Typed Revision Control

N. E. Davis ~lagrev-nocfep
Urbit Foundation

Abstract

Contents

1 Introduction

A revision control system[1] is responsible for tracking the state and history of assets and asset changes within a particular continuity. It may also manage the production of assets through a build system, as used for instance by a continuous integration/continuous deployment (CI/CD) workflow. Trivially, an RCS is a logical filesystem: a database for managing files. (Conventional RCSs are agnostic to the underlying disk hardware,

[1] Also "version control system" (VCS); "source control system".

Manuscript submitted for review.

Address author correspondence to ~lagrev-nocfep.

however.[2]) Computer revision control systems have developed from the 1960s onwards once hard drives permitted files to be stored on tape rather than simply as punchcard files. Succeeding generations have introduced new approaches and features, converging more or less on the broad functionality today afforded by Git (**Torvalds2005**, **Torvalds2005**) in its elaborated form as a distributed source control system.[3] However, although some modest interest has been expressed for a more complete notion of type in revision control, no current system besides Urbit's Clay fully implements a typed revision-controlled filesystem. In this article, we explore the approach of current RCSs and elaborate Clay's contributions to the ongoing task of information management and auditable, reproducible source builds.

2 A History of Type in RCSs

The problem of type in revision control has been of both academic and practical interest for decades (**Perry1987**, **Perry1987**). Essentially, can file artifacts be organized and classified in such a way as to make them susceptible of comparison across their history? The simplest arrangement which is commonly made is to designate a file as either plaintext or binary. A plaintext file can be updated by specifying the character offset and run length to be replaced (an example of a "diff", or difference between two examples of text; typically these are resolved at the line level rather than character level). Plaintext files, whether ASCII or some form of UTF, are defined by a regular format with well-understood and predictable symbol widths and offsets. For instance, a little-endian representation of the text `'clay'` would be reproduced in ASCII hexadecimal as `0x7961.6c63` and ASCII binary as `0b111.1001.0110.0001.0110.1100.0110.0011`. (In binary, each character has a leading zero; that is, each entry

[2]Compare other logical file systems: network file systems like Microsoft Server Message Block (SMB) drives or distributed file systems like the Hadoop Distributed File System (HDFS) or the InterPlanetary File System (IPFS).

[3]Notably, Git has seemingly failed to fulfill its original promise of being a fully decentralized RCS: most users prefer to use the affordances of centralized Git services such as GitHub or GitLab.

is seven bits wide plus a 1-bit zero header.) In contrast, binary files (which is a byword for "everything else") have arbitrary layout and offset; it may be difficult to determine how and why to compare two revisions of any given file. Many rcss simply treat all registered data as byte streams.

All modern RCSs support plaintext and binary representations of files. Plaintext diffing at the level of line changes is straightforward; the Hunt–McIlroy algorithm is often employed (**Hunt1976**, **Hunt1976**).[4] However, the most popular revision control systems (such as Subversion and Git) have included only minimal direct support for binary diffs. Some systems only support storing successive file copies of a binary file; that is, the diff is the entire deletion of the old file and addition of the new file (e.g. CVS). Other systems may support binary file diffs by granular block and offset. Some RCSs support diffing particular binary files using a custom automatic merge tool. For example, the `textconv` tool converts each version of the file to a plaintext representation and compares those representations. (To show a diff for a changes in a PDF file, a configured Git instance could compare the text output of `pdfinfo` in each case as a proxy. This is generally legible as metadata but does not of itself include sufficient information to reproduce or reverse the intervening changes; the underlying RCS still stores the entire binary in its conventional representation.) In any case, type beyond file extension is not generally supported by RCSs; they cannot meaningfully merge files classified as binary.[5]

A general-purpose notion of file type is not isomorphic to conventional type in programming language theory. For instance, consider a JSON file. The following are equivalent as JSON artifacts:

[4] An implementation in Hoon is available at `@sigilante/diff`.

[5] As always, there are exceptions to the rules. The Git RCS, for instance, can use a "binary diff" format on exact preimages. The Subversion RCS does provide an elaborated MIME support system, wherein users can assign each file a MIME type that Subversion can use in applying and displaying diffs. However, in practice this only results in the classic text/binary dichotomy with slightly more granularity. "Subversion treats all file data as literal byte strings, and files are always stored in the repository in an untranslated state" (**Collins2016**, **Collins2016**).

```
{
  "name": "Alice",
  "age": 30
}
```

```
{
  "age": 30,
  "name": "Alice"
}
```

```
{ "name": "Alice", "age": 30, }
```

Indeed, since JSON is structurally agnostic to syntactic whitespace, an infinite number of equivalent files could be constructed. Clearly as plaintext representations, each of these differ, yet they are fully equivalent as JSON artifacts. An RCS which can recognize a JSON input can meaningfully store them as equivalent, while an RCS that makes only the trivial distinction between plaintext and binary must consider them all to be different. Furthermore, a JSON-aware RCS could meaningfully merge these files whereas a plaintext/binary RCS could not. Finally, a JSON-aware RCS could yield a different plaintext output when asked to produce the file, as it could normalize the whitespace to a canonical form. This may be considered by other systems to be an "incorrect" round trip.

Thus we must consider two intersecting notions of file type:

1. The file type as understood by the system, typically based on bitwise reproducibility (e.g. plaintext, binary).

2. The file type as understood conceptually, typically based on structural relationships (e.g. JSON, XML/HTML, Word `docx`, etc.).

The employment of an intermediate representation allows us to track changes to files and manage their history—state—more effectively. Consider the popular JavaScript framework React (**React**, **React**), aspects of which were developed to resolve difficulties with slow repainting in browsers (for a popular explanation, see **Arora2017** (**Arora2017**)). React's virtual

Document Object Model (DOM) system acts as an intermediate representation between the actual DOM and the developer's code. When a developer updates the virtual DOM, the framework compares the new virtual DOM with the previous virtual DOM and updates the real DOM only if necessary. This allows React to optimize the rendering process and improve performance. It also means that we can consider the structural representation of the DOM object as an intermediate representation, and the production and application of the diff as a type-aware RCS operation. Higher-level interfaces like Microsoft Word's "Track Changes" feature (**MS2024**, **MS2024**) similarly resolves diffs across text and formatting within a document, an artifact processed from a file into a user-friendly representation. While the "Track Changes" user interface is optimized for interactivity, conceptually it is quite similar to conventional merge tools from RCSs, and de facto resolves the same sorts of diffs for its particular file type. (Indeed, one can imagine a Word-aware command-line RCS that could meaningfully merge changes to a Word document, rather than simply storing successive copies of the file.) Thus a significant part of managing typed file data is deciding what the input, output, and diffs should look like, and the remainder is systematic application of these rules to the data.

Given this frame, consider **Perry1987**'s taxonomy of version control for module interfaces:

1. No version control. Files are simply replaced.

2. Basic version control. Files are versioned with a number, but no diff is applied.

3. Strongly-typed version control. Files are resolved at a structural level, such as procedures or arrays, and diffs are applied at that level.

In this scheme, contemporary Git-style RCS is a "weakly-typed" RCS: the system is aware of versions and diffs, but typically not below the file or line level.[6] Perry was concerned with the

[6]Oddly, Perry did not call out weak typing in his enumeration, although he did address it elsewhere in the article.

behavior and representation of semantic objects, rather than files or data:

> Version equivalence in this type of version control mechanism is defined in terms of syntactic equivalence: data objects are equivalent if their types or structures are equivalent; operations and modules are equivalent if their signatures are equivalent. (**Perry1987**, **Perry1987**)

Strongly-typed version control requires coupling to the build system, since the system must be aware of the structure of the data in order to apply diffs. (Indeed, **Blackman2024** (**Blackman2024**) discusses one solution to the linking problem which is well-suited to Urbit's global namespace.)

3 Clay

Urbit provides the Clay vane at `/sys/vane/clay` to manage Unix-style files and source builds. Clay acts as both the Unix-like filesystem and the source control vane. While Hoon itself (`/sys/hoon`) is responsible to compile a source noun into a Hoon abstract syntax tree (AST) and ultimately into executable Nock, the actual construction of source input from files is delegated to the `++ford` arm of Clay. This build process includes compiling the correct import files (such as libraries and shared structure files). Clay is also tightly coupled to the userspace vane Gall since system updates and agent updates both occur via Clay.[7]

To motivate how Clay functions as a revision control filesystem, we need to briefly examine some major data structures utilized by Clay. At the highest level, files are organized into "desks", described in previous literature as being analogous to Git branches but really more like Git repositories. A desk is a self-contained continuity of files; its state is the result of a

[7] We omit discussion of several other notable properties of Clay as a single-level store: referential transparency, global addressability, and event-level persistence, for instance.

history of commits. Desks may not include links to off-desk resources. Each desk must expose a few files at definite paths; primarily `/mar` for the marks used to read the desk contents.[8]

Ultimately, Arvo and Clay deal in nouns. A noun is a binary tree of unsigned integers. These may be evaluated as Nock formulas or manipulated as data. A noun is formally agnostic to the underlying hardware and is almost always accessed via some representation path (i.e. as phenomenon). A noun serves as a universally legible intermediate representation for all data and code.

Urbit has an immutable data model; the system state of Arvo is the unique result of the events in its history, stored as its event log. If a noun changes, it is the result of definite discrete changes to the system state. This yields very nice properties of referential transparency and has some ramifications in library management; see **Blackman2024** (**Blackman2024**) for more details.

Typically, a noun is altered either directly via subject-modifying code (`%~` censig, `=.` tisdot) or via a mark transformation, on which more later.

The state of a desk is a history of sequential commits, with each commit after the first having a parent or parents. TODO

While conventional file systems denote file type primarily by file suffix or a cursory search using "magic tests" on the file header, Urbit instead stores all data as a noun accessed via a mark. A mark is essentially a representation rule for nouns. Marks permit nouns to be stored with more granularity than the text/binary dichotomy facilitates, since details of conversion are stored as part of the mark.

A mark may be considered an executable MIME type.[9] A mark essentially describes how to map a data representation to a noun, and from a noun back out to a data representation. This conversion may be trivial (e.g. the binary storage of an

[8] While Arvo, the vanes, and userspace tools look at other definite locations, in principle one could construct a desk of arbitrary URL-safe paths as long as they are legible to Clay's build process via `/mar`.

[9] A media type (formerly MIME type) is a tag identifier for the intended data format of a particular data entry. We will use "MIME type" as this terminology remains in common use.

audio file) or sophisticated (from a text stream to a linked-list UTF representation). Data are accessed via a particular mark, and the appropriate conversion routine is invoked.[10] Clay is capable of searching for transition paths between mark representations, and in principle permits conversion between all compatible representations (if there were a path from Markdown `md` to rich text `rtf` to plaintext `txt` in marks, then the conversion is automatically supported by Clay).

> A mark is a term (symbol) which is the Urbit equivalent of a MIME type, if MIME types were names of typed validation functions. ([, [)p. 51]Whitepaper

Unix filesystem mounting and committing involves synchronizing a desk's state with a Unix logical representation of the files involved. The actual file data are imported as `%mime`-typed nouns and then converted to their target mark as file type.

Each mark defines several standard arms:

1. `++grab` converts from other marks (by arm name) to a given mark.
2. `++grow` converts from the mark to other marks.
3. `++grad` defines the diffs applicators for the mark.

Marks do not need to be symmetrical or round-trip. A JSON input may yield an equivalent JSON output but altered by structural whitespace. Clay is also proactive in searching for possible mark paths—if an immediate conversion is not available, then a mutually available intermediary form may be used. To implement a custom filetype, all that is required is the production of a suitable mark file.

While Clay has at times supported diffs, it currently simply stores the entire file at each commit. (No technical limitation is in place; this was merely a stylistic change while Urbit data

[10] This is commonly used with JSON and the Eyre HTTP server vane, for instance.

storage is relatively small and it is expected to be reverted as larger loom sizes are supported.)

diffs etc.

Another possible move is to abstract away from the file as the fundamental artifact being tracked. While Clay does not fully commit to this, in a concession to its POSIX-compliant host OS, it gestures towards the possibility of conceiving of build sources and artifacts as objects (cores) other than files.

3.1 Shortcomings of Clay

As currently implemented, Clay presents several wrinkles in developer ergonomics. Chief among these is the way that marks, as simple type tags, are underspecified. The original philosophy of marks centered on network transmission of nouns as tagged data:

> Like a MIME type, the mark is just a label. There is no way to guarantee that the sender and receiver agree on what this label means. A noun which doesn't normalize to itself is a packet drop. ([, []pp. 51–52]Whitepaper

In practice, different Clay desks and different Urbit versions can have different mark processors (and thus different behvaior for the same tag). No simple version system tracks these; there is no global type registry (other than the scry namespace), and any given mark is simply identified by its desk and path. Developers have expressed a desire for a more robust mark management capability, and solutions have ranged from a global type registry (e.g. that proposed by Archetype (**TODO**, **TODO**)) to using paths instead of simple tags for marks.

The current architecture of Clay (largely in place by 2016 except for the unification of the former Ford vane into Clay in 2020) leads to three ongoing developer concerns:[11]

1. Implementing RCS behavior at the file system layer seems to be incorrect for Urbit (that is, too much of a sop to

[11]This section benefited from discussion with `~tiller-tolbus` about the upcoming "shrubbery" project to rework Urbit's userspace.

Unix). trying to do revision control at base layer, but file system is wrong layer (you just want history of actions/tx log rather than full state); see migrev codebases for clay state

2. Desks seem to be the wrong abstraction in practice. The theory of the desk doesn't account for actual distribution patterns people would like to use; we want one desk but one spot in file tree is capable of acting as a desk (chroot analysis); overuse of desks makes Urbit not feel as much like a filesystem you can explore

3. The application system (userspace) and the build system and filesystem should be co-located. Currently, Clay and Gall are deeply entangled but must interact via a somewhat awkward message-passing interface. This leads to a number of ergonomic issues, such as the need to suspend a desk in order to suspend an agent. Employing paths rather than simple tags would lead to a desire to constrain what can exist at certain paths, which of course relates to the definition of files as data structures.

The future of Clay likely sees it being restricted to source and build management, rather than expansion to a more fully-featured filesystem. In this contingency, the userspace management vane may take over conventional file storage, as Gall or a successor. Alternatively, the build system, RCS, and application engine fuse into a single userspace vane, which is one possible end point of the "shrubbery" project. Marks in that case will be replaced with simpler conversion rules permitting only straightforward type casts between nouns.

4 Conclusion

In its current instantiation, Clay exhibits some notable characteristics as a typed revision control system. It is capable of tracking file changes at a structural level, permitting meaningful diffs and merges for files of the same mark. It is also capable of converting between marks, permitting a more flexi-

ble approach to file type than the conventional text/binary dichotomy. However, the future of userspace and code building is still in flux and may see a reworking of the Clay vane to better suit the needs of developers and users.

Perry1987: https://users.ece.utexas.edu/ perry/work/papers/icse9b.pdf and refs

Yarvin2016: https://urbit.org/blog/toward-a-new-clay - frodwith