Ford Fusion

Ted Blackman ~rovnys-ricfer
Urbit Foundation

Abstract

Ford Fusion was an overhaul of Urbit's over-the-air upgrade process and a rewrite of its build system. The new update system corrected a few long-standing bugs with the previous one, and the new build system is simpler, smaller (by around 5,000 lines), and easier to manage. This historical report was published on the Urbit Blog as a capitulation of the project, which successfully revamped the Hoon code build system. It is lightly annotated to update the minor technical changes that have occurred since the original publication; this is intended as a living document to some extent. Ford Fusion remains the state of the art for building Urbit software as of writing.

Contents

Urbit Systems Technical Journal I:2 (2024): 89–104.

Address author correspondence to ~rovnys-ricfer.

This Ford Fusion description was released by ~rovnys-ricfer *on* ~2020.7.14.

1 Overview and Rationale

Ford Fusion was an overhaul of Urbit's over-the-air upgrade process and a rewrite of its build system. The new update system corrects a few long-standing bugs with the previous one, and the new build system is simpler, smaller (by around 5,000 lines), and easier to manage.

Since deployment of Ford Fusion to the livenet in late June, over-the-air updates (OTAs) have been much smoother. Before Ford Fusion, it was common for an OTA to take several hours, use too much memory, and leave ships in inconsistent states. After Ford Fusion, multiple OTAs have been pushed out, including kernelspace changes, and most users didn't even notice.

Urbit has always been able to update itself OTA, but this process has often been rocky. Updating an operating system kernel on-the-fly is a difficult problem in general, like performing heart replacement surgery on yourself while running a marathon. Code that allows Linux to update its kernel in this way became a startup called Ksplice, won multiple awards, and sold to Oracle. Even that, as impressive as it is, and as brilliant as its programmers are, can only perform certain limited kinds of patches to the kernel.

Urbit isn't exactly a traditional operating system, so the comparison is somewhat unfair, but the purpose of better ar-

chitecture is to create unfair comparisons. In this case, because the Nock layer is frozen, upgrading everything above that layer is easier. Upgrades are also facilitated by pure-functional semantics, transactional event processing, a type system oriented toward concrete data, and orthogonal persistence. These features make it feasible for Urbit to upgrade itself in the general case, not just some special cases.

Ford Fusion has fixed the major upgrade issues of the past by guaranteeing three properties that in retrospect are obvious requirements, but, like much of Urbit, took many years and rewrites to identify as such:

- Atomic: the update should complete or fail in one transaction. If it fails, the system shouldn't get stuck.
- Self-contained: there must be no implicit dependencies or hysteresis (dependence on previous system states) when building the new software from source.
- Ordered: updates must be monotonically sequenced from the system's lowest layer to highest.

Let's walk through them one by one.

1.1 Atomicity

In previous versions of Urbit, updates failed atomicity by deferring parts of the update to later events, which are separate transactions that can fail independently. Generally, each deferral causes an exponential increase in the number of failure states that needed to be handled.

We've learned that asynchronicity is an entropic state. A system will tend toward more asynchronicity over time unless effort is put into keeping it synchronous. As developer Jonathan Blow has noted (Blow (2019), 42m27s), the language server protocol has turned every editor plugin into a distributed system, since now it has to communicate asynchronously with the main editor process.

Consider an update system that took multiple Arvo events to complete. An ad-hoc higher-level transaction system would

need to be built to roll back the effects of the first few events in case of failure. It's important for various parts of the system to be able to emit effects on upgrade; since those effects would need to be rolled back if a later event in this upgrade fails, the system would need to maintain a queue of those events and only apply them once all the other upgrade events have completed.

Note that the asynchronicity has now spread. Some effects that would normally be guaranteed to be processed synchronously might now be asynchronous. Entropy has begun to take hold, chipping away at the set of invariants the system is capable of guaranteeing.

This observation is not purely theoretical. False modularity was the cause of internal asynchronicity in Clay where it had to wait for responses in a complex dance with Ford, which was another vane (Arvo kernel module); moving Ford into Clay allowed function calls that were synchronous from Clay's perspective, which allowed further simplifications, culminating in about a twenty percent reduction of source code size of the Arvo kernelspace.

Steve Yegge's "platform rant" (Yegge, 2011) describes a Bezosian edict prohibiting synchronous communication among modules through direct linking. This can be seen as an acknowledgment of the difficulty Amazon was going to have when it needed to turn internal services external. If your software needs to run in hell, build it that way from the start.

An Urbit ship is not an enterprise SaaS product and does not need to run in this hell; it exists for just one person, with natural pressure pushing it in the opposite direction from Amazon's web services. Instead of needing a ship's state and functionality sharded into microservices strewn across multiple clusters, an Urbit instance is easiest to manage as a single server with all its state unified into one data structure and its event log stored as one totally ordered sequence of state updates – the antithesis of a distributed system.

1.2 Self-Containment

Before Ford Fusion, each commit to the Clay filesystem validated its files using filetypes (called "marks") defined by files in the previous commit (see `~lagrev-nocfep` (2024), pp. 33–48 in this issue). This could cause bugs if the filetype definitions had changed in a backward-incompatible manner. It also meant a commit could not add both a new filetype and new files of that type; instead, you needed two commits: one to define the filetypes, and a second to add files of that new type. More theoretically, it caused history-dependence. The validated contents of files in a commit could vary based on the history of commits that led to this one.

Another way source code failed to be self-contained was that it had access to symmetry-breaking information at build-time, namely ship, desk (Urbit's answer to a GIT branch), and (faked) date. A build recipe should be able to shared, cached, and rerun without dependence on local conditions, so user code now no longer learns its ship, desk, or date until runtime.

Source code also had build-time access to Urbit's immutable global namespace, called the "scry namespace", which the kernel makes available as an implicit argument to userspace Hoon code. The scry namespace is immutable and referentially transparent, i.e., a request must always yield the same result for all time, but if an agent asks the kernel for a resource that's from the future, hosted on another ship, or to which that agent doesn't have permission, the kernel will deny the request.

If the kernel denies a scry request that user code made during the build process, the build system has no choice but to treat it as a nondeterministic error. Nondeterministic errors can never be fully eradicated, if for no other reason than that the user always has the option to defenestrate the machine—there's nothing deterministic about that. But we try to minimize them, and especially to minimize uncertainty as to under what conditions they might occur.

No build should be killed by the absence of files outside the desk, so as of this update, user code can no longer scry at build time. Once built, userspace programs can scry if run

in a context with a scry handler; a Gall agent's runtime scry requests still work just fine.

1.3 Order

The final kind of failure fixed by Ford Fusion was the lack of ordered layering during a software update. The most common form of this failure was that old Ford had a tendency to try to build userspace code using the previous version's standard library. This didn't work too well, unsurprisingly.

Emerging from this underworld required making a number of changes to the Arvo kernel, Clay, Gall, and the procedure for kernel updates. To avoid turning into a pillar of salt, I'll skip the details of how the old system worked and instead describe the new update procedure.

2 How Updates Work Now

These are the layers of the stack that update themselves on the fly, from lowest to highest:

1. `/sys/hoon`: the Hoon language definition and compiler
2. `/sys/lull`: common type definitions for Arvo, its vanes, and userspace
3. `/sys/arvo`: the Arvo kernel proper and related type definitions
4. `/sys/zuse`: the standard library
5. Vanes: Arvo kernel modules, including Clay itself
6. Userspace: apps, marks, ancillary source code like libraries, and user data

An update to one layer necessitates a reload of all layers above it; e.g., a change to Zuse should trigger updates to the vanes and userspace. Conversely, an update to a higher layer should not cause a spurious reload of lower layers, which

should not be affected by the change; for example, an update to just userspace should not cause any reloads of system code.

Clay is responsible for enforcing the layering of updates. An update to a module is triggered when an attempt is made to commit a change to Clay that affects one or more files needed to build the module. For example, if the `foo` agent's source, defined in `/app/foo/hoon`, imports the `bar` library from `/lib/bar/hoon`, then a modification to `/lib/bar/hoon` triggers an update to the `foo` agent. All vanes and userspace files depend on Zuse, which depends on the Arvo and Hoon sources, so a change to the Hoon, Arvo, or Zuse sources will trigger updates to all vanes and userspace files.

When asked to perform a commit, Clay determines which layers need to be updated based on which files have changed and which modules depend on those files. For now, all running programs load their source from the `%base` desk, so only changes to `%base` trigger stateful updates. Files in other desks can be built, but not installed into the system. This might be relaxed in the future.

The process of updating varies by layer. The Hoon and Zuse layers are stateless, so their newly rebuilt cores (Nock executables) must be stored (somewhere in the system's Nock tree, in memory; remember, Urbit is a single-level store), but they have no state that would need to be migrated. The Arvo kernel, vanes, and userspace agents are all live, stateful programs, so in order to update one of those, the system must extract the state from the old program, pass that data into the newly built program, then discard the old program and store the new one. Arvo and agent state injection routines can emit effects, but vane updates cannot.

To work around this limitation, Gall has a two-phase update process. First it enters a dormant "pupal" phase that stores not running agent cores, but only the agent states that the old Gall had extracted from its agents. When Clay notifies Gall that its agents have been rebuilt, Gall "molts" back into normal functionality by loading the agent cores from Clay and then running their `+on-load` routines to inject the old state.[1]

[1] Including agent state upgrade handling.

If there's a change to files in `/sys` on the `%base` desk, Clay asks Arvo to update kernelspace. Clay sends a sequence of moves (effects) to Arvo to ask Arvo to perform any necessary updates to Hoon, the kernel, Zuse, and vanes. This sequence is terminated by an extra move back to Clay itself, which will be received by the updated version of Clay after migrating its state. The rebuilt Clay can then use the newly rebuilt version of Zuse to rebuild userspace and notify clients of the update. One client is Gall, which molts when Clay notifies it.

Clay triggers updates, but the Arvo kernel is responsible for performing updates to all kernelspace layers, and Gall is responsible for updating userspace agents. Agents are stored in Gall's state, but all other layers are stored directly in the Arvo core's state, so the Arvo kernel contains the routines that reload Hoon, the kernel itself, Zuse, and the vanes.

The Arvo kernel reloads itself by compiling the future version of itself, then calling the new core's `+load` routine with the relevant parts of the old state. The state passed to the new Arvo now includes not just the vane cores and their states, but also the Arvo "duct" call stack, which maintains a stack of queues of moves to be passed from one vane to another, and a list of effects to emit to Unix at the end of the current Arvo event. If needed, Arvo could migrate the outstanding moves themselves – if, say, the duct datatype changes.

Passing the Arvo call stack state to new Arvo allows a kernel update to happen in the middle of a more complex event without disturbing other sequences of processing steps happening concurrently in the vanes.

This entire update process happens in one Arvo event and doesn't break event-dispatching semantics. This not only provides atomic rollback, but allows the update to be combined with other actions into a larger transaction – for example, to stage complex changes, user code could trigger two kernel updates in a row, both in the same event.

Note that this is the opposite situation from the entropically leaking asynchronicity described earlier. Now the guarantees don't deteriorate; they can be composed into stronger guarantees.

3 How Clay Validates a Desk

A desk is Urbit's answer to a Git repository. It's almost identical, except all files are typed and validated, and whenever a commit becomes the equivalent of Git's `HEAD`, it's assigned a semantically meaningful revision number, and all files are typed and validated.

If Clay has been asked to perform a commit, it needs to validate all the files in this desk and notify all subscribers to live queries of this desk's data. Gall, for example, maintains live queries on builds of its live agents. Validation uses the Ford build system, which as of this update is no longer a standalone vane but a core within Clay.

A Clay commit, like a Git commit, is specified as the current value of all its changed files (and, separately, references its parent commits by hash), not as the diff from a parent commit. Unlike Git, Clay is typed, and every file must be validated according to its "mark". A mark is named like a file extension, e.g., `%txt`, `%png`, or `%noun`, and Clay maintains a mapping from that name to behaviors of values of that type under various operations. The last segment of any Clay path specifies the mark to use for operations on that file, including validation.

Mark operations include conversion to and from other marks (such as converting `%json` to `%txt`), revision control operations (diff, patch, and merge), and validating an untyped noun. Operations for mark `%foo-bar` are defined by a core built using the source code at `/mar/foo-bar/hoon`, or if that doesn't exist, at `/mar/foo/bar/hoon`.

Consider a file at `/web/foo/json`. In order to validate this file, Clay must load the mark definition core and use its validation routine to ensure the untyped value of `/web/foo/json` is in fact valid JSON. To obtain this core, Clay must build the file at `/mar/json/hoon` from source and then process the resulting raw mark core using some mild metaprogramming to get a standard interface core for dealing with marks, called a `$dais`, whose type is defined in Zuse.

Since building a source file only makes sense if the file has been validated as a `%hoon` file, but mark definitions themselves

must be built from source, there's a logical dependency cycle – who validates the validators? To break this cycle, Clay hard-codes the validation of `%hoon` files. This allows mark definitions to be built from source, and in fact any file can depend on any other file of any mark as long as there are no cycles. As of Ford Fusion, Ford performs a cycle check to ensure acyclicity.

Since building a file is a pure function, Clay memoizes the results of all builds, including builds of marks, mark conversions, and Hoon source files. These memoization results are stored along with the desk and are used by later revisions of that desk. Future work should allow merge commits to pull memoized builds from all parents, but for now only the previous revision of the current desk is used. This is a major simplification of previous Ford architectures, which maintained much more complex caches with less clear eviction semantics. Now on every commit, we just throw away any unused memoized builds from the previous revision's Ford cache.

Once Clay has validated every file in this new revision of a desk, it constructs and sends updates to any subscriptions that other vanes or agents have requested. More Ford builds may be run to fulfill these requests, including builds for any running agents whose dependencies changed in this commit.

When Gall receives a newly rebuilt agent from Clay, it calls the gate produced by the `+on-load` arm of the new agent with the state extracted from the old agent. If there is a crash in any `+on-load` calls or in the handling of any effects they emit (which can include more agent activations), then the whole event crashes, canceling the commit. This effectively gives any agent the ability to abort a commit by crashing.

It is a bit counterintuitive that an app reload failure could prevent a kernel update. The reason is that we don't want the system to update itself into a broken state. An Urbit can be rendered practically unusable by the presence of broken agents, even if the kernel hasn't lost integrity, so it's kinder to the user not to break their agents by installing an incompatible kernel update. This also puts virtuous pressure on kernel developers not to "break userspace", the importance of which has been insisted on for decades by Linus Torvalds, among others.

If an agent does crash a commit event that included a kernel update, the attempted commit is now trivially rolled back, and the system can deliver an error message to the user. This does not leave the system in an inconsistent or stuck state, so the user could modify the failing agent and try the kernel update again later. Supporting better workflows for keeping third-party agents up-to-date will be an important aspect of Urbit's upcoming software distribution work.

4 Ford Build Semantics

4.1 The Three Types of Ford Builds: Files, Marks, and Casts

The Ford build semantics have been simplified. There are now three kinds of builds that Ford can perform: files, marks, and casts, all of which happen synchronously as function calls inside Clay and are available (without memoization) as scry interfaces.

4.1.1 File Builds

A file build takes in a filepath containing Ford runes and Hoon source, runs the Ford runes to perform imports, and then compiles the source, producing a `$vase`, a noun tagged with its Hoon type.

Clay exposes file builds into the scry namespace with `%ca`: as an example, `.^(vase %ca /~zod/base/3/lib/sole/hoon)` will build the `sole` library.

4.1.2 Mark Builds

A mark build produces a `$dais` mark-interface core. It first performs a file build on the Hoon file in `/mar` that defines the mark core, then it does some metaprogramming to make the operations more convenient to use. If the raw mark core delegated revision control operations to another mark core, the mark build will also load the delegate mark core and resolve the result into the `$dais`.

Clay exposes mark builds into the scry namespace with `%cb`: as an example, `.^(dais:clay %cb /~zod/base/3/mar/foo/hoon)` builds a `$dais` for the `%foo` mark.

4.1.3 Cast Builds

A cast build produces a `$tube`: a gate that takes a value of one mark as input and converts it to a valid value of another mark or crashes. To convert from mark `%foo` to mark `%bar`, Clay tries the following operations, in order:

1. direct grow from ‘
2. direct grab from ‘
3. indirect jump from ‘
4. indirect grab from ‘

The `%foo` mark can “grow to” `%bar` by providing an arm in its `+grow` core named `+bar`. `%bar` can convert from `%foo` using a `+foo` arm in its `+grab` core. `%foo` can also chain a conversion through an intermediary using an arm in its `+jump` core, and `%bar` can specify an “indirect grab” by having a `+grab` arm produce a delegate mark instead of directly defining a conversion gate.

Clay exposes cast builds into the scry namespace with `%cc`: as an example, `.^(tube:clay %cc /~zod/base/3/foo/bar)` builds a `$tube` conversion gate from `%foo` to `%bar`.

4.2 Ford Runes

There are now only seven Ford runes. A file can contain zero, one, or many of each, but each Ford expression can only be one line, and they must be in the standard order of `/-`s, `/+`s, `/=`s, and then `/*`s.

```
/-  foo, *bar, baz=qux
```

The `/-` rune imports a structures file from `/sur`. You can import it as just `foo`, in which case the build result of that file

(usually a core with mold definitions) will be pinned into the compilation subject with the face `foo`. If you prefix it with a `*` as in `*bar`, the result will be pinned into the subject with no face; if the structures file compiled to a core, this exposes all the arms into the namespace of the compilation subject. Finally, if you import it as `baz=qux`, the `baz` face will be applied instead of `qux`. This is similar to "import as" in other languages.

```
/+  foo, *bar, baz=qux
```

The `/+` rune imports a library file from `/lib`. Aside from the different source folder, the syntax and semantics are the same as for `/-`.

```
/=  clay-raw  /sys/vane/clay
```

The `/=` rune imports the result of building a Hoon file from a user-specified path (the second argument), wrapping it in a face specified by the first argument. The final `/hoon` at the end of the path must be omitted. This is mostly useful for importing a file for testing. The file at the specified path will be built as a normal userspace Hoon file; i.e., its compilation subject will be Zuse augmented with the results of any Ford runes it has at the top of the file.

```
/*  hello-gen  %hoon  /gen/hello/hoon
```

The `/*` rune imports the contents of a file in the desk, specified as the third argument with the full path including the trailing mark, converted to the mark specified by the second argument, and pinned into the compilation subject wrapped in the face specified by the first argument. This can be used to import static data at build-time, such as a data file, a media file, or, in the case of this example, a Hoon file as source text rather than already built.

A valid userspace Hoon file must contain a nonempty list of `hoons` (Hoon source expressions) below the Ford runes, separated by gap (more than one space, or at least one newline). The system wraps this list of `hoons` in a `=~` expression so that the result of the previous `hoon` is used as the subject of the next `hoon`. The result of the Ford runes is used as the compilation

subject for this `=~ hoon`; informally, the shape of the compilation subject can be thought of as:

```
:*  fastar-2  fastar-1
    fastis-2  fastis-1
    faslus-2  faslus-1
    fashep-2  fashep-1
    <zuse>
==
```

```
/$  some-face  %from-mark  %to-mark
```

The `/$` rune imports a mark conversion gate between two types. These are marks on the same desk as the file.

```
/~  some-face  some-type  /some/directory
```

The `/~` rune imports, builds, evaluates, and pins the results of many `hoon` files in a directory. Each Hoon file in the specified directory will be built and evaluated. The result of evaluating each file will be added to a `++map` and pinned with the specified face `some-face`. The keys of the map will be the name of each file, and the values of the map will be the result of evaluating each file and casting its result to the type specified `some-type`.

All of the `hoon` files in the specified directory, when evaluated, must produce data of a type that nests under the type specified `some-type`. File with a mark other than `%hoon` will be ignored.

```
/%  some-face  %some-mark
```

The `/%` rune imports a mark definition from the `/mar` directory. The mark definition will be built and pinned with the specified face `some-face`.

5 Future Work

Urbit still needs to make better use of desks other than `%base` and the development process should be adjusted given the

tighter coupling between source code and kernel and tighter criteria for accepting an update.

This work also hopefully provides a good foundation of a package management and software distribution system for Urbit. As ~wicdev-wisryt has said, a user should be able to run |install ~norsyr-torryn %canvas to load and build remote source. No one should experience dependency hell on Urbit, but we're not there yet.

At least now, building a desk has no dependencies, other than a Ford with a compatible Hoon compiler. No decisions have been made on this yet, but Ford might get moved to inside the desk, possibly by making Zuse callable. This could allow a desk to expose a Nock interface in addition to a typed Hoon interface, which could even let a desk be used as a "pill" bootloader.

6 Conclusion

~littel-ponnys and I (~rovnys-ricfer) spent most of 2018 rewriting Ford with the intent of improving its performance. Compared to its predecessor, its result was better in some ways but worse in others. The caching system was labyrinthine and poorly factored, making the system difficult to debug or prove correct, even informally. Some things were faster, but the caching and dependency tracking were actually complex enough that a number of common operations, like mark conversion, were too slow.

In early 2020, ~master-morzod suggested moving Ford into Clay to reduce asynchronicity. It seemed absurd at first, but at some point I realized I could combine that idea with a simpler build-caching scheme and self-contained desk builds, and ~wicdev-wisryt realized he could use that to further simplify Clay's commit and merge code, which he did as part of this project.

The first time I rewrote Ford, it took me six months, with help from ~littel-ponnys, and it weighed in at 6,000 lines of code. The second time, in late 2018, took a few weeks. The third time, in January 2020, took a week. I wrote +ford in Ford

Fusion in one long day, and it's about 500 lines of synchronous, functional code.

It has taken me two or three years to understand this problem as well as I do, and I expect there are parts of it I still don't understand. The code itself isn't the issue; it's finding the right answer to ontological and teleological questions. What *is* Ford? What will it be in a hundred years? I'm confident Ford Fusion is more similar than its predecessor to the Ford of 2120, because it's smaller, more functional, and easier to understand and administer.

As an engineering discipline and organizational practice, working on a system intended to be frozen yields surprising simplifications like this every so often. Urbit is now reaching the point where we're starting to see more of the obsidian edges of the frozen future system emerge from the lava.

References

Blow, Jonathan (2019) "Preventing the Collapse of Civilization". URL: `https://youtu.be/pW-SOdj4Kkk?t=2547` (visited on ~2024.8.30).

`~lagrev-nocfep`, N. E. Davis (2024). "Clay as a Typed Revision Control System." In: *Urbit Systems Technical Journal* 1.2, pp. 33–48.

Yegge, Steve (2011) "Stevey's Google Platforms Rant". URL: `https://gist.github.com/chitchcock/1281611` (visited on ~2024.8.30).