# Files-as-Filesystems for POSIX Shell Data Processing

Michael Greenberg
michael.greenberg@stevens.edu
Stevens Institute of Technology
Hoboken, New Jersey, USA

## Abstract

The POSIX shell is 'stringy', and its ecosystem primarily supports line-oriented formats. While such formats are popular and common, contemporary programming often involves semi-structured data, like JSON or YAML. Dealing with such formats, the shell's stringiness leaves users out in the cold—the POSIX ecosystem struggles with semi-structured data. New command-line tools work well with 'modern' data formats, but each tool is its own complex language to learn.

The tree-like filesystem is the shell's only real data structure. By mapping 'modern' formats onto file hierarchies, we can work effectively in the existing ecosystem.

We introduce `ffs`, the file filesystem, a new tool for mapping semi-structured data formats to filesystems in userspace. Like `/proc` and `/sys`, our filesystem-based approach helps the shell (and other tools) manipulate structured data.

**ACM Reference Format:**
Michael Greenberg. 2021. Files-as-Filesystems for POSIX Shell Data Processing. In *11th Workshop on Programming Languages and Operating Systems (PLOS '21), October 25, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3477113.3487265

## 1 Introduction

The POSIX shell is a critical and widespread tool for configuration, deployment, management, and forensics. As researchers think more about how best to support and improve the command-line experience in general and the POSIX shell in particular [13, 14, 24, 27], it is critical to also consider the command-line ecosystem.

While the POSIX shell's semantics are powerful [9, 12], shell scripts tend to spend their time running other programs. The shell is a coordinator, sending data to and from programs primarily as *strings*: the program arguments in `execve` are strings; command substitutions work with strings; pipes and redirections are strings or binary data. The shell's semantics itself is 'stringy', i.e., it deals only with flat strings. Word expansion rewrites strings before actually evaluating commands [10]. (There are non-POSIX shells with other approaches, of course—see Section 4.)

Historically, the shell's stringiness lived in an ecosystem built for such stringy data. Tools like `grep`, `cut`, `head`, and `tail` are line-oriented. The `awk` programming language [1, 2] is the apotheosis of line-oriented processing.

The shell's line-oriented approach falls over on more structured data. Popular formats like JSON, YAML, and XML are recursively structured, without reliably meaningful whitespace. Fortunately, tools have emerged to help command-line users process these formats, like `jq`, a command-line JSON processor in the spirit of `awk` [6], and `gron`, a query tool for JSON in the spirit of `grep` [16]. Unfortunately, each such tool is its own `awk`-level tool to learn.

We are left in an awkward situation. The shell remains an excellent tool for one-off work, and is often the right tool for more substantial tasks. But as soon as a 'modern' format is involved, things get much more difficult. While it's very easy to process JSON in JavaScript or Python or R, it is hard to 'live' in these tools as one does the shell. Everything is unpleasantly polyglot: one must bounce around between tools for simple, one-time tasks.

The shell's difficulty working with semi-structured data is both a language issue and an ecosystem issue. The shell excels in working with strings, but it really only has one data structure: the file system (Section 2). To work effectively in the shell with data of any kind, the data must either be mapped to strings or to the filesystem itself. We introduce `ffs`, the file filesystem, a tool for mapping semi-structured data into file hierarchies using filesystems in userspace (Section 3). The *idea* of mapping highly structured data to file hierarchies of course predates `ffs` (Section 4). The filesystem is the *lingua franca* of the POSIX world. Tools that map worthwhile formats and system structures into filesystems are enabling technology in the command-line ecosystem and beyond (Section 5).

## 2 Data structures in the POSIX shell

The POSIX shell has only two data structures: strings in the environment and the tree-like filesystem. While alternative shells offer much more interesting structures (Section 4), portable shell scripts can't rely on them. The filesystem is well structured in a way that the shell's environment is not. By translating structured data to the filesystem, we make it easier for the shell to process than if it were in plain files.



*PLOS '21, October 25, 2021, Virtual Event, Germany*

ACM ISBN 978-1-4503-8707-1/21/10.
https://doi.org/10.1145/3477113.3487265

```
$ echo '{}' >demo.json   # (A)
$ ffs -i demo.json &     # (B)
[1] 17309
$ cd demo
demo $ echo 47 >favorite_number # (C)
demo $ mkdir likes
demo $ echo true >likes/dogs
demo $ echo false >likes/cats
demo $ touch mistakes
demo $ echo Michael Greenberg >name
demo $ echo https://mgree.github.io >website
demo $ cd ..
$ umount demo
$
[1]+  Done                    ffs -i demo.json
$ cat demo.json # (D)
{"favorite_number":47,"likes":{"cats":false,
"dogs":true},"mistakes":null,"name":
"Michael Greenberg","website":
"https://mgree.github.io"}
```

**Figure 1.** Command-line use of `ffs` to create a JSON file, by (A) creating and (B) mounting an empty JSON file and then (C) editing it in-place (`-i`) to contain strings, numbers, and nested objects. (Output is semantically wrapped but otherwise unedited.) Unmounting (D) saves the file.

Programs in the POSIX shell use environment variables to store local and system-wide state. Environment variables are, at core, strings, though they are sometimes also interpreted as numbers (e.g., arithmetic expansion). The stringiness of POSIX environment variables limits what the shell can do. Between newline trimming of command substitutions and issues with null bytes, environment variables *can't* be used to process binary data. Shell operators like prefix/suffix patterns allow some primitive string operations; tools like `sed` and `grep` offer more power. The shell semantics makes it not too difficult to use strings as lists: experts can set `IFS` to control how field separation breaks up inputs to `for` loops and other forms. None of these tools let the shell reliably process recursively structured data.

The filesystem, however, is basically a tree. Hard links make it DAG-like, and symbolic links make it a nearly arbitrary graph. The shell is quite adept at moving around the filesystem: manual use is easy, with `cd -` for backtracking and `pushd`/`popd` for more complex situations; programmatically, `for` loops with recursion and the `find` tool both make it easy to traverse large directory structures in the shell.

Our idea is to map structured data formats onto the filesystem. Navigating and manipulating the filesystem is a breeze in the shell ecosystem as it exists today. By utilizing existing tools, users can do the usual shell things—spot checks and quick fixes—on data in modern formats. Rather than learning specialized tools (like `jq` or `gron`), users can go on using their preferred tools, whether that means `sed` or `vim` or `emacs`.

```
$ curl https://api.github.com/
    repos/mgree/smoosh/commits >commits.json
$ cat commits.json
[ { "sha": ..., "commit": ..., ... }, ... ]
$ ffs commits.json --no-output &
[1] 2883
$ for msg in $(find . -name message); do
    if grep test $msg >/dev/null; then
      echo $(cut -c 1-7 $(dirname $msg)/../sha)
           matches: $(cat $msg);
    fi;
  done
a930035 matches: two more tests from old ...
cd923b5 matches: set path for integration tests
75ff2b4 matches: drop fds test---it's gh
da8bd4d matches: log ulimit for macOS test ...
0321690 matches: switch to gh actions, ...
451824a matches: more tests
2ee1784 matches: Exit status carryover (#42) ...
639d70d matches: test fds output
dd897d1 matches: abort test early if fd ...
ce28260 matches: test
8b9647b matches: close fds before testing, ...
4b55ccd matches: plain fds test
```

**Figure 2.** Exploring a large JSON file with `ffs` (with light elisions for space constraints).

## 3 `ffs`: the file filesystem

The `ffs` tool [11] uses filesystems in userspace (FUSE) to treat semi-structured data as inspectable and editable file hierarchies. Users mount a file as a filesystem by running `ffs` in the background; the filesystem lets them manipulate (Figure 1) and explore (Figure 2) the mounted fileystem. Altering the mounted filesystem updates the data in the file. As of version 0.1.1., `ffs` supports JSON, YAML, and TOML formats. `ffs` is 3k SLOC of Rust, with 2.5k SLOC of (fairly redundant) shell scripts as tests and benchmarks. Adding formats that match `ffs`'s data model (Section 3.1) is easy. The save/load logic is independent of individual formats (Section 3.2). While `ffs` is a usable prototype, issues and questions remain (Section 3.3).

### 3.1 Data model

`ffs`'s job is to take semi-structured data and map it to a filesystem hierarchy of inodes. FUSE filesystems support all seven POSIX XSI file types, but `ffs` only makes use of regular files and directories. For a given *format*, the `ffs` data model comes in three parts (Figure 3): first, the format is characterized as some set of *values*; next, *types* characterize the shape

**Formats**

| | | |
|---|---|---|
| Format values | $v$ | $\in \mathcal{V}$ |
| | toNode | $: \mathcal{V} \to \mathcal{N}$ |
| | fromUtf | $: \mathcal{T} \times \Sigma^*_{\text{UTF-8}} \to \mathcal{V}$ |
| | fromRaw | $: \mathbb{BV}^*_8 \to \mathcal{V}$ |
| | fromMap | $: (\Sigma^+_{\text{UTF-8}} \rightharpoonup \mathcal{V}) \to \mathcal{V}$ |
| | fromVec | $: \mathcal{V}^* \to \mathcal{V}$ |

**Types**

| | | |
|---|---|---|
| Directories | $dir$ | ::= named \| list |
| File types | $fty \in \mathcal{T}$ | ::= auto \| null \| boolean \| integer \| float \| datetime \| string \| bytes |

**Nodes**

| | | |
|---|---|---|
| Nodes | $node \in \mathcal{N}$ | ::= utf($fty, s$) \| raw($b$) \| map($s_i^+ \mapsto v_i$) \| vec($v_i$) |
| UTF-8 | $s$ | $\in \Sigma^*_{\text{UTF-8}}$ |
| —"— nonempty | $s^+$ | $\in \Sigma^+_{\text{UTF-8}}$ |
| Bytes | $b$ | $\in \mathbb{BV}^*_8$ |

**Figure 3.** The `ffs` data model

of files and directories and determine how data is serialized back from the filesystem; finally, *nodes* are an internal data model that translate directly to inodes (Section 3.2).

The `ffs` type system characterizes each directory *dir* as either a *named directory* (named) or a *list directory* (list). In JSON terms, named corresponds to objects and list corresponds to arrays. In actual filesystems, every file has a name—when we work with list directories, we track names in the filesystem but load and serialize based on the filename sorting order in the current locale. New directories default to named. The type system characterizes each file as one of eight types. Half correspond to familiar types (boolean, integer, float, string). We include the datetime type to support formats with a notion of date, like TOML. The null type supports formats with a notion of a null value (JSON, YAML). The bytes datatype is for binary data—which none of JSON, YAML, or TOML actually support. Files of type bytes will be written back in base64 encoding.[1] Finally, auto means "please guess the type". New files are given the auto type.

The core data model is made up of *nodes*. A *node* is one of four things: a typed UTF-8 string (`utf`), raw bytes (`raw`), a sequence of abstract values (`vec`), or a map from nonempty UTF-8 strings to abstract values (`map`). The `utf` and `raw` nodes correspond to regular files; `utf` nodes can have any type (including bytes), while `raw` nodes always have type bytes. The `map` and `vec` nodes correspond to named and list directories, respectively. These directory nodes hold *abstract format values*, not further nodes. By not converting values to nodes up front, we can write a generic save/load implementation (Section 3.2) and leave room for alternative regimes, like lazy loading (Section 5).

To actually map data into the data model, a format must define three things: a notion of value $v$, a value-to-node function toNode that translates a value $v$ into a *node*, and a suite of four *node*-to-$v$ functions (fromUtf, fromRaw, fromMap, fromVec). All `ffs` needs to support a format are these parts (and routines for parsing and serializing format strings to and from abstract values).

For example, the following definitions map JSON into `ffs`'s data model. Define JSON values $j$ as follows, where $n$ is a 64-bit floating point number and $s$ is a UTF-8 string:

$$j ::= \texttt{null} \mid \texttt{true} \mid \texttt{false} \mid n \mid \texttt{"}s\texttt{"} \mid \{s_i^+ : j_i\} \mid [j_i]$$

We map JSON values $j$ into nodes in a straightforward way, where $\epsilon$ is the empty string and format renders numbers:

$$\begin{aligned}
\text{toNode}(\texttt{null}) &= \texttt{utf}(\text{null}, \epsilon) \\
\text{toNode}(\texttt{true}) &= \texttt{utf}(\text{boolean}, \text{true}) \\
\text{toNode}(\texttt{false}) &= \texttt{utf}(\text{boolean}, \text{false}) \\
\text{toNode}(n) &= \begin{cases} \texttt{utf}(\text{integer}, \text{format}(n)) & \text{if } n = \lfloor n \rfloor \\ \texttt{utf}(\text{float}, \text{format}(n)) & \text{otherwise} \end{cases} \\
\text{toNode}(\texttt{"}s\texttt{"}) &= \texttt{utf}(\text{string}, s) \\
\text{toNode}(\{s_i^+ : j_i\}) &= \texttt{map}(s_i^+ \mapsto j_i) \\
\text{toNode}([j_i]) &= \texttt{vec}(j_i)
\end{aligned}$$

Translating nodes back to values is mostly boring: fromMap and fromVec just create objects and arrays respectively, and fromRaw base64 encodes its input and returns a JSON string. But fromUtf is more interesting: it uses the type information to try to generate appropriate JSON values, with auto meaning "please guess". Other types are given their best effort, defaulting to strings. We show the auto and null cases:

$$\begin{aligned}
\text{fromUtf}(\text{auto}, s) &= \begin{cases} \texttt{null} & \text{if } s = \epsilon \\ \texttt{true} & \text{if } s = \text{true} \\ \texttt{false} & \text{if } s = \text{false} \\ n & \text{if strtod}(s) = n \\ s & \text{otherwise} \end{cases} \\
\text{fromUtf}(\text{null}, s) &= \begin{cases} \texttt{null} & \text{if } s = \epsilon \\ s & \text{otherwise} \end{cases} \\
\text{fromUtf}(\ldots, s) &= \ldots
\end{aligned}$$

We settle for best effort over warnings to avoid being noisy; it would be easy to make this configurable.

### 3.2 Saving and loading inodes

Given a format definition, `ffs` can parse a value and convert it to a list of *inodes*, numbered data structures tracking

[1] `ffs` doesn't try to detect base64 data in the input. Every alphanumeric string is (more or less) valid base64. Guessing wrong would be common and confusing! There are existing command-line tools for converting base64 to and from binary data, so it seems prudent to leave conversion to the user.

file/directory contents and types, along with standard filesystem metadata. These inodes service the various FUSE system calls, which refer to files and directories by inode number, with inode 1 being the root of `ffs`'s filesystem (and its own parent). When the filesystem is unmounted, `ffs` walks the inode list to generate and emit a new value. The saving and loading is generic, thanks to the abstract *node* architecture.

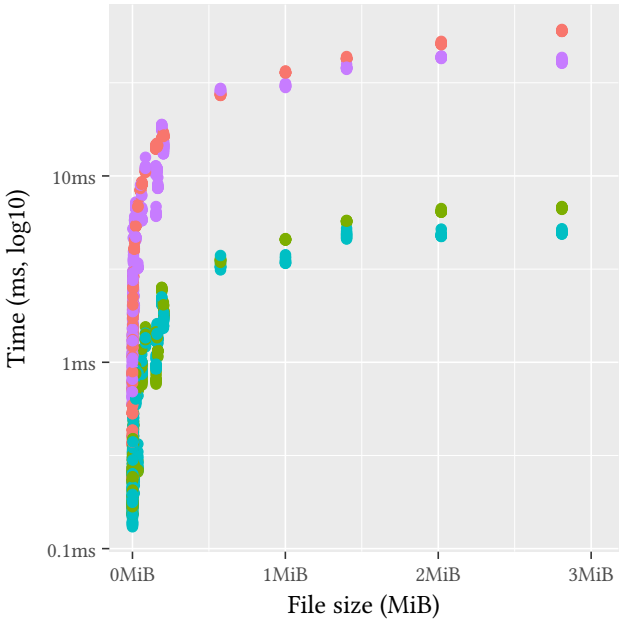
***Loading.*** `ffs` generates inodes by depth-first search on a format value $v$, using toNode to convert values to nodes as the search proceeds. Invalid filenames must be *munged* (Section 3.3, "Name munging"). Munged names are stored in metadata, and fieldnames are restored unmunged if those files weren't destructively renamed. In list directories, we simply number the files, starting at 0. In order to ensure that directory listings are sorted properly, filenames will be 0-padded by default, i.e., if there are 120 items, the first file will be `000`, the next will be `001`, and so on through `119`.

***Saving.*** When the filesystem is unmounted, `ffs`'s default behavior is to translate the inodes back into the initial format and emit it on `STDOUT`. Flags let you alter that behavior, e.g., the `-i` in-place flag (Figure 1) or the `--no-output` silencing flag (Figure 2). We traverse the inodes depth-first, using fromVec and fromMap to convert directories. When adding new files to list directories, any filename is allowed; we serialize back in sorted order for the current locale. We restore the original, unmunged name where possible. Some files are ignored by default (e.g., `._` files holding extended attributes on macOS). When converting files, if the contents of the file are valid UTF-8 and the file's type isn't bytes, we use fromUtf; otherwise we use fromRaw.

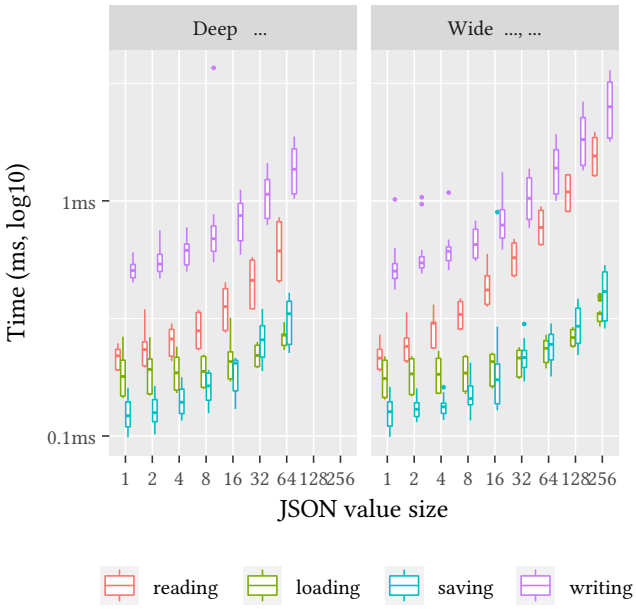
***Operations.*** The implementations of the actual FUSE operations are not particularly interesting: we look things up in the inode list and perform the operations requested. There are 41 such operations in the `fuser` Rust bindings [3], which only offers the (more efficient) low-level FUSE bindings. We leave 18 operations with the default implementation (e.g., `create` forces the OS to use `mknod` and `open`) or unsupported (`bmap`, `copy_file_range`, `ioctl`, links, locks, syncing, `readdirplus`, and some macOS calls all return `ENOSYS`).

### 3.3 Challenges

***Performance.*** `ffs` works well for quick tasks, with zippy interactive performance on the small files one typically finds in formats like JSON, YAML, and TOML (~10ms load times on files 100KiB and under; Figure 4). All tests were run on a MacBook Pro running macOS 10.13 with a 2.5 GHz Intel Core i7 (4 cores) and 16GB RAM. The computer was unloaded and the network was off. Each benchmark is run 10 times, and all runs are shown. As input files grow above 0.5MiB, the startup and shutdown times become noticeable. Startup is dominated by reading and parsing values while shutdown is dominated by unparsing values and writing; in both cases, saving and




(a) 53 real-world JSON files of varying sizes and 4 tiny synthetic JSON files (included to ensure that various renaming features are exercised). The y-axis is log scaled; the x-axis is linear.




(b) Synthetic JSON files exercise deep and wide nesting. The linear relationship is presented log-scaled in both axes for readability. There are no values for depths greater than 64; both the Python and Rust JSON parsers hit recursion limits.

**Figure 4.** Initial performance evaluation of `ffs` startup and shutdown. 'Reading' is reading and parsing JSON from storage; 'loading' is generating inodes; 'saving' is generating a JSON value from inodes; 'writing' is unparsing and writing JSON back to storage.

loading the inode structure is comparatively cheap (Figure 4 (a) and (b)). There are several bottlenecks: files are parsed wholesale into format values $v$; inodes are generated eagerly before mounting happens.

In general, it is impossible to avoid parsing overhead entirely—the recursive nesting in these formats means intermediate structures must be at least scanned, if not parsed all the way. Some of the format-parsing libraries support streaming or lazy conversion to data structures, which would reduce the parsing bottleneck. In any case, the Rust libraries for these formats are generally fast, parsing hundreds of megabytes per second. It would be relatively easy to reduce the inode generation bottleneck by introducing a lazy inode that held a format value $v$, to be partially traversed on demand. A multithreaded implementation could continue loading in the background, between system calls. Loading might also benefit from learning parsers, like Mison [19].

When saving and emitting the final output, it might be possible to reuse any unedited format values, trading time for memory. In some circumstances, it might even be possible to leave most of the file itself intact—though detecting that situation might take more effort than is saved.

***Name munging.*** Some valid field names are invalid filenames, and vice versa. Any UTF-8 string is a valid JSON field; TOML 'bare' keys are quite limited, but quoted keys can be any string, even the empty one; Yaml keys can be arbitrary *structures*. On the other hand, filenames must be nonempty, the filenames `.` and `..` are both reserved, and the null byte is forbidden. Characters like `-` and `*` and `?` are legal, but they are inconvenient to work with, as are whitespace characters.

`ffs`'s current approach is ad hoc. A `--munge` flag switches between renaming and filtering fields. Renaming can lead to collisions, which not only demand more renaming in turn, but also make the internal format value's ordering observable. If no edits are made, munged names will be correctly unmunged when saving the data back. Not so for filtering: filtering is a nice way to say "data loss". On the one hand, offering a panoply of configuration flags to control these behaviors is an unappealing solution; on the other hand, warnings, errors, or interactivity all reduce `ffs`'s programmatic usability in scripts and other automation.

***Metadata.*** We use the extended attribute `user.type` to track the *dir* and *node* types; these can be set to, e.g., convert a list directory to a named one, or force a file to be interpreted as bytes. Such an approach works, but is a poor affordance: one must read the documentation and learn to use a tool for managing extended attributes. There are alternatives, all of which introduce more name clashes: we could use filename extensions to indicate types; or we could use some kind of file-based metadata scheme, where `foo`'s metadata is in `.foo.type` or `.types/foo` (allocating and managing inodes for many such files) or in a per-directory file `.types` (making individual type updates more cumbersome).

***Constraints.*** `ffs`'s data model is presented here as unconstrained, but in fact formats and filesystems come with constraints. FUSE filesystems must have a directory at the root, but JSON and YAML have no such constraints (TOML requires the root to be named). On the flipside, it's possible to use extended attributes to force the root to be a list directory, which violates TOML's constraint. In `ffs`, all of these situations lead to runtime errors.

YAML has an incredibly rich data model, making the full YAML specification notoriously difficult to get right. Two features complicate `ffs`'s work: arbitrary keys and aliases. In YAML, a "mapping"—a named directory and all its contents—can be a *key* to another mapping. That would correspond to the baffling notion of treating a directory and its contents as a filename. When we encounter such structured keys, we just use their hash. YAML's aliases allow named values to be repeated. Our YAML parser handles these automatically, but may not work in all cases—and there are no hooks to treat such aliases as, say, symbolic links.

Finally, we have not yet implemented XML bindings for `ffs`. XML's data model is rich with attributes and PCDATA, i.e., interleaved text and child tags. It's not at all clear which of the many possible mappings would be appropriate; following xmlfs [20] might be easiest approach.

## 4 Related work

***Complex data as file hierarchies.*** Killian introduced `/proc`, a pseudo-filesystem allowing file-based access to system and process information, for Unix V8 [18]. Faulkner and Gomes [7] ported the concept to UNIX System V. `/proc` remains a popular interface on Linux.

Wimmer [28] proposes "files as directories", of which `ffs` is an instance; he gives an overview of previous approaches. We refine his proposal, arguing that mapping structured data to file hierarchies is particularly well suited to the POSIX shell ecosystem. Tchernavskij [23] offers an ambitious vision of file-to-filesystem mappings, and suggests investigating the "cognitive and operational details". `ffs` can be seen as investigating the operational details for the particular task of working with semi-structured data. Kell [17] agrees with Wimmer's approach, but recharacterizes "the desired affordance" as "not directories per se, but the ability to swap straightforwardly between a byte-stream and a directory 'view' of the data". `ffs`'s in-place editing gives some of that feel, but `ffs` does not support simultaneous editing of the JSON file and its filesystem mapping. Kell's concerns around "bidirectionality" recur in our setting (Section 3.3).

Augeas [21] is a configuration tool that maps files in `/etc` to a unified tree structure and edits the tree in a custom shell, using lenses [5] to map edits back to system configurations. `ffs` works with the user's choice of shell and ecosystem. Modulo constraints, `ffs`'s mapping is not just a lens, but in fact a bijection.

***Filesystems in Userspace.*** Filesystems in userspace, a/k/a FUSE, are not a new idea. The FUSE API is available on a variety of Unix-like systems (Linux, FreeBSD, macOS); Windows uses its own Projected File System API. 9p is a simpler protocol than FUSE; it is network-based and implemented on Linux via FUSE. FUSE filesystems range from toys (including toy implementations of JSON filesystems) to commonly used systems like fuse-zip to serious implementations of cloud storage products; see Vangoor et al. [25] for an overview.

Unsurprisingly, FUSE is not as performant as in-kernel filesystems, with nearly 6x (-83%) slower throughput than conventional filesystems [25, 26]. XFUSE [15] is a userspace filesystem framework that offers significant speedups over FUSE. Since XFUSE is FUSE-compatible, we imagine that dropping `ffs` into XFUSE would offer speedups. The works just cited evaluate *stackable* filesystems: passthrough shims that are backed by ext4. XFUSE simulates a variety of possible underlying block devices, including RAMDisks—where the overheads of FUSE cause it to perform much worse than XFUSE, which also performs worse than ext4. Since `ffs` works entirely in-memory after parsing its input, it probably suffers similarly high overheads.

Various projects aim to speed up FUSE by crossing the user/kernel boundary less often. Direct-FUSE [29] offers 1.1x speed ups by moving more of FUSE into userspace. ExtFUSE [4] specializes the FUSE interface, running some filesystem logic as eBPF inside the kernel. ExtFUSE speeds up 'stackable' or 'passthrough' filesystems, which just modify some other filesystem's behavior; it's not clear how `ffs` might benefit. Bento [22] offers 7x speed ups by moving everything into kernelspace, relying on Rust for safety. `ffs` is already written in Rust, and would be a good fit for Bento.

***Alternative shells.*** We focus on supporting the POSIX shell with `ffs`, but each system uses some *particular* shell. `bash` is almost certainly the most popular such shell, and is mandated as the language for shell scripts at Google [8]. It offers one-dimensional arrays, which simplify ad hoc list handling—but arrays aren't a good fit for recursive data. `bash` also offers 'loadable builtins' for, e.g., CSV processing. `zsh` is also popular, and it goes well beyond POSIX. It has both arrays and dynamically loadable 'modules', which offer new builtins for working with structured data (e.g., the termcap database). Neither has tools for semi-structured data.

Both `bash` and `zsh` are POSIX-like, but new shells take wildly different paths. `fish` feels POSIX-ish interactively, but it offers a more conventionally structured scripting language—that doesn't support semi-structured data. PowerShell is an object-oriented shell for Windows that exposes the underlying OS objects. PowerShell can be run on any .NET platform, but it's most useful on top of the object model underlying Windows. `nushell` uses a tabular data model; it works with formats like JSON and TOML natively. Shell-like scripting languages and libraries offer general purpose programming tools; see Greenberg [10] for a review.

## 5 What's next?

What is the best way to evaluate `ffs`? Prior work on FUSE filesystems focus on block-device backed filesystems, rather than in-memory systems, where we can expect the relative overhead of context switches to be much higher. What is the right baseline? Should `ffs` be compared to using an editor for interactive work, or to scripts in JavaScript or Python or a notebook? What tasks and benchmarks are appropriate? Lazy loading promises reductions in startup time on large files. What other optimizations are worthwhile?

It would be good to guarantee that each format's mapping to and from the `ffs` data model was correct, i.e., a bijection. Supporting more formats would make `ffs` more useful. XML is a natural choice, as are binary formats like CBOR and BSON. More broadly, what other file-to-filesystem mapping tools would be useful? ELF and other executable formats? Media and their metadata? What else?

## Acknowledgments

Konstantinos Kallas, Nikos Vasilakis, Philip Guo, and Ralph Wimmer provided encouragement and insight. Alex Denisov, Andy Chu, John Regehr, and various tweeps brainstormed. Ciprian Manea reported bugs. Erez Zadok, Jay McCarthy, and Vasily Tarasov helped with related work. The PLOS reviewers provided stimulating and helpful feedback, too.

## References

[1] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. 1988. *The AWK Programming Language.* Addison-Wesley.

[2] The Austin Group. 2018. POSIX.1 2017: The Open Group Base Specifications Issue 7 (IEEE Std 1003.1-2008). "Shell & Utilities", `awk`. https://pubs.opengroup.org/onlinepubs/9699919799/utilities/awk.html

[3] Christopher Berner. 2021. `fuser`. https://github.com/cberner/fuser Accessed 2021-08-25.

[4] Ashish Bijlani and Umakishore Ramachandran. 2019. Extension Framework for File Systems in User space. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 121–134. https://www.usenix.org/conference/atc19/presentation/bijlani

[5] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. 2008. Boomerang: resourceful lenses for string data. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 407–419. https://doi.org/10.1145/1328438.1328487

[6] Stephen Dolan. [n.d.]. `jq`. https://github.com/stedolan/jq Accessed 2021-08-25.

[7] Roger Faulkner and Ron Gomes. 1991. The Process File System and Process Model in UNIX System V. In *Proceedings of the USENIX Association Software Tools User Group, Winter 1991, Dallas, TX, USA, January, 1991.*

[8] Googlers. 2020. Shell Style Guide. https://google.github.io/styleguide/shellguide.html Accessed 2021-08-25.

[9] Michael Greenberg. 2018. The POSIX shell is an interactive DSL for concurrency. DSLDI.

[10] Michael Greenberg. 2018. Word expansion supports POSIX shell interactivity. In *Programming Companion (presented at Programming eXperience (PX))*. ACM. https://doi.org/10.1145/3191697.3214336

[11] Michael Greenberg. 2021. `ffs`. https://github.com/mgree/ffs Accessed 2021-08-25.

[12] Michael Greenberg and Austin J. Blatt. 2020. Executable Formal Semantics for the POSIX Shell. *Proc. ACM Program. Lang.* 4, POPL, Article 43 (Dec. 2020), 30 pages. https://doi.org/10.1145/3371111

[13] Michael Greenberg, Konstantinos Kallas, and Nikos Vasilakis. 2021. The Future of the Shell: Unix and Beyond. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 240–241. https://doi.org/10.1145/3458336.3465296

[14] Michael Greenberg, Konstantinos Kallas, and Nikos Vasilakis. 2021. Unix Shell Programming: The Next 50 Years. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 104–111. https://doi.org/10.1145/3458336.3465294

[15] Qianbo Huai, Windsor Hsu, Jiwei Lu, Hao Liang, Haobo Xu, and Wei Chen. 2021. XFUSE: An Infrastructure for Running Filesystem Services in User Space. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 863–875. https://www.usenix.org/conference/atc21/presentation/hsu

[16] Tom Hudson. 2021. `gron`. https://github.com/tomnomnom/gron Accessed 2021-08-25.

[17] Stephen Kell. 2018. Critique of 'Files as Directories: Some Thoughts on Accessing Structured Data within Files' (2). In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming* (Nice, France) *(Programming'18 Companion)*. Association for Computing Machinery, New York, NY, USA, 175–179. https://doi.org/10.1145/3191697.3214325

[18] T. J. Killian. 1984. Processes as Files. In *Proceedings of the USENIX Association Software Tools User Group, Summer 1984, Salt Lake City, UT, USA, June, 1984*.

[19] Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. 2017. Mison: A Fast JSON Parser for Data Analytics. *Proc. VLDB Endow.* 10, 10 (June 2017), 1118–1129. https://doi.org/10.14778/3115404.3115416

[20] Henrik Lindberg. 2010. `xmlfs`. https://github.com/halhen/xmlfs Access 2021-09-20.

[21] David Lutterkort. 2008. Augeas–a configuration API. In *Linux Symposium, Ottawa, ON*. Citeseer, 47–56.

[22] Samantha Miller, Kaiyuan Zhang, Mengqi Chen, Ryan Jennings, Ang Chen, Danyang Zhuo, and Thomas Anderson. 2021. High Velocity Kernel File Systems with Bento. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 65–79. https://www.usenix.org/conference/fast21/presentation/miller

[23] Philip Tchernavskij. 2018. Critique of 'Files as Directories: Some Thoughts on Accessing Structured Data within Files' (1). In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming* (Nice, France) *(Programming'18 Companion)*. Association for Computing Machinery, New York, NY, USA, 171–174. https://doi.org/10.1145/3191697.3214324

[24] Priyan Vaithilingam and Philip J. Guo. 2019. Bespoke: Interactively Synthesizing Custom GUIs from Command-Line Applications By Demonstration. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology* (New Orleans, LA, USA) *(UIST '19)*. Association for Computing Machinery, New York, NY, USA, 563–576. https://doi.org/10.1145/3332165.3347944

[25] Bharath Kumar Reddy Vangoor, Prafful Agarwal, Manu Mathew, Arun Ramachandran, Swaminathan Sivaraman, Vasily Tarasov, and Erez Zadok. 2019. Performance and Resource Utilization of FUSE User-Space File Systems. *ACM Trans. Storage* 15, 2, Article 15 (May 2019), 49 pages. https://doi.org/10.1145/3310148

[26] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. 2017. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 59–72. https://www.usenix.org/conference/fast17/technical-sessions/presentation/vangoor

[27] Nikos Vasilakis, Konstantinos Kallas, Konstantinos Mamouras, Achilles Benetopoulos, and Lazar Cvetkovic. 2021. PaSh: light-touch data-parallel shell processing. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar (Eds.). ACM, 49–66. https://doi.org/10.1145/3447786.3456228

[28] Raphael Wimmer. 2018. Files as Directories: Some Thoughts on Accessing Structured Data within Files. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming* (Nice, France) *(Programming'18 Companion)*. Association for Computing Machinery, New York, NY, USA, 166–170. https://doi.org/10.1145/3191697.3214323

[29] Yue Zhu, Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, Muhib Khan, and Weikuan Yu. 2018. Direct-FUSE: Removing the Middleman for High-Performance FUSE File System Support. In *Proceedings of the 8th International Workshop on Runtime and Operating Systems for Supercomputers* (Tempe, AZ, USA) *(ROSS'18)*. Association for Computing Machinery, New York, NY, USA, Article 6, 8 pages. https://doi.org/10.1145/3217189.3217195