this remains an unsatisfactory solution because it would lead to Urbit runtime instances intentionally producing different code (rather than a jet mismatch which would require correction).

4.1.3 Support a single hardware platform

> Marriage is a fine institution, but I'm not ready for an institution. (Mae West)

A permanent commitment to a single hardware platform—either for the Urbit platform as a whole or for a particular running instance—could solve the determinism problem. This configuration would be tenable for single-purpose ships with lifetime control (likely moons or comets), but inconvenient for the "hundred-year computer" model touted for planets and superior ranks in Urbit.

To make a lifelong commitment to a particular hardware platform when the lifetime of a deterministic computer is unknown is therefore deemed foolhardy.

4.1.4 Dock floating-point results

What about trimming floating-point values of their least significant bits? When would this take place—at each step of a multi-step computation? At the level of single-bit rounding errors, this would potentially work, and amounts to selecting

a rounding mode towards even (last digit `0`). Accrual across multiple calculations could potentially render this unreliable, particularly if different computational paths are supposed to lead to the same result and do not as a result of docking.

One could also envision docking more than the last bit. This introduces a step to check and adjust the floating-point value, and in addition breaks IEEE 754 compliance—at which point the trouble of trying to reconcile IEEE 754 with determinism fails.

In general, we cannot assign a high degree of significance to figures beyond the first few, but accruals across large data sets (such as large language models) can become significant (as attested to by the need for compensated summation).

A related technique could pack bits of larger floating-point values into smaller ones, but this is functionally a software-defined solution (see, e. g., Brun (2018)).

4.1.5 Consistency checks

Another option is to compare Nock and jet code for every computation and only accept the C code if it is "correct". This immediately runs into a very undesirable characteristic: every floating-point calculation is run twice, obviating at least one calculation and destroying any efficiency gains from jetting the code.

One could cache floating-point computations somewhere in the system.[34] This is liable to become prohibitively large for systems as every individual floating point calculation of all time becomes archived against future need.

We conclude that, at the current time, naïve hardware-defined floating point is not viable for deterministic systems.

4.2 Software-defined floating-point library

In the absence of a dedicated floating-point unit (FPU) and floating-point assembly instructions, floating-point computations are carried out in software. The type can be decomposed from bits, operated on, then packed back into the single type of

[34]Indeed, something like this cache system was employed on Sun SPARC architecture, as discussed in Section 4.4.

appropriate value. For instance, prior to the widespread advent of 64-bit consumer hardware, applications that needed `double` values on 32-bit PC architecture utilized software emulation using two 32-bit numbers together.

Urbit's current solution for floating-point computation is to utilize a software-defined floating-point library, the SoftFloat library by Hauser. SoftFloat is an implementation in software of a subset of IEEE 754 for five floating-point types.[35] Urbit statically links the library into its runtime binary so it is always available for Nock to utilize as a jet.

While formally correct, software FP is slower than hardware floating point, and likely prohibitively slow for many large matrix applications such as LLMs. ("Correctness is more important than performance.") Performance is the dolorous stroke against software-defined floating point. (On the other hand, some early versions of the Apple–IBM–Motorola PowerPC RISC architecture did not have dedicated hardware floating-point units (FPUs) or floating-point assembler instructions at all, requiring full software implementation.[36]) GCC has supported a software floating-point mode using `fp-bit.c` (GNU Project, 2008); this was particularly used to accommodate the PowerPC limitations rather than to provide either speed or determinism (cf. Sidwell and Myers (2006)).

An optimized portable deterministic software library for floating-point calculations may be a sufficiently fast solution to meet Urbit's needs even for vector computations. A different avenue worthy of investigation is to take IEEE 754 compliant floating-point values as inputs and outputs, then transform into a local representation for an optimized portable deterministic calculation. For instance, Thall (2007) presents the concept of "unevaluated sums", a generalized technique for accru-

[35] "The current release supports five binary formats: 16-bit half-precision, 32-bit single-precision, 64-bit double-precision, 80-bit double-extended-precision, and 128-bit quadruple-precision" (Hauser, 2018).

[36] "There are several 680x0-based Macintosh computers that do not contain floating-point coprocessors" (Apple Computer, 1994); on the other hand, "floating-point calculations are performed even faster under the … emulator than on a real 680x0-based Macintosh computer," indicating that optimized software acceleration is possible, modulo chipset versions and tuned libraries. The PowerPC 601, introduced in 1991, had a native FPU.

ing error in situations where additional precision is necessary for accuracy. However, even with an agreed-upon standard library like SoftFloat, it is important to keep in mind that exact floating-point results for transcendental functions are still not correctly known in many cases.[37] This particular poses a problem for functions like sin which may be calculated by different routes in Hoon/Nock and in C/Rust. For the time being, we conclude that Urbit's discipline requires only using Hoon/Nock implementations of transcendental functions.

4.3 Opaque calculations

When a request for data is made over the network, one is not certain what the resulting data will be. Their value is epistemically opaque. In Urbit's event log, the results of network calls are persisted as effects in the modified state (for successful events).

What if Urbit treated a call that had a floating-point computation as if it were a network call, that is, as if it were a referentially opaque injection into Urbit's state? One difference is that network calls result as side effects from hints to the runtime which then handles the plumbing, as it were, and injects the resulting `gift` task back into Arvo as if a *deus ex machina*, from Arvo's perspective. (It should of course know how to handle such a contingency.) There are two main objections that can be made here:

1. From the programmer's standpoint, every floating-point computation would need to be bundled as if it were a network call, and the result treated as if it were a new move passed back into the kernel. This destroys synchronicity and changes floating-point computations from lightweight programmer choices into heavy and occasional calls.

2. The storage of every result of every floating-point computation could become prohibitively large. Work on

[37] To correctly calculate a trigonometric function for `double` may take over a hundred bits of precision before correct rounding can be determined. Furthermore, the C `math.h` implementation of sin may or may not use `fsin`.

> large matrices in numerical analysis or machine learning could rapidly balloon the event log since every intermediate state would also become part of the ship's immutable history.

To the first objection, we can point to the current design pattern utilized in scrying (or the request for values from the bound scry namespace). Local scry values (such as values exposed by a system service or vane) are accessed synchronously using the `.^` dotket operator. This is straightforward and easy to integrate into a program. Remote scry values must be requested asynchronously from another ship, and return at an indeterminate future time as `gifts` to be processed in another part of the vane or application.

To the second, we observe that although Urbit is a state machine whose history is part of its state, in practice we can mitigate event log growth by either *chopping* the event log by storing its state and permitting replay forward from that point or *tombstoning* data which should never be available again.[38]

In a successful implementation of this scenario, one could imagine distinguishing slower software execution (treated synchronously) from faster hardware acceleration (treated asynchronously).

4.4 Stored results

Instead of repeating computations that have been made in the past, what if we cached the result of all of them, so that any new computations with the same values are guaranteed to result in the same value via a cache lookup instead of a calculation? Urbit uses memoization frequently in Arvo and in the runtime, so this is an aesthetically compatible option; we consider its feasibility.

> A recently proposed hardware acceleration technique is to store the results of previous multiplication and division operations in a cache, reusing

[38] The memory implications of these are not necessary here, but take place in different arenas: the runtime versus the Arvo noun arena.

> rather than recalculating the result whenever possible. (Dynamic profiling has found that a high percentage of these operations share the same operands as previous operations.) (Jones (2008), p. 1148)[39]

On Urbit, this introduces an $O(1)$ average-case/$O(n)$ worst-case cache lookup from a MurmurHash3 hash key calculation (what Urbit calls a `++mug`). This must be weighed against the floating-point algorithm in consideration, as well as what is actually hashed (likely the Nock of the calculation contained in the dynamic hint).

This bears some similarities to aspects of the network call suggestion above, in that the second objection to that one holds here. Event log and state bloat (via the cache) are liabilities. Such a cache would be a feature of the Arvo instance, not the runtime VM. Unlike a truncated event log, the cache must be a permanent feature of the ship's state rather than a convenience.

"Storing results" could also be met by the use of SPARC-style logging. In that hardware platform, suspicious computations are flagged and hashed into a lookup table by site in the originating program. Such events are logged not by timestamp or by computation hash but by callsite in the originating program (Kahan (1997), p. 6).[40] Sun implemented this in SPARC for "retrospective diagnostics" but the technique could allow a more lapidary operation for Urbit. (Follow-on considerations include whether such computations should now be considered "bound" in a sense like that of the scry namespace.)

4.5 Proscribing IEEE 754

What if the Scylla of IEEE 754 is avoided for some Charybdis? We can approach this solution space at two levels: either by sector or entirely.

[39] Cf. Jones' citation of Citron, Feitelson, and Rudolph.

[40] What constitutes "suspicion" is only sparsely elaborated by Kahan in that article.

Proscribe by sector. One solution to the speed-vs.-reproducibility dilemma is to permit hardware-accelerated IEEE 754 operations, but only in a verified subset permissible for jets. This would require careful vetting of the hardware stack and compiler options to define a permissible subset of IEEE 754 operations as "known good". Coupled with the epoch system, it may be a feasible solution.

What degree of vetting will reliably answer the gap between IEEE 754 and hardware implementation for any particular operation? (Jones (2008), pp. 330ff.) and Goldberg (1991) provide a careful analysis of accuracy errors inherent to IEEE 754 as a standard, but due to the variety of possible scenarios do not treat of real compilers and chipsets much.[41] Trivially, as demonstrated above in the Python example, $(a + b) + c \neq a + (b + c)$, and even modest reordering of operations by a zealous compiler optimization is susceptible of introducing nonportable and thus nondeterministic (in our sense) behavior.

Having identified an appropriate subset of operations, we may imagine that the use of `#ifdef`, Autotools' `configure`, and a jetting library may answer to our need. Any jet library would have to be carefully constructed to avoid imposing tight discipline directly on the end user (modal Hoon author). We cannot recommend this path today but do not consider the way to be shut, especially given liberal use of `volatile`.

In particular, fused multiply-add operations are subject to reordering by an optimizing compiler. Avoiding these would require some discipline on the part of the jet developer, since code that does not explicitly `fma` may yet reduce to it in a compiler pass. A jetting library would be advantageous in this case.

As an example of a refactoring of IEEE 754 operations for determinism, consider the `ReproBLAS` project (last update ~2016.2.21). `ReproBLAS` seeks to produce a set of reproducible deterministic algorithms reflecting the standard operations of BLAS (Ahrens, Nguyen, and Demmel, 2018). It accomplishes this by introducing a binned data type and a set of basic operations carefully built on IEEE 754 for the objective of completely

[41] See particularly the note on "Common Implementations" on p. 346 of Jones (2008).

portable reproducibility.[42] This is similar to our proposal for a vetted jetting library and may be worth attention, particularly in association with requirements around `-O0`.

Proscribe by replacement. Finally, we face the possibility of jettisoning decades of floating-point libraries entirely and forging a new trail. We explicitly omit attempting to implement a new standard as hubristic, but would like to explore some alternatives.

Posits. In 2015, John Gustafson proposed a new standard for representing values drawn from $\mathbb{R}$ called universal numbers or unums (Gustafson, 2015).[43] The current version supports interval arithmetic and greater resolution near 1.0, at the cost of decreased resolution for extremely large and extremely small values. Unums also guarantee associativity and distributivity of operations.

Gustafson's criticisms of IEEE 754 focused on determinism and exactness; underflow and overflow; fixed bit widths for mantissa and exponent; rounding; and the large wasted block of NaNs (Risse, 2016). Unums likewise must provide sign, exponent with bias, and significand; they may additionally signal whether the value is an interval. Unlike IEEE 754's use of multiple bit widths, 32-bit "posits" (fixed-size unums intended to facilitate hardware requirements) are argued to be sufficient for almost all applications.

The unum proposal appears to have settled somewhat after its initial state of relative flux (as Type III unums, cf. Posit Working Group (2022)). Although most implementations have been in software, the project has been specified in Verilog several times and implemented on FPGAs (Chen, Al-Ars, and Hofstee (2018), and VividSparks, `https://vividsparks.tech/`). In-

[42] "Using our default settings, reproducibly summing n floating point types requires approximately $9n$ floating point operations (arithmetic, comparison, and absolute value). In theory, this count can be reduced to $5n$ using the new 'augmented addition' and 'maximum magnitude' instructions in the proposed IEEE Floating Point Standard 754-2018." (Ahrens, Nguyen, and Demmel, 2018)

[43] See also Gustafson (2017), Gustafson (2017), and Gustafson and Yonemoto (2017).

triguingly, some initial work has been carried out towards a fundamental BLAS-like library built on posits (van Dam et al., 2019).

A unum/posit implementation for Urbit would be as straightforward as its implementation of IEEE 754. For jetting, there is a software library for posits available called SoftPosit based on the SoftFloat library (Cerlane, 2018). (Until commercial hardware implementations become available, the effect of optimizations on determinism cannot yet be assessed; it is presumed that the situation will be better than IEEE 754 given the advantages of a clean slate.)

Hand-rolled `floats`. If IEEE 754 presents too many difficulties to be viable at high speed, then hand-rolling a custom hybrid hardware–software scheme via bitmasking could be attractive. This returns to the more "Wild West" days before IEEE 754's introduction, but is presaged by the recent introduction of `bfloat16`, `TensorFlow-32`, and other types designed for machine learning applications. Without access to hardware manufacturers, however, this amounts in the end to software-defined floating point and seems unlikely to be competitive speedwise. (We cite the idea put forth previously in this article to convert to an intermediate representation for computation, yielding IEEE 754 as necessary.)

It may also be worth considering the use of a 3-tuple of sign, exponent, and significand (with only software jetting), and leave details of jet implementation to library authors. Hoon provides such a primitive in `++fn`, a tuple for base-2 floating-point arithmetic supplying fields for sign, a signed exponent without bias, and an arbitrary-precision significand.

Fixed-point and $\mathbb{Q}$. A fixed-point representation differs from a floating-point scheme in that the exponent is fixed by the protocol or metadata and thus only the sign and significand need be included in the bit representation. (With an offset, even the sign can be elided.) The advantage of such a scheme is that it affords the benefits of floating-point mathematics at near-integer operation speeds (e. g. left-shift to multi-

ply by two). One disadvantage is that there is a smallest representable value; this lack of subnormals requires either an underflow handler or the possibility of inadvertent division by zero. Fixed-point operations could also be used as intermediates in calculations. (This echoes once again the idea of conversion to an intermediate representation then conversion back out to IEEE 754.)[44]

If a rational number scheme is implemented, then a variety of possible implementations are possible, ranging from bitpacked fixed-width integers to pairs of arbitrary-width integers. Reduction to "simplest" values introduces some overhead; fractions are formally an ordered pair (a, b) with $b \neq 0$, but there is an equivalence class of multiples. (That is, if we write $\frac{1}{2}$ as $(1, 2)$, we have also to consider $(2, 4)$, $(3, 6)$, indeed an infinite sequence of such ordered pairs.) Rational numbers are a superset of floating-point numbers and fixed-point numbers, but accrue processing overhead due to dereferencing arbitrary integers and other aspects of computation on operations.

However, deviation from the proscription scheme, even inadvertently, would mean that a ship is considered invalid in a sense equivalent to double-booting or breaking the scry namespace. This option is deemed worth investigation, likely viable, but bearing unknown risks.

4.6 Irregularities

Any approach to modeling real numbers runs the risk that different calculation pathways will yield a different kind of inexactness in the result. These can be mitigated by some of the approaches suggested above, and also by checking the correspondence of the Hoon code and the underlying jet, particular for known edge cases in behavior. While Hoon–jet compliance is an open research problem,[45] we can apply principles of unit

[44] "One solution to implementing floating-point types on processors that support fixed-point types is to convert the source containing floating-point data operations to make calls to a fixed-point library" (Jones (2008), p. 346). Note that the sense of our current interest is reversed.

[45] Particularly as regards co-generation of Hoon and C/Rust, or formal proofs.

testing together with a period of testing Nock and jet compliance.[46]

Jet mismatches have been rare in the current era.[47] Some jet "mismatches" occur because the runtime raises a different error than the corresponding Hoon—these are relatively innocuous. Others may occur because actually different results are produced for different input. These are grave, and ultimately motivated the introduction of the epoch system so that event log replays can take into account the previous less-perfect jet version in the runtime (`~mastyr-bottec`, 2023).

5 Linear Algebra in Hoon

Lagoon[48] is an Urbit library to facilitate Hoon-native mathematical operations. It envisions six native types,

1. `%real`, an IEEE 754 floating-point value
2. `%uint`, an unsigned integer
3. `%int2`, a twos-complement signed integer
4. `%cplx`, a BLAS-compatible ordered pair
5. `%unum`, a unum/posit value
6. `%fixp`, a fixed-precision value

for which `%real` allows the rounding mode to be specified; `%cplx` consists of a pair of two values, real and imaginary parts; and `%fixp` requires the expected precision.

Lagoon implements algorithmically correct reference implementation in Hoon with the expectation that `/lib/lagoon` will be jetted. Operations include basic arithmetic, vector and

[46]The Vere runtime supports a debugging flag which runs both the Nock and the jet and checks for identical results.

[47]By "current era", we mean after the last global network breach on ~2020.12.8 (Tlon Corporation, 2020).

[48]Linear AlGebra in hOON

matrix row/column operations, matrix multiplication, and matrix inversion. The jetting scheme may take advantage of software libraries or appropriate hardware, but must hew to the dictum that "if it's not deterministic, it isn't real."[49]

Lagoon has passed through several implementations and remains in active development. The current implementation is the `lagoon` branch of the `urbit/urbit` repository (Urbit Foundation, 2023).

6 Conclusion

To summarize, the most promising solutions for floating-point mathematics on Urbit per the above analysis include:

1. Hardware floating point on single machine for entire lifetime.
2. Optimized software floating point with vetted jetting library.
3. Opaque calculation as callback.
4. Cached results by callsite.
5. Utilizing a subset of IEEE 754 in hardware.
6. Replacing IEEE 754 with another approach of sufficient speed, fixed-point and unum/posits chief among these.

Several recent efforts on Urbit have encountered difficulties in producing reliably deterministic and sufficiently fast

[49]It is a worth a final digression to address reproducibility on parallel systems. We do not consider this a design goal for Lagoon at the current time. Operations like reduction take place on a single computer; while jets may in principle utilize parallelism their points of entry and exit are unique. However, we note that the `ReproBLAS` project has addressed this issue in the context of reproducible parallelism (Ahrens, Nguyen, and Demmel, 2018), as have Chohra, Langlois, and Parello (2016).

floating-point calculations on a Nock-based system.[50] We anticipate that, water finding its own level, each will adopt a suitable deterministic solution for evaluation in Nock. We do not anticipate these to be the last foundational numerical libraries built on Urbit, but instead among the first. Thus we have documented the paths we have explored as an annotated map for future travelers in search of a one true representation for continuous mathematics. ⊠

References

@KloudKoder (2022) "Floating-point rounding mode control prototyping (WebAssembly Issue #1456)". URL: `https://github.com/WebAssembly/design/issues/1456` (visited on ~2024.3.10).

Ahrens, Peter, Hong Diep Nguyen, and James Demmel (2018) "`ReproBLAS`: Reproducible Basic Linear Algebra Subprograms". URL: `https://bebop.cs.berkeley.edu/reproblas` (visited on ~2024.3.10).

Andryso, Marc, Ranjit Jhala, and Sorin Lerner (2016). "Printing Floating-Point Numbers: A Faster, Always Correct Method." In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*, pp. 555–567. DOI: 10.1145/2837614.2837654. URL: `https://dl.acm.org/doi/10.1145/2837614.2837654` (visited on ~2024.3.10).

Apple Computer, Inc. (1994). *Inside Macintosh: PowerPC System Software*. Boston: Addison-Wesley. URL: `https://developer.apple.com/library/archive/`

[50] Cf. UrWasm, which as a WebAssembly implementation directly relies on floating-point computations (`~dozreg-toplud`, 2023) (see e. g. @KloudKoder (2022) for a discussion of the issues involved, and in particular `~dozreg-toplud`, pp. 133–150 in this issue), `Quodss/urwasm`; `/lib/math`, implemented purely in Hoon, `sigilante/libmath`; and Lagoon, which includes IEEE 754 operations with its linear algebra library, `urbit/numerics` pending development and integration into Urbit.

documentation/mac/pdf/PPC_System_Software/Intro_to_PowerPC.pdf (visited on ~2024.3.10).

Babuška, Ivo (1969). "Numerical stability in mathematical analysis." In: *Information Processing* 68, pp. 11–23.

Brun, Laurent Le (2018) "Making floating point numbers smaller". URL: https://www.ctrl-alt-test.fr/2018/making-floating-point-numbers-smaller/ (visited on ~2024.3.10).

C++ Reference (2023) "Floating-point environment". URL: https://en.cppreference.com/w/c/numeric/fenv (visited on ~2024.3.10).

Cerlane, Leong (2018) "SoftPosit". URL: https://gitlab.com/cerlane/SoftPosit (visited on ~2024.3.10).

Chen, Jianyu, Zaid Al-Ars, and H. Peter Hofstee (2018). "A matrix-multiply unit for posits in reconfigurable logic leveraging (open) CAPI." In: *Proceedings of the Conference for Next Generation Arithmetic*, pp. 1–5.

Chohra, Chemseddine, Philippe Langlois, and David Parello (2016). "Efficiency of Reproducible Level 1 BLAS." In: *Scientific Computing, Computer Arithmetic, and Validated Numerics*. Ed. by Marco Nehmeier, Jürgen Wolff von Gudenberg, and Warwick Tucker. Berlin: Springer International Publishing, pp. 99–108.

Citron, D., D. Feitelson, and L. Rudolph (1998). "Accelerating multi-media processing by implementing memoing in multiplication and division units." In: *Proceedings of 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pp. 252–261.

Dawson, Bruce (2013) "Floating-Point Determinism". Random ASCII – tech blog of Bruce Dawson. URL: https://randomascii.wordpress.com/2013/07/16/floating-point-determinism/ (visited on ~2024.3.10).

Demmel, James et al. (2017) "A Proposal for a Next-Generation BLAS". URL: https://docs.google.com/document/d/1DY4ImZT1coqri2382GusXgBTTTVdBDvtD5I14QHp9OE/edit#heading=h.jtgipeoidy9 (visited on ~2024.3.10).

~dozreg-toplud, K. Afonin (2023) “UrWasm”. URL: https://github.com/Quodss/urwasm (visited on ~2024.3.10).
— (2024). “The urwasm WebAssembly Interpreter Suite on Urbit.” In: *Urbit Systems Technical Journal* 1.1, pp. 133–150.
Edelman, Alan (1997). “The Mathematics of the Pentium Division Bug.” In: *SIAM Review* 39.1, pp. 54–67. DOI: 10.1137/S0036144595293959.
Figueroa del Cid, S. A. (2000). “A Rigorous Framework for Fully Supporting the IEEE Standard for Floating-Point Arithmetic in High-Level Programming Languages.” PhD thesis. New York University.
Foote, Brian and Joseph Yoder (1999) “Big Ball of Mud”. URL: http://laputan.org/mud/mud.html (visited on ~2024.3.10).
GNU Project (2008) “Software Floating Point”. URL: https://gcc.gnu.org/wiki/Software_floating_point (visited on ~2024.3.10).
Goldberg, David (1991). “What Every Computer Scientist Should Know About Floating-Point Arithmetic.” In: *ACM Computing Surveys* 23.1, pp. 5–48. URL: https://dl.acm.org/doi/pdf/10.1145/103162.103163 (visited on ~2024.3.10).
Gustafson, John L. (2015). *The End of Error: Unum Computing*. A. K. Peters/CRC Press. ISBN: 978-1-4822-3986-7.
— (2017a) “Beyond Floating Point: Next Generation Computer Arithmetic (Stanford Seminar)”. URL: https://www.youtube.com/watch?v=aP0Y1uAA-2Y (visited on ~2024.3.10).
— (2017b) “Posit Arithmetic”. URL: https://posithub.org/docs/Posits4.pdf (visited on ~2024.3.10).
Gustafson, John L. and Isaac Yonemoto (2017). “Beating Floating Point at its Own Game: Posit Arithmetic.” In: *Journal of Supercomputing Frontiers and Innovations* 4.2, pp. 71–86. DOI: 10.14529/jsfi170206.
Hauser, John R. (2018) “Berkeley SoftFloat Release 3e”. URL: http://www.jhauser.us/arithmetic/SoftFloat-3/doc/SoftFloat.html (visited on ~2024.3.10).

Higham, Nicholas J. (2002). *Accuracy and Stability of Numerical Algorithms*. 2nd ed. Philadelphia: SIAM.

Huffman, Carl (2024). "Pythagoras." In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta and Uri Nodelman. Spring 2024. Metaphysics Research Lab, Stanford University.

IEEE (2008). *754-2008 IEEE Standard for Floating-Point Arithmetic*. Tech. rep. Institute of Electrical and Electronics Engineers. URL: https://ieeexplore.ieee.org/document/4610935 (visited on ~2024.3.10).

ISO/IEC (2018). *ISO/IEC 9899:2018 Information technology – Programming languages – C*. Tech. rep. International Organization for Standardization. URL: https://www.iso.org/standard/74528.html (visited on ~2024.3.10).

Izquierdo, Luis R. and J. Gary Polhill (2006). "Is your model susceptible to floating point errors?" In: *Journal of Artificial Societies and Social Simulation* 9.4. URL: https://www.jasss.org/9/4/4.html (visited on ~2024.3.10).

Jones, Derek M. (2008). *The New C Standard: An Economic and Cultural Commentary*. URL: http://www.knosof.co.uk/cbook/cbook.html (visited on ~2024.3.10).

Kahan, William (1965). "Further remarks on reducing truncation errors." In: *Communications of the ACM* 8.1, p. 40. DOI: 10.1145/363707.363723.

— (1997a) "Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic". URL: https://people.eecs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF (visited on ~2024.3.10).

— (1997b) "The Baleful Effect of Computer Languages and Benchmarks upon Applied Mathematics, Physics and Chemistry (John von Neumann Lecture)". URL: https://people.eecs.berkeley.edu/~wkahan/SIAMjvnl.pdf (visited on ~2024.3.10).

Kharva, Paresh (2020) "TensorFloat-32 in the A100 GPU Accelerates AI Training, HPC up to 20×". URL: https:

//blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/ (visited on ~2024.3.10).

Koenig, Jack (2018). *A Hardware Accelerator for Computing an Exact Dot Product*. Tech. rep. UCB/EECS-2018-51. University of California, Berkeley. URL: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-51.pdf (visited on ~2024.3.10).

MacDonald, Tom (1991). "C for Numerical Computing." In: *Journal of Supercomputing* 5.1, pp. 31–48. DOI: 10.1007/BF00155856.

~mastyr-bottec, Matthew Levan (2023) "Epoch System". URL: https://roadmap.urbit.org/project/epoch-system (visited on ~2024.3.10).

Peters, Christoph (2021). "FMA: A faster, more accurate instruction." In: *Moments in Graphics*. URL: https://momentsingraphics.de/FMA.html (visited on ~2024.3.10).

Posit Working Group (2022). *Standard for Posit™ Arithmetic*. Tech. rep. Posit Working Group. URL: https://posithub.org/docs/posit_standard-2.pdf (visited on ~2024.3.10).

Risse, Thomas (2016). "It's Time for Unums—an Alternative to IEEE 754 Floats and Doubles." In: *Proceedings of the Fifth International Conference on Signal & Image Processing (SIP-2016)*, pp. 50–51.

Sidwell, Nathan and Joseph Myers (2006). *Improving Software Floating Point Support*. Tech. rep. CodeSourcery. URL: https://hashingit.com/elements/research-resources/2006-01-improving_software_floating_point_support.pdf (visited on ~2023.3.9).

Steele Jr., Guy L. and Jon L. White (1991). "How to print floating-point numbers accurately." In: *Proceedings of the 1990 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '90)*, pp. 372–389. URL: http://kurtstephens.com/files/p372-steele.pdf (visited on ~2024.3.10).

Thall, Andrew (2007). *Extended-Precision Floating-Point Numbers for GPU Computation*. Tech. rep. CIM-007-01. The

University of North Carolina at Chapel Hill. URL: http://andrewthall.org/papers/df64_qf128.pdf (visited on ~2024.3.10).

Tlon Corporation (2020) "The Last Network Breach". URL: https://roadmap.urbit.org/project/last-network-breach (visited on ~2024.3.10).

Urbit Foundation (2023) "Urbit". URL: https://github.com/urbit/urbit (visited on ~2024.3.10).

van Dam, Laurens et al. (2019). "An Accelerator for Posit Arithmetic Targeting Posit Level 1 BLAS Routines and Pair-HMM." In: *Proceedings of the Conference for Next Generation Arithmetic*, pp. 1–10.

Wang, Shibo and Pankaj Kanwar (2019) "BFloat16: The secret to high performance on Cloud TPUs". URL: https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus (visited on ~2024.3.10).

~wicdev-wisryt, Philip C. Monk (2020a) "Urbit Precepts". URL: https://urbit.org/blog/precepts (visited on ~2024.3.10).

— (2020b) "Urbit Precepts (Discussion)". URL: https://urbit.org/blog/precepts-discussion (visited on ~2024.3.10).

The urwasm WebAssembly Interpreter Suite on Urbit

K. Afonin ~dozreg-toplud

Abstract

WebAssembly is a low-level language for a portable virtual machine. Wasm is designed to be a compilation target for a variety of programming languages and its design is hardware independent and relatively simple, making its support ubiquitous in modern browsers. Its simple design made it a perfect first candidate for a first emulator of an conventional computational system on a novel functional computer: Urbit. In this paper I discuss the current state of the urwasm project and some technical details, as well as describe the strategy to jet the interpreter of a state machine in a functional environment.

Contents

Urbit Systems Technical Journal I:1 (2024): 133–150.

Address author correspondence to ~dozreg-toplud.

1 Introduction

> … Martian code is so perfect that it cannot be contaminated, even in this sticky environment. The general structure of cross-planet computation is that Earth always calls Mars; Mars never calls Earth. The latter would be quite impossible, since Earth code is non-Maxwellian. There is only one way for a Maxwellian computer to run non-Maxwellian code: in a Maxwellian emulator. Any direct invocation is an implementation error by definition. Thus, Mars remains completely pure and Martian, even while glued firmly to Earth. (`~sorreg-namtyv`, 2010)

The Urbit computer is specified as a Nock interpreter and a storage of an event log, represented as a list of nouns (`~sorreg-namtyv` et al., 2016). Nock is a Lisp-like typeless functional language with a tiny specification (`~sorreg-namtyv`, 2013), which defines a function `nock(sub, fol)`, where subject `sub` and formula `fol` are both nouns: binary trees of unsigned integers with an arbitrary length, or "S-expressions without an S". The state of the Urbit computer is thus defined as `nock(epic, [2 [0 3] [0 2]])`, where `epic` is the event log.

This environment may seem quite limiting at first glance, but by feeding proper events to this computer (via the Hoon-

to-Nock compiler, the Arvo OS written in Hoon, etc.) we can create a personal server used by thousands of people today, which can host Web applications and communicate with other Urbit computers.

If we desire to run conventional ("Earth") programs on Urbit, we would have to build an interpreter in Nock that would evaluate the imported code within Urbit. WebAssembly (Wasm) is an excellent candidate language for having an interpreter in Nock:

1. Wasm is a low-level language, supported by many languages as a compilation target;
2. Wasm's specification is small and hardware-independent when compared to other alternatives;
3. Wasm is widely used in browsers as well as in other contexts, creating an incentive for developers to add Wasm support in their projects.

For these reasons, we set out to build a Wasm interpreter native to Urbit (`Quodss/urwasm`). As of writing, the project consists of the following components:

1. A parser from `wasm` binary format to a Nock noun;
2. An interpreter, which fully satisfies the Wasm Core Specification from the latest published version (Rossberg and the WebAssembly Working Group, 2024);
3. A parser from `wat` text format to a Nock noun, implemented by calling a Wasm module with a `wat2wasm` parser and then feeding the result to the binary parser. This parser serves as a testing method and is very slow with the interpreter being unjetted.

This article first describes the implementation of the interpreter in `urwasm`. It then lays out a strategy for efficiently jetting the interpreter. It closes by discussing the need for an interpreter of Language for Invocation of Assembly (Lia) to encapsulate the Wasm interpreter. A sketch of the specification of Lia language and `++lia` interpreter is provided.

2 Urbit-Native Wasm Interpreter Implementation

The strategy for interpreting Wasm expressions was to model each instruction as a function

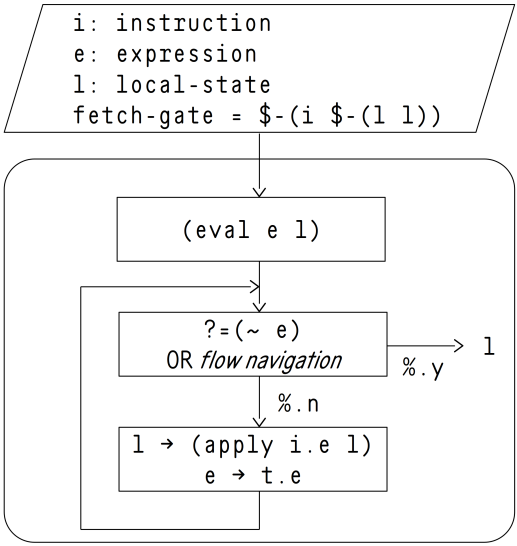
$$\texttt{local-state} \rightarrow \texttt{local-state},$$

where `local-state` is a noun which describes the state of the interpreter at any point during the computation: stack and local values, linear memory, and so forth. An expression is a list of instructions, and can be modeled as a composition of all instructions in the list, which also makes it a function from `local-state` to `local-state`. Most of the Wasm instructions were implemented as Hoon gates with that type, while five instructions (`call`, `call_indirect`, `block`, `loop`, and `eval`) were treated directly in the evaluation loop.

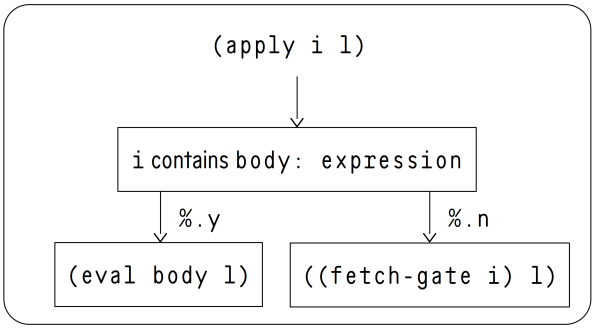
The evaluation loop consists mainly of two functions: `++eval` and `++apply`. `++eval` takes an expression and a `local-state` to produce a `local-state`. It pops an instruction from the expression and applies it to the local state with `++apply`, stopping if it reaches the end of the list or if an instruction triggered execution flow navigation. For example, the `return` instruction causes `++eval` to stop applying the rest of the instructions and return the local state as is, with the branching coordinate contained in the local state.

The `++apply` gate takes an instruction and `local-state`, returning `local-state`. While for most instructions it finds an appropriate gate `$-(local-state local-state)` and applies it to the given local state, in the case of the five instructions listed above it instead calls `++eval` on the body of those instructions. Thus stack frames are divided not with a label value, as in the formal specification of Wasm, but with the depth of mutual recursion of `++eval` (Figure 1a) and `++apply` (Figure 1b).

The gates above are internal; the outside caller will typically interact via two other gates, `++prep` and `++invoke`. `++prep` instantiates the module: it loads the data segments into the linear memory, instantiates global values, and runs



(a) `++eval` data flow.



(b) `++apply` data flow. Some details like entering/exiting frames are elided.

Figure 1: Data flow in the `urwasm` interpreter.

a start function if specified. The product of this gate is either `global-state`, which describes the state of the Wasm module in between function invocations, or a block on an unresolved import, or a crash or "trap" in WebAssembly terms.

`++invoke` takes a name of an exported function, input values with type annotation and `global-state`. It performs a typecheck on the supplied values and calls an appropriate function. The result (as with `++prep`) either succeeds with output values and updated state, or blocks/traps.

3 The WebAssembly VM and Determinism

The state of an Urbit computer, as already mentioned above, is a pure function of its stream of events with the lifecycle function defined in Nock. No information other than the event log may impact the computation of this state, including the hardware on which the interpreter runs or the implementation details of the interpreter. For a Wasm interpreter in Nock to be practical it needs to be paired with some fast implementation of the Wasm interpreter in another language, and the equivalence of both algorithms must to be confirmed. In this way, the Urbit computer can be a practical personal server without violating its simple definition: the code of functions in Nock must be sufficient for an Urbit programmer to reason about the system.

This demand of equivalence, however, poses some difficulties. Some instructions of Wasm are defined non-deterministically. For example, the `memory.grow` instruction can return -1 and not increase the length of the linear memory even if the maximum size of the memory described in the module file is not exceeded. The choice is left to the embedder, which can opt to keep the size of the memory buffer based on its resources.

In addition, some numerical operators return a set of values; that is, a Wasm VM could return any value from that set. This typically happens with floating-point operators when they are given a NaN value, with the set being defined by the value being a canonical NaN or not.[1] Some numerical opera-

[1] Cf. `~lagrev-nocfep` (2024), pp. 93–131 in this issue.

tors return an empty set of values. In other words, their behaviour is set to be undefined for certain inputs, and those operators are described as partial. This can happen, for example, with the `idiv_N` operator when the second operand is equal to zero. The behaviour of Wasm VM is undefined in this case.

Therefore, the Wasm interpreter in `urwasm` describes a deterministic Wasm machine, limited to a subset of behaviors:

1. If an empty set of values must be returned, the interpreter "traps": it returns a deterministic error as a result, similar to `++mink`;

2. If a set of values must be returned, then a single result from that set is returned. The choice is particular to each operator, but typically, if the set was a union of sets that depend on multiple input parameters, then the choice was made from the set given by the first parameter. (Refer to `/lib/op-def` for more information.)

3. `memory.grow` and similar instructions always attempt to grow the buffer when the limits in the module file permit.

4 Examples of `urwasm` Programs

Some simple programs are included in the test suite. Most of them involve only numerical operations, while one of them calls a Wasm program compiled from Rust source which flips the order of characters in the string. Evaluation of this function requires multiple Wasm function invocations in order to e.g. allocate memory for the input string, and interaction with the global-state to read from and write to linear memory.

When Rust source is compiled, the compiler produces JavaScript code which takes care of low-level handling of the state of Wasm runtime. In the case of Wasm interpreter in Hoon, JS code was manually translated to Hoon.

In addition, some other programs were run successfully. WebAssembly text format parsing in `urwasm` is implemented by composing the `wat2wasm` parser from the `wat` Rust crate

with a binary parser in Hoon. Another example was presented at the Urbit Assembly in October 2023: an algorithm to simplify debts between a group of people, using the `FlowGraph` Dinic algorithm for maximum flow computation from the `contest_algorithms` Rust crate (`~dozreg-toplud` et al., 2023).

These cited examples highlight the incredible inefficiency of the interpreter: it takes about a second to flip a string with 27 characters, and about a minute to parse a small Wasm binary with compiled `wat2wasm`. While there is most likely room for improvement, the Hoon code serves first and foremost as a formal specification of a deterministic Wasm VM, translated from the mix of pseudocode and mathematical formulas to a tightly-specified language. This specification would then serve for the verification of jet correctness.

5 Jetting

To jet the interpreter, the gate written in Hoon must be paired with a code in C that must be extensionally equivalent to the Hoon code. In that sense function definition in Hoon would act as a formal mathematical specification of what the interpreter returns, while C code would act as the actual implementation of Wasm runtime by arriving to the same conclusion as Hoon code but faster. The purpose of the Hoon specification is then to provide a test bed for verifying correctness of the interpreter, either by testing or formal analysis of both programs via e.g. the K verification framework (cf. `runtimeverification/knock`, `~bithex-topnym` (2023)).

Since Wasm is a portable language for a state machine, each invocation of a function from a Wasm module would either return a successful result with returned values and an updated state, or some flavor of failure (trap or blocking on unresolved external request, e.g. function call of an imported function). This gives us several possible strategies for jetting:

1. Have a bespoke Wasm interpreter in C that operates on nouns, and jet invoke gate.

Listing 1: Rust function to reverse the order of characters in a string using Wasm.

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn process(input: String) -> String {
  let output_string: String = input.chars().rev().
    collect();
  output_string
}
```

2. Use an established Wasm runtime in C, and add serializer/deserializer to the jet, to convert Hoon representation of the module state to a representation in C and vice versa, and jet invoke gate.

3. Use an established Wasm runtime in C, and add serializer/deserializer to the Hoon specification, and use a representation of state close to the one in C as input and output in Hoon invoke gate, and jet invoke gate.

4. Don't jet the `++invoke` gate at all. Instead, have a higher level function that executes a series of operations on a module and doesn't return the entirety of Wasm module state. Hoon specification of this function would use the Wasm interpreter in Hoon, and the jetting code in C would use Wasm runtime in C.

But first, why do we have to have access to the state in the first place?

Even for a simple source code the generated Wasm code might require multiple function invocations to get a desired result. Consider a Rust function that flips the characters in a given string (Listing 1). After compiling this function to Wasm, you would get a module with five exported functions: `process` itself, `add_to_stack_pointer`, `malloc`, `realloc` and `free`. The call of the compiled function in JS would look like:

Listing 2: JavaScript function to reverse the order of characters in a string using Wasm. Compare Listing 1.

```
  export function process(input) {
  let deferred2_0;
  let deferred2_1;
  try {
    const retptr = wasm.
    __wbindgen_add_to_stack_pointer(-16);
    const ptr0 = passStringToWasm0(input, wasm.
    __wbindgen_malloc, wasm.__wbindgen_realloc);
    const len0 = WASM_VECTOR_LEN;
    wasm.process(retptr, ptr0, len0);
    var r0 = getInt32Memory0()[retptr / 4 + 0];
    var r1 = getInt32Memory0()[retptr / 4 + 1];
    deferred2_0 = r0;
    deferred2_1 = r1;
    return getStringFromWasm0(r0, r1);
  } finally {
    wasm.__wbindgen_add_to_stack_pointer(16);
    wasm.__wbindgen_free(deferred2_0, deferred2_1, 1);
  }
}
```

Multiple invocations are necessary, some with the arguments received from outputs of other invoked functions. Having the access to the state of the module is thus crucial for any practical interpreter. In addition, we may have to read data from the state and save it as a noun, in order to perform I/O on that piece of information, since Arvo may only send effects when the computation of a given event has finalized. While jets could in theory perform system calls during their evaluation, doing so is considered a gross violation of jetting mechanism: jets as functions are supposed to be as pure as possible, to imitate their Nock definitions.

Let us review our options.

5.1 Bespoke Wasm interpreter that operates on nouns

Maybe this is the perfect option in the long term, but this option is unfeasible in the short to medium term due to the required development time. In addition, it would require speculative kinds of optimizations added to a Nock interpreter that do not yet exist, such as "data jets", when a noun is represented in a way that allows certain operations faster to be performed faster. (For example, the Nock runtime could have a list of bytes `(list @D)` represented as a byte array in its memory, with some jetted functions configured to operate on that kind of nouns more efficiently.)

5.2 Serialization/deserialization in the jet

The next solution is to add serializer and deserializer functions to the jet of the `++invoke` arm, which would convert the noun representation of the state of the module to a representation legible to the jetting interpreter before evaluating the invoked function, and convert it back into a noun when the computation is finished. However, having to translate the state between two models twice for each function invocation would impose a lot of computational overhead.

5.3 Serialization/deserialization in Hoon

A less obvious variation is to put the de/serializer in the Hoon code of `++invoke`, and have the jetted gate take and return module state in the same representation as in the Wasm interpreter in C. The jet would then operate on the given state directly, without having to translate the state of the interpreter. Here the problem is that the implementation strategy leaks into the formal specification, making Hoon code jet-dependent. Replacement of the jetting Wasm runtime, including in the case of switching from Vere to Ares, would make us have to change the Hoon code, which I find to be antithetical to the Urbit project itself.

In addition, current model of jetting in both Vere and Ares requires that the core produced by the jetted arm is left read-only, without modification, so the output of the jet function cannot overwrite the core's sample. That would require the interpreter to copy the entire state of the module and operate on that, which creates prohibitive overhead for memory-heavy computations, like running a Linux VM.

5.4 Higher level interpreter function

This strategy would involve writing a function that takes a Wasm module, a list of operations to be performed and some other parameters, and then jetting this function. No intermediate state would be returned, saving us from having to convert it between different representations. However, a practical jet implementation would have to cache the intermediate state of the interpreter between Arvo events, otherwise it would have to reevaluate the operations each time the jetted function encounters a block on an unresolved import, or if the caller appended new operations to the list to interact with the module.

6 Lia Interpreter

Let's return to our example with string flipping in Rust. How would our hypothetical higher-level function run the program?

Ignoring imports for now, consider this gate:

```
++  lia
|=  [module=octs actions=(list action)]
^-  $%  [%0 out=(list value)]
        [%2 ~]
    ==
::  (...)
```

This gate accepts a binary file of a Wasm module and a list of actions to be performed, and returns either a list of values in the event of successful resolution, or an error. A value here is either a WebAssembly numerical value or a slice of the linear memory called `octs` in homage to Hoon. To flip a string, we would pass it the module file obtained from the Rust compiler and a list of actions obtained from parsing this code (ignoring `realloc` for simplicity), as shown in Listing 3.

Listing 3: Lia function to reverse the order of characters in a string using Wasm. Compare Listings 1 and 2.

```
# Lia (Language for Invocation of Assembly) scripting
#   language
# "add_to_stack_pointer", "malloc" and "process"
# are functions exported by the wasm module from Rust
# example above
#
string0 = "Hello world".to_octs
retptr = __wbindgen_add_to_stack_pointer(-16)
ptr0 = __wbindgen_malloc(string0.len, 1)
memory.write(string0, ptr0)
process(retptr, ptr0, len0)
i32 r0 = memory.read(retptr, 4)
i32 r1 = memory.read(retptr+4, 4)
return memory.read(r0, r1)
```

This code is written in Language for Invocation of Assembly (Lia for short). We imagine this language to have a very small specification: its only purpose is to describe sequences of actions to be performed with a Wasm module. These include

function invocation and memory reads and writes, as well as variable declarations, `for` loops and conditionals for expressivity.

Listing 3 above is essentially identical to the JS example in Listing 2. Ideally, Lia will be generated by the compiler itself, just like the JS code was also generated, with placeholders for input values, where the caller could insert their parameters.

A jet of `++lia` would have to perform the same operations but with an interpreter in C. However, considering the tiny specification for Lia, we do not foresee implementing the algorithm identically in both Hoon and C to be a challenge.

The Lia interpreter as described has been non-interactive; it takes a module and a list of actions and returns a result. But the WebAssembly module can and should be interactive: we want to invoke a function, perform some I/O based on the result, then invoke a function again etc. In addition, the module might contain imported functions, which we wish to resolve to provide I/O to the Wasm program.

Interacting with an instantiated module in the case of `++invoke` jetting is straightforward: the state of the module is a noun, and to interact with the state we would call various jetted gates that invoke functions and perform I/O on the store. Handling import function calls is also straightforward: `++invoke` might return a blocked result with the name of the imported function, which is then resolved in the embedding context, modifying state of the module if necessary. How would the same be achieved in a stateless fashion?

In Listing 4, we have added details to the block or trap. Now the interpreter can return a block with `%1`, asking for an external reference to be resolved and the result of the resolution to be appended to shop. The interpreter now takes a list of list of values `shop`, which represents the resolved Lia imports; `ext-func`, which provides definitions for imported functions for a given Wasm module; and `diff`, whose contents are appended to an appropriate field.

Notice the difference between imports in Lia and in Wasm: the namespace of the latter is defined with a pair of strings in compliance with the formal specification, and the input parameters of `++lia` contain definitions of the imported functions for

Listing 4: Upgraded ++lia arm.

```
++  lia
  |=  $:  module=octs
          actions=(list action)
          shop=(list (list value))
          ext-func=(map (pair cord cord) (list action)
    )
          diff=(each (list action) (list value))
      ==
  ^-  $%  [%0 out=(list value)]
          [%1 name=path args=(list value)]
          [%2 ~]
      ==
  =>
    ?:  ?=(%.y -.diff)
      .(actions (weld actions p.diff))
    .(shop (snoc shop p.diff))
  ::  (...)
```

the Wasm module. Lia imports, on the other hand, are bound to a path namespace and are external to ++lia.

In a trivial case the list of actions of a function definition in ext-func would contain a single call to a function which is external to Lia and is named with path, not (pair cord cord) like Wasm import functions. In a nontrivial case the list of actions could contain multiple actions to be performed, e.g. a memory read followed by calling a Lia import function which takes octs. The purpose here is for the import calls to be able to surface not only Wasm values but Lia values like octs, giving us the richness of import calls that we could have with ++invoke jetting model without exposing the entirety of the module's state.

6.1 Caching of store

Since Wasm execution is deterministic, then the state of a module, whether in Nock or in the jetting environment, is referen-

tially transparent with regards to the input parameters of Lia. Here `diff` appears to be semantically useless, since instead of placing changes there we could have placed them directly into a proper field of the sample.

But `diff` is necessary for efficient computations: each time `++lia` jet computes something with a Wasm runtime, it will save a cache of Lia interpreter in the jetting environment tagged with a (hash of a) noun `[module actions shop ext-func]`. If `++lia` is computed later with the same first four arguments, then the jet would first look for a cache, and if it finds one, only compute the diff, either injecting results of Lia import resolution into a suspended jetting Lia interpreter, or performing a list of appended actions. Failing to find a cache, jet of `++lia` would do the same thing as `++lia` in Hoon: append the `diff` to an appropriate field and run the whole thing.

At the price of having to bother with cache reclamation and perhaps slightly longer event log replays we get the possibility of running Wasm runtime almost full speed, without having to de/serialize nouns to structs and vice versa or to copy the state of the Wasm module for every interaction with it. In Nock the state of the Wasm module is represented lazily with the first four parameters of `++lia`, while the actual state on which we perform computations exists as a cache.

A typical interaction with `++lia` would look like this:

1. Call `++lia` for the first time, with an empty diff.
2. Lia returns one of:
    (a) success `%0`;
    (b) block `%1`; or
    (c) failure `%2`, which is resolved outside of Lia.

The resolution result is placed into `diff`, and `++lia` is called again. Next time that same result must be placed directly to shop to get the right cache and avoid recalculating from scratch.

If the result is success, new actions can be put in diff and `++lia` can be called again to continue interacting with the

module. After that these new actions must be placed directly to actions on subsequent calls of `++lia` to get the right cache.

As for the cache reclamation, `++lia` could include another input parameter hint ignored in Hoon code, that would serve as a hint to the jet on how to handle the cache, e.g. for how long to keep it in the memory. Dojo generators and Gall agents, for example, would likely benefit from different cache reclamation strategies.

7 Next Steps

What is the roadmap forward for the `urwasm` project? The first priority is to add unit tests and module validation in Hoon. This will allow us to verify the correctness of the interpreter and to ensure that the jetting process is successful. The next stage will consist of enabling caching in the jet of `++lia` and adding a parser from text to `(list action)`. This will enable us to construct a Lia interpreter in Hoon, jetted with the Lia interpreter in C/Rust. This facilitates a default Gall agent interface to interact with Lia and enable building Gall apps in conventional languages. Finally, we will add Wasi support and Lia generation to common Wasm compilers in the same way they generate JS wrappers for the modules.

8 Conclusion

This paper presents a novel method of modeling state machines in Nock and jetting the associated modeling algorithm. In this method the state is represented implicitly through a list of actions performed on the initialized state of the machine, and the state of the jetting emulator of the state machine is kept as a cache identified with the list of actions. This gives the interpreter full computational speed due to lack of overhead caused by translating the state from noun representation to the jetting representation and back and by the fact that the jetting function is free to edit the state in place, as long as it updates the naming tag of the cache. ⊠

References

~bithex-topnym, Rikard Hjort (2023) "KNock: Nock Semantics in K". URL: https://github.com/runtimeverification/knock (visited on ~2024.3.7).

~dozreg-toplud, K. Afonin et al. (2023) "UWasm (sic) presentation ~dozreg-toplud". URL: https://www.youtube.com/watch?v=h5V_gX33RiM (visited on ~2024.3.7).

~lagrev-nocfep, N. E. Davis (2024). "The Desert of the Reals: Floating-Point Arithmetic on Deterministic Systems." In: *Urbit Systems Technical Journal* 1.1, pp. 93–131.

Rossberg, Andreas and the WebAssembly Working Group (2024). *WebAssembly Core Specification (W3C Working Draft)*. Specification. World Wide Web Consortium (W3C).

~sorreg-namtyv, Curtis Yarvin (2010) "Urbit: functional programming from scratch". URL: http://moronlab.blogspot.com/2010/01/urbit-functional-programming-from.html (visited on ~2024.1.25).

— (2013) "Nock 4K". URL: https://docs.urbit.org/language/nock/reference/definition (visited on ~2024.2.20).

~sorreg-namtyv, Curtis Yarvin et al. (2016). *Urbit: A Solid-State Interpreter*. Whitepaper. Tlon Corporation.