Trees, Walks, and Words: A Logic for File System Interactions

ERIC C. ZHAO, Brown University, USA
ZEKAI LI, Brown University, USA
LUKAS LAZAREK, Brown University, USA
FEIYANG WANG, Brown University, USA
KONSTANTINOS KALLAS, University of California, Los Angeles, USA
ROBERT Y. LEWIS, Brown University, USA
NIKOS VASILAKIS, Brown University, USA

Programs often interface with file systems for a variety of tasks, from retrieving and storing computation data to configuring the operating system. These interactions are mediated by *paths*, and the complex interplay between programs, paths, and hierarchical file system state can lead to unexpected behaviors ranging from silent failures to irreversible data loss and security vulnerabilities. Yet, this domain remains underserved by formal reasoning. This paper addresses that gap by developing foundations for automated formal analysis of program interactions with the file system. Grounding our study in a survey of APIs across several widely used standard libraries, we distill core file system behaviors into a representative object language, WHILE$_{FS}$. We then develop a theory of *tree-walking word equations* (TWE) and show that it is adequate to reason about file system interactions via sound and complete symbolic semantics for WHILE$_{FS}$. We demonstrate that the satisfiability question for TWE is as hard as the longstanding open problem of word equations with length constraints. Nonetheless, we also show that two useful fragments remain decidable in PSPACE by reduction to standard word equation theories. Together, these results demonstrate that TWE and WHILE$_{FS}$ constitute a rich foundation for formal reasoning about programs that interact with file systems.

1 INTRODUCTION

Interactions with file systems are pervasive in software programs, which routinely read from and write to files and directories to store and retrieve user data, cache intermediate computations, read configuration files, and interface with other system resources such as I/O channels and devices. File systems are used across every layer of the software stack, from shell scripts and package managers to compilers, databases, and web servers, and the standard libraries of most programming languages expose dedicated APIs for file system operations.

The interplay between program execution, file system state, and the *paths* used to mediate between them is often subtle and gives rise to substantial complexity in reasoning about these programs. As a result, developers and agents often fail to reason effectively about the file system, producing and executing software with bugs ranging from silent failures to dangerous behaviors such as irrecoverable data loss. Indeed, file system–related subtleties underlie notable catastrophic incidents: for example, a Steam installation update script, run in an unexpected context, inadvertently computed an empty installation path and invoked `rm -rf` to delete all user-writable data under the root directory [68]. In other cases, an agentic coding tool asked to organize a user's desktop instead deleted an archive of family photographs [44], and npm v5.7.0 irrecoverably botched entire system installations [19]. Misuse of the file system can also expose security vulnerabilities; thousands of known CVEs in high-profile software are linked to improper file system path handling [54, 55], such as the "Zip Slip" vulnerability [64]. Despite the evident challenge and irreversible consequences of such bugs, this domain remains underserved by formal reasoning techniques:

Authors' addresses: Eric C. Zhao, Brown University, Providence, USA; Zekai Li, Brown University, Providence, USA; Lukas Lazarek, Brown University, Providence, USA; Feiyang Wang, Brown University, Providence, USA; Konstantinos Kallas, University of California, Los Angeles, Los Angeles, USA; Robert Y. Lewis, Brown University, Providence, USA; Nikos Vasilakis, Brown University, Providence, USA.

most prior work is limited to specific kinds of programs (*e.g.*, no file system modifications, or no first-class path values) or does not reason about the file system's hierarchical structure (§7).

In this work, we introduce a framework for reasoning formally about file system interactions, laying the theoretical foundations for automated analyses. The key elements are (1) a core model of the file system and programs interacting with it (expressed as an object language WHILE$_{FS}$), and (2) a theory of *tree-walking word equations* (TWE) to express fundamental facts and queries about possible file system states during program execution.

We ground the framework and WHILE$_{FS}$ through a survey of the file system APIs of the standard libraries of widely used languages and distill key data abstractions and universal operators into a WHILE-like imperative language (§3). As a core semantics akin to Featherweight Java [38] or λJS [33], WHILE$_{FS}$ provides a shared basis atop which to formalize and compare additional, dedicated semantics capturing variations in execution model, error-handling, and platform-specific behavior (see §8). Nevertheless, this core is sufficient to capture many real-world incidents.

We demonstrate that the theory of tree-walking word equations (TWE) is a suitable base theory of paths and file systems by giving a sound and complete symbolic semantics for WHILE$_{FS}$, capturing the space of possible file system states and behaviors using TWE-formulae (§4). The symbolic semantics provide a basis for verification and bug-finding using TWE and can be used to witness real-world bugs and vulnerabilities, including several of the incidents mentioned above. Automated reasoning depends, however, on decision procedures for the satisfiability of TWE-formulae. Though the theory is small, it turns out that fundamental interactions arising from file system path behavior make decidability hard. In fact, path behaviors can encode the theory of word equations with length constraints (WE + LEN), whose decidability is a longstanding open problem [13, 28] (§5).

Although the full logic's decidability is therefore an open question, these encodings, which carefully exploit upward path traversals, also direct us to useful, decidable fragments. In particular, we can establish decision procedures for satisfiability in *linear* and *bounded* fragments of TWE, which limit upward traversals (§6). Since such upward traversals are often restricted in practice, these fragments provide a starting point for rigorous automated analyses of common classes of programs. We show reductions of TWE$_0$ and TWE$_k$ to standard theories of word equations, which enjoy decidability and efficient support in industrial solvers such as Z3 [21] and cvc5 [8]. Using bounds as low as $k = 1$, we can automatically discover the same real-world bugs.

Contributions. We begin with an informal exposition of challenges in reasoning about paths and hierarchical file systems (§2). The following sections describe the paper's contributions:

- A systematic survey of several real file system libraries and a representative object language WHILE$_{FS}$ distilling core file system behaviors and interactions (§3).
- A quantifier-free theory of *tree-walking word equations* (TWE) that can be used to reason about WHILE$_{FS}$, as demonstrated with a sound and complete symbolic semantics (§4).
- A proof that the satisfiability question for TWE is at least as hard as that of word equations with length constraints, a longstanding open problem (§5).
- Proofs of decidability for two restricted yet useful fragments of TWE (§6).

All results are fully mechanized in Lean 4 [22], in our artifact [72]. We conclude with a discussion of related work (§7) and future directions to apply and extend the framework (§8).

2 UNDERSTANDING PATHS AND HIERARCHICAL FILE SYSTEMS

In this section, we consider three different perspectives on paths, each of which illustrates one additional facet of their complexity. First, we see paths as addresses in the file system, similarly to how pointers refer to memory heap cells (§2.1). Then, we uncover path resolution, which treats paths

```
rmr(path) ≜
1   st := stat(path)
2
3   if st == ● then
4     unlink(path)
5   else if st == ○ then
6     ds := opendir(path)
7     while ds != [] do
8       name, ds := readdir(ds)
9       rmr(join(path, name))
10    done
11    rmdir(path)
```

```
rmr(/app)
  ① = stat(/app)
  [config, db] = opendir(/app)
  config, [db] = readdir([config, db])
  rmr(/app/config)                          ┐ for config
    ③ = stat(/app/config)                   │
    [] = opendir(/app/config)               │
    rmdir(/app/config)                      ┘
  db, [] = readdir([db])                    ┐ for db
  rmr(/app/db)                              │
    ❹ = stat(/app/db)                       │
    unlink(/app/db)                         ┘
  rmdir(/app)
```

Listing 1. **A simplified implementation `rmr` of recursive deletion from busybox [14] (left) and the execution trace of `rmr(/app)` on the tree in figure 1 (right).** The execution trace tracks the sequence of calls to file system APIs across recursive calls to `rmr`. Note that while `stat` actually returns a summary of the node at the specified location, for convenience we show the actual node (*e.g.*, ①) from figure 1.

as procedures that *walk* the file system to their destinations, imposing constraints on the whole sequence of visited nodes (§2.2). Finally, we introduce programs into the mix, which manipulate paths as sequences and generate sequence-level constraints in addition to file system ones (§2.3).

2.1 Paths are addresses

Paths locate data stored in the file system, similarly to how pointers locate memory in the heap. However, rather than a flat array the file system is usually hierarchical, resembling a tree of named locations such as the one shown in figure 1. Each location is either *absent* or has a *node*. Each node is a *directory* ○, which consists of named child *entries*, or a *file* ● at a leaf position, which holds data such as byte content. The tree starts at the *root directory* (⓪ in figure 1) and a path identifies a location in the tree using a *sequence of names* separated by a delimiter (*e.g.*, `/`).

Listing 1 (left) showcases paths as addresses through a simplified version of the implementation of `rm -r` from busybox [14]—a popular implementation of Unix coreutils—that recursively deletes directories and files at a given `path`. Suppose that we call `rmr(/app)` for the file system tree in figure 1. On line 1, `stat` retrieves the node at `path`. In this case, since `/app` locates the directory ①, the program iterates through its entry names via a directory stream and recursively invokes `rmr` for each one. The program constructs the recursive call paths using the *path expression* `join(path, name)` that extends `path` with `name`.

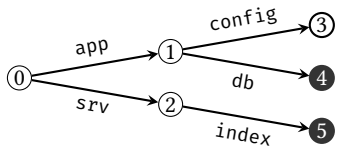


Figure 1. Tree representing a file system.

2.2 Paths are procedures

The address view captures the purpose of paths, but hides the actual process with which a path indexes into the file system. This process is called *path resolution*, and instead of directly indexing a node, it recurses over the name sequence traversing tree nodes. That process can *fail* if an intermediate node is not a directory but rather a file or absent. Resolution succeeds only if the entire path is processed without failure, implicitly constraining all *ancestor* nodes to be directories. For instance, successful resolution of `/app/db` implies that the nodes at `/` and `/app` are directories.

The picture is complicated by the *parent reference*, "`..`", that walks up instead of down the tree: *e.g.*, resolving `..` from ③ in figure 1 traverses ①. Parent references introduce *aliasing*, since a node can now be identified by infinitely many paths: ② in figure 1 is identified by `/srv`, `/app/../srv`, `/app/../app/../srv`, and so on. Note that resolution *saturates* when resolving `..` at the root directory: *e.g.*, `/`, `/..`, and `/../..` all resolve to the root. File system APIs often provide "normalization" operations that eliminate parent references to derive a single *linear* path to a node. For example, `/app/../srv` can be normalized to `/srv` by eliminating the pair `app/..`. Note that syntactic normalization *does not* strictly preserve the behavior of paths, since it might hide a path resolution failure: *e.g.*, while `/home/../srv` fails, its normalized version (`/srv`) *succeeds*.

In light of path resolution, a path can be understood as a *procedure*: while an address describes *where* a location is, a path dictates *how* to reach it by specifying a *walk* along the file system tree. For example, resolving `/app/../srv` on the tree in figure 1 yields the successful walk ⓪ → ① → ⓪ → ②. We call the last node the *target* and the preceding sequence the *footprint* of the walk.

Ignoring the procedural perspective can lead to bugs. Listing 1 does not account for paths as procedures, causing it to misbehave on inputs with `..` components. Consider `rmr(/app/config/..)` on the same figure 1 tree, which should remove ① just like `rmr(/app)`. Until the second recursive call, the trace (listing 2) closely resembles the happy path. However, `stat(/app/config/../db)` fails, because resolving the path means following `config` from ①, but it was deleted by the first recursive call. The program fails silently and leaves ① *partially* removed.

```
rmr(/app/config/..)
  ① = stat(/app/config/..)
  [config, db] = opendir(/app/config/..)
  config, [db] = readdir([config, db])
  rmr(/app/config/../config)
    ③ = stat(/app/config/../config)
    [] = opendir(/app/config/../config)
  rmdir(/app/config/../config)
  db, [] = readdir([db])
  rmr(/app/config/../db)
    ✕ = stat(/app/config/../db)
```

Listing 2. **The failing execution trace for `rmr(/app/config/..)` on the tree in figure 1.**

Improper accounting for walks can also lead to security vulnerabilities, such as faulty access control and the "Zip Slip" vulnerability mentioned above (see CWE-22 [54] and 23 [55], which record over 2,700 high or critical severity CVEs). A common pattern for restricting application access to directories and files is to check path prefixes, *e.g.*, limiting access to `/srv` only by checking `isPrefix(/srv, path)`. However, these checks can be bypassed with `..` (*e.g.*, `/srv/../app/db`).

2.3 Paths are dynamic

An additional layer of complexity is that programs can arbitrarily manipulate paths, by building them up from sequences (*e.g.*, `join`), decomposing, comparing (*e.g.*, `isPrefix`), and passing them around as ordinary data. To statically reason about a program therefore means to reason about constrained (or unconstrained) sets of paths, how they are transformed by sequence operations, and whether the walks they describe on the possible sets of file system states fail or succeed.

Listings 1 and 2 illustrate this complexity concretely through the program's combination of control flow, syntactic path construction, and file system mutation. The behavior of the program depends on the combination and interaction of constraints across paths, walks, file system states, and control flow. The bug can only be identified by simultaneously tracking constraints at (1) the syntactic level (the construction of `/app/config/../config` via `join`), (2) file system state (the absence of `/app/config` on the second iteration), (3) control flow (the loop over entries of the directory), and (4) walks (which fail for `/app/config/../db` once `/app/config` is deleted).

In the rest of the paper, we develop foundations to formally reason about such programs and their intertwined complexity. Indeed, we model these kinds of programs (§3) and show that the

bugs above can be witnessed by symbolic execution traces based on satisfiability of formulae in the theory of TWE (§4, see example 8). The satisfiability question is hard in general (§5) yet decidable in cases of bounded upward traversal such as this one (§6, see example 10). Note that we have avoided addressing many additional features present in real file systems, such as links, mounts, overlays, and access control. Even without these, we see significant fundamental complexity in reasoning about file system interactions, so we defer the study of such features for future investigation.

3 SURVEY AND CORE OF FILE SYSTEM INTERACTIONS

To ground formal analysis, we first conduct a systematic survey of file-related APIs across the standard libraries for several widely used programming languages (§3.1). The survey reveals a stable *core* of abstractions and operators via which programmers interface with the file system (§3.2), and this core informs the design of an archetypal object language WHILE$_{FS}$ (§3.3) which will serve as the basis for the remainder of the paper.

3.1 Survey

We survey the standard libraries of G glibc 2.41, P Python 3.13, N Node.js 24.1, and R Rust 1.96, four widely used language ecosystems with well-documented APIs, examining official documentation for abstractions and operations which interface with the file system. From the individual APIs we identify primitive behaviors, organize them into loose families, and distinguish the data abstractions [α] on which they operate. A single API function may couple several different kinds of primitive behaviors, and each behavior may operate on several different data abstractions. For instance, `fopen` G both opens and creates files depending on the arguments passed: with mode `"r"` it opens an existing file, whereas with `"w"` it also creates the file if it is absent. The survey distinguishes these behaviors and records `fopen` G as exhibiting both. We do not consider concurrency (*e.g.*, atomicity and thread-safety), synchronicity (*e.g.*, synchronous vs. asynchronous execution), platform-specific variation (*e.g.*, case-sensitivity and other conventions), or differences in error-handling, and leave these aspects for future work atop WHILE$_{FS}$'s core (see §8).

The complete results comprise 524 file system–related APIs capturing 128 primitive behaviors operating on 20 data abstractions across 33 behavior families. Table 1 gives a compact summary of these behavior families, collected loosely into 4 groupings; the complete results can be found in the supplementary material (Appendix A).

Paths. Locations in the file system are addressed by *paths* [P], which are sequences of components. The libraries provide specific operators to manipulate paths syntactically: *e.g.*, composition primitives concatenate paths with the delimiter (*e.g.*, `PurePath.joinpath` P, `path.join` N), while decompositions extract the last component (*e.g.*, `basename` G, `PurePath.name` P, `path.basename` N) or the prefix sub-path (*e.g.*, `dirname` G, `PurePath.parent` P, `path.dirname` N). More complex decompositions may further extract stems or extensions from the basename (*e.g.*, `PurePath.stem` P, `PurePath.suffix` P, `path.parse` N). Other syntactic operators compute the relative path between two paths (*e.g.*, `PurePath.relative_to` P, `path.relative` N), common sub-paths (*e.g.*, `os.path.commonpath` P), or normalizations (*e.g.*, `path.normalize` N, `os.path.normpath` P, `Path::components` R). Some libraries also convert between path values and encoded or string representations (*e.g.*, `os.fsencode` P, `Path::to_str` R). Relative paths connect path syntax to ambient process state, where the current working directory is prepended when no base directory is supplied explicitly (*e.g.*, `getcwd` G, `chdir` G, `Path.cwd` P).

File system tree. A path is used to access data in the *file system tree* [t]. Tree-level operators create (*e.g.*, `link` G, `mkdir` G, `fs.mkdirSync` N), delete (*e.g.*, `unlink` G, `rmdir` G, `Path.unlink` P), rename (*e.g.*, `rename` G, `Path.rename` P), and traverse (*e.g.*, `opendir` G, `fs.opendirSync` N) nodes within

Table 1. **Summary of the library survey.** Each family of primitive behaviors is shown with the # of behaviors and the # of distinct APIs in the library which realize at least one of the behaviors (or – if none do): G glibc 2.41, P Python 3.13, N Node.js 24.1, R Rust 1.96, Σ all; and the data abstractions on which it operates: p path; t file tree; y file type; b block device; h character device; o FIFO; k socket; s directory stream; c file content; x file descriptor; a file storage; m file metadata; r file permissions; i file timestamps; l soft link; g glob pattern; f temporary file; w current working directory; e environment; q string.

behavior family	#	G	P	N	R	Σ	abs.
total	128	99	163	121	141	524	
paths	25	3	34	13	39	89	
path decomposition	12	3	19	6	19	47	p
path composition	3	–	5	4	11	20	p q
path conversion	4	–	3	1	8	12	p q
path computation	5	–	5	1	3	9	p
path normalization	1	–	3	2	2	7	p
tree-level	39	40	62	41	50	193	
file examination	6	4	18	9	18	49	p t y
file creation	4	9	9	4	10	32	p t r
directory traversal	11	10	7	8	5	30	p t y s
file deletion	3	4	8	3	3	18	p t
file copying	2	2	6	2	1	11	p t c
file renaming	2	2	6	1	1	10	p t
path canonicalization	1	2	2	2	2	8	p t
soft link examination	2	2	5	3	5	15	p t y l
special file examination	4	–	–	8	4	12	p t y
soft link creation	1	2	2	1	1	6	p t l
special file creation	3	3	2	–	–	5	p t y b h o k r
file-level	41	47	48	56	65	216	
file permissions	8	13	23	13	21	70	p t r
file timestamps	3	9	10	14	14	47	p t i
file access	9	11	11	15	8	45	p t c x a
file opening	4	8	7	13	16	44	p t c x r
file metadata	3	4	5	10	15	34	p t x m
file synchronization	4	4	3	2	2	11	t c x
file descriptors	5	6	3	–	1	10	s c x
file locking	3	2	–	–	5	7	t c x
file comparison	2	–	2	–	–	2	p t c
extra	23	18	33	19	2	72	
process state	4	5	7	1	2	15	p r w
file system metadata	3	2	4	9	–	15	p t x a
temporary files	4	6	7	1	–	14	p t c x r f
tree traversal	4	4	6	1	–	11	p t r g
pattern matching	1	1	3	1	–	5	p g
process environment	3	–	4	1	–	5	p t e q
URI manipulation	2	–	2	2	–	4	p q
tree watching	2	–	–	3	–	3	p t

the tree. The libraries also feature semantic path manipulation operators which normalize paths with respect to the actual tree (*e.g.*, `realpath` G, `Path.resolve` P). Beyond regular files and directories, nodes may take special types—soft links, FIFOs, sockets, and block or character devices—which the libraries create (*e.g.*, `symlink` G, `os.mknod` P) and examine (*e.g.*, `Path.readlink` P, `Stats.isSocket` N). Nodes in the file tree may carry additional data: for instance, *permissions* r restricting user access to the node (*e.g.*, `chmod` G, `Path.chmod` P); *timestamps* i recording creation, modification, and access events (*e.g.*, `utime` G, `fs.utimesSync` N); and, in the case of regular files, *content* payload c in raw bytes (*e.g.*, `Path.read_text` P, `fs::read` R).

Extra. The behaviors under *extra* are heterogeneous: allocation of temporary files f (*e.g.*, `mktemp` G); querying file system metadata (*e.g.*, `statvfs` G, `fs.statfsSync` N); glob-based pattern g matching on paths (*e.g.*, `path.matchesGlob` N); conversion between paths and URIs (*e.g.*, `Path.as_uri` P); variable expansion (*e.g.*, `Path.expanduser` P); and access to ambient process state such as the current working directory (*e.g.*, `getcwd` G, `chdir` G) or process root (*e.g.*, `unix::fs::chroot` R).

3.2 An API core with enrichments

Through the survey we identify a core subset of behavior families (*paths* and most of *tree-level*) that is expressive enough to describe many programs that interact with the file system, by manipulating

paths and using them to access and mutate the file system tree. The remaining behaviors relate to this baseline in one of three ways. Most augment the data at *nodes* within the tree, some enrich the tree *structure* itself, and others are *peripheral* beside the core.

3.2.1 The core. Paths are sequences of components and hence compose via concatenation and factor into dirname and basename.

$$\text{join} \;:\; \text{Path} \times \text{Path} \to \text{Path} \qquad\qquad \text{dir}, \text{base} \;:\; \text{Path} \to \text{Path}$$

Every family of behaviors is mediated by the resolution of a path against the file system tree—the interpretation of a path to reach a location from which file system data is accessed and mutated. This triad is fundamental to nearly all behaviors: the file system is, at its core, the apparatus to manipulate a tree via the resolution of paths.

The tree is a named hierarchy of directories and files, equipped with primitive actions to query, create, delete, and move nodes, and to read through the entries of a directory:

$$\begin{array}{rcl@{\qquad}rcl}
\text{stat} & : & \text{Tree} \times \text{Path} \rightharpoonup \text{FileType} & \text{rename} & : & \text{Tree} \times \text{Path} \times \text{Path} \rightharpoonup \text{Tree} \\
\text{create}, \text{mkdir} & : & \text{Tree} \times \text{Path} \rightharpoonup \text{Tree} & \text{opendir} & : & \text{Tree} \times \text{Path} \rightharpoonup \text{Entries} \\
\text{unlink}, \text{rmdir} & : & \text{Tree} \times \text{Path} \rightharpoonup \text{Tree} & \text{readdir} & : & \text{Tree} \times \text{Entries} \rightharpoonup \text{Name} \times \text{Entries}
\end{array}$$

Following the terminology used in §2, each operator (except for readdir) resolves each path to a *location*, representing some (possibly absent) position in the tree.

As seen in §2.2, normalization simplifies the path into a form consisting of only ordinary names, that is, free of `..`, `.`, and the empty component. These are the canonical (simple) paths to locations in the tree. In fact, normalization *is* the mapping from paths to locations, *exposing path resolution as a user-facing operator*. However, normalization comes in two forms with complementary guarantees. *Syntactic* normalization (*e.g.*, `path.normalize` N, `Path::normalize_lexically` R) rewrites a path purely as syntax, eliminating non-linear components without consulting the tree; it is thus *total*, but the result need not resolve to the same location, or even resolve at all. *Semantic* normalization (*e.g.*, `realpath` G, `Path.resolve` P, `fs.realpath` N, `fs::canonicalize` R) instead resolves the path against the tree and reconstructs the canonical path to the target; it is *partial*—it may fail when the path does not resolve—but faithful, always yielding the canonical path to the same target.

$$\text{norm} \;:\; \text{Path} \to \text{Path} \qquad\qquad \text{realpath} \;:\; \text{Tree} \times \text{Path} \rightharpoonup \text{Path}$$

3.2.2 Enrichments of the core. *Per-node* enrichments extend the core with additional data attached to individual nodes. These include permissions, timestamps, and file content, with families of operators to query and mutate each. Hard links allow files to share an identity and therefore content and metadata. Additional examples include device, FIFO, and socket nodes.

Other enrichments are not localizable to individual nodes but alter the effective positional relationships between locations. Soft links, for instance, redirect resolution to follow the linked path before continuing with the original path. Similarly, mounts, overlays, and chroot each attach another file system at a specific path, so that resolution crossing a mount transparently descends into the grafted tree. Unlike node-level enrichments, these additions affect *where* paths resolve and give the tree a much more complex resolution structure admitting cycles and aliasing among linear paths.

The remaining behaviors sit at the periphery of the core. Some express auxiliary conveniences (*e.g.*, globbing, temporary files, and URIs), while others reflect ambient state relevant to program–file system interaction (*e.g.*, file descriptors). The current working directory facilitates use of relative paths, and certain environment variables (*e.g.*, `HOME`) usually give paths to file system resources.

Name	a			Directory	Δ	$\in$	$\mathcal{P}_{\text{fin}}(\text{Name})$
Component	q	$::=$	$a \mid \epsilon \mid . \mid ..$	Val	v	$::=$	true | false | p | y | Δ
Path	p	$::=$	$\varepsilon \mid p/q$	Expr	e	$::=$	$x \mid v \mid e = e \mid !e \mid e \,\&\&\, e \mid e \,\|\|\, e$
FileType	y	$::=$	file | dirc			$\mid$	join(e, e) | base(e) | dir(e) | norm(e)

Command	c	$::=$	skip | $x \coloneqq e$ | $c \fatsemi c$ | if e then c else c | while e do c done
		$\mid$	create(e) | mkdir(e) | unlink(e) | rmdir(e) | rename(e, e)
		$\mid$	$x \coloneqq$ stat(e) | $x \coloneqq$ realpath(e) | $x \coloneqq$ opendir(e) | x, $y \coloneqq$ readdir(e)

Figure 2. Syntax of WhileFS.

3.3 The WhileFS language

The core identified from the survey informs the design of an archetypal programming language that faithfully captures essential, primitive file system behaviors and can serve as the basis for further study. Described in figure 2, WhileFS extends a standard imperative While-style base with paths and file trees. Each construct is motivated by a core behavior from the survey. Commands operate over a program state consisting of a variable store ρ, which maps variables to values, and a file tree τ. Each file system command evaluates arguments, resolves paths, and mutates the file tree.

3.3.1 Paths and expressions. We model paths p as (possibly empty) sequences of components q, each of which is either a name a, the empty component ϵ, the self reference '.', or the parent reference '..'. They are composed and decomposed via primitive expressions join, base, dir, and norm, which reflect the core syntactic path behaviors discussed above. Paths can also be compared for syntactic equality via $p_1 = p_2$. Besides paths, other values include booleans, file types file and dirc, and directory entry name sets Δ. In WhileFS, all paths are absolute, but relative paths could be accommodated by expansion to absolute ones using a current working directory.

Along with standard control flow constructs, these simple primitives are sufficient to model a wide variety of syntactic manipulations and checks.

Example 1 (`isPrefix` in WhileFS). Recall `isPrefix` from §2.2. Define `isPrefix(pr, p)`:

```
while p != pr && p != ε do p := dir(p) done; return p == pr
```

3.3.2 Commands and program state. WhileFS also provides nine file system commands, reflecting the core described above. As noted in §3.1, these commands mirror primitive behaviors and thus often resemble but do not correspond exactly to library functions or system calls. The file system state over which commands operate is implicit in the program state rather than passed explicitly. Formally, a program state ς is a pair $\langle \rho, \tau \rangle$ with a variable store ρ and a file tree τ.

File trees. As in figure 3, we model file trees inductively. A file tree τ is a file or a (dirc δ) with the entries δ, a finite mapping of names to child trees. This name-centric model diverges from common file system *implementations* (which are often inode- or file-ID-centric) to directly reflect the *program*'s view of the file system, which it accesses via paths and names.

To facilitate local updates on a tree, we adopt standard machinery using one-holed contexts [1, 52]. In this case, a context C is a file tree with a single hole $\square$ at a particular location, and filling the hole with a subtree τ, the focus, recovers a tree $C[\tau]$. In essence, the context and focus together represent a tree with some subtree picked out. A context may also be filled with $\perp$ to erase the hole without planting a subtree. In the definition of $C[f]$, we write $\delta[a \mapsto \tau]$ to denote an update to the map.

Successful resolution of a path returns a one-holed context C for the target's location and the *focus* f—the subtree at that location, or $\perp$ if there is none. Since .. within a path may dictate an upward traversal, to model path resolution we use *zippers* à la Huet [37], which encode one-holed

$$\begin{array}{llll} \text{FileTree} & \tau & ::= & \text{file} \mid \text{dirc}\,\delta \\ \text{FileEntries} & \delta & \in & \text{Name} \rightharpoonup_{\text{fin}} \text{FileTree} \end{array} \qquad \begin{array}{llll} \text{FTCtx} & C & ::= & \square \mid \text{dirc}\,\delta[a \mapsto C] \\ \text{FTFocus} & f & ::= & \tau \mid \bot \end{array}$$

$C[f] \in \text{FileTree}$ *Fill the hole in* C *with* f.

$$\begin{array}{rcl} \square[\tau] & \triangleq & \tau \\ (\text{dirc}\,\delta[a \mapsto \square])\,[\bot] & \triangleq & \text{dirc}\,\delta \\ (\text{dirc}\,\delta[a \mapsto C])\,[f] & \triangleq & \text{dirc}\,\delta[a \mapsto C[f]] \end{array}$$

$\text{resolve}\,p\,\tau \in \text{FTCtx} \times \text{FTFocus}$ *Resolve* p *in* τ.

$$\text{resolve}\,p\,\tau \triangleq \langle \text{ctx}\,z, \text{focus}\,z \rangle \quad \text{iff traverse}\,p\;(\text{zip}\,\tau) = z$$

Figure 3. File trees, one-holed contexts, and path resolution.

contexts in an inverted fashion, maintaining upward structure from the hole to facilitate movement to both the parent and children. In this setting, a zipper z is a stack of *frames* encoding a context in layers down to the root, paired with the focus extracted from the hole. resolve $p\,\tau$ constructs a zipper at the root of τ, then traverses through the tree along each component, first checking that the focus is on a directory, then going down to a child for names, going up to the parent for .., and staying put for . and ϵ (see Appendix C for exact definitions). If the target location is successfully reached, we reify the zipper back into a context (ctx z) and extract the focus (focus z).

Small-step semantics. Figure 4 gives the operational semantics for file system–related operators and commands. $e \Downarrow^{\rho} v$ denotes evaluation of expressions under the variable store. join, base, and dir are straightforward, and norm proceeds by folding over the path and popping off the last name for each '..'. For commands c, $(c, \varsigma) \longrightarrow^{1} (c', \varsigma')$ and $(c, \varsigma) \longrightarrow^{*} (c', \varsigma')$ denote single and many steps.

The semantics for file system commands employ resolve for path resolution and context-filling for mutation. For instance, create(e) resolves the path (evaluated from e) to a context with an absent focus and fills the hole with a new file node. mkdir, unlink, rmdir, and stat follow similarly with different focus checks and fills. rename(e_1, e_2) resolves the source path to a non-$\bot$ (*i.e.*, file or dirc) focus, erases the hole by filling with $\bot$, resolves the destination path to a $\bot$ focus, and fills that hole with the source's subtree. $x \coloneqq$ realpath(e) resolves the path then reifies the target context C into a canonical path. Finally, $x \coloneqq$ opendir(e) sets x to the set of entry names dom δ, and $x,\ y \coloneqq$ readdir(e) selects a name out of the set, unless it is empty.

Example 2 (rmr). Listing 1 is a WhileFS program, whose semantics match the executions above.

4 TWE: A THEORY FOR FILE SYSTEM REASONING

This section develops a theory to reason about the core operations distilled in WhileFS (§3). Introduced in §4.1, TWE draws upon the observations identified from the survey to enable reasoning over interactions between paths and the walks they induce. Central to the theory are *tree-walking word equations*, with further ingredients to support path normalization and directory reasoning. To enable mutations to the tree, TWE supports functional updates similarly to standard theories of arrays. §4.2 then demonstrates its adequacy for the WhileFS core through a symbolic semantics which emits constraints in the theory. The symbolic semantics are sound and complete with respect to concrete executions (theorems 1 and 2), and symbolic traces can capture real bugs such as the failing execution of rmr in listing 2. Note that we do not develop a full program logic nor claim whole-program specification; TWE is an underlying layer capturing the constraint problem of path and file tree behavior. Even so, the theory turns out to be quite hard; §5 shows that decidability is at least as difficult as longstanding open problems. §6 identifies decidable fragments.

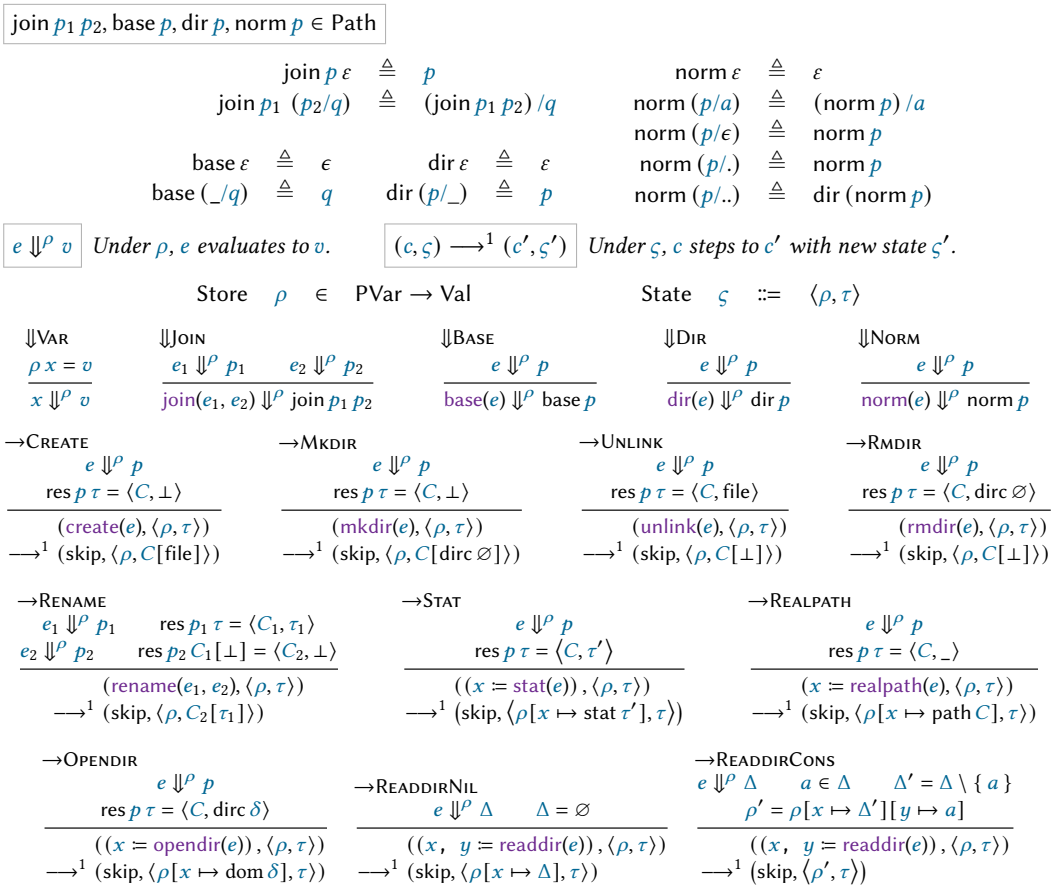


Figure 4. **Operational semantics for WhileFS.** We elide the rules for standard constructs (see Appendix E.2).

4.1 A theory of tree-walking word equations with updates

TWE (figure 5) consists of quantifier-free formulae constructed over predicates on paths. First, we recall the standard theory of word equations (WE) and use it to reason syntactically about paths. Then, to reason about state and path resolution, we introduce *walks* over trees with updates.

4.1.1 Syntactic reasoning with word equations. Recall that a path p is a finite sequence over an infinite alphabet of components. In a program, the value of a path may not be known or fixed, so we also admit *path variables* X standing for unknown paths; in general, a *word* U is constructed by concatenating components and variables (figure 5), and these describe path expressions. For instance, the variable-free word `app` · `config` corresponds to the path `/app/config`, and `app` · X stands for paths of the form `/app/`X. The empty word ε corresponds to the root path `/`, which has no components. Since path manipulations (*e.g.*, base) cannot be evaluated directly when their operands' values are unknown, we reason about their behavior via *equations* between words $U = V$.

This theory of *word equations*, which we now briefly recall, is well-studied in the literature [40, 51, 57]. A *substitution* σ maps variables and, by extension, words to sequences of letters from a finite or infinite alphabet; we write $\sigma\, X$ and $[\![U]\!]_\sigma$, respectively. A substitution is a *solution* satisfying the equation $U = V$ if and only if $[\![U]\!]_\sigma = [\![V]\!]_\sigma$. An instance of the word equation problem asks

$$
\begin{array}{rrcl}
\text{Name} & A & ::= & X_A \mid a \\
\text{Comp} & B & ::= & X_B \mid \epsilon \mid . \mid .. \mid A \\
\text{Path} & P, Q & ::= & X \mid \varepsilon \mid B \mid P \cdot Q \\
 & & \mid & L \mid \mathsf{norm}\, P \\
\text{Names} & S & ::= & R \mid \varnothing \mid S \cup A \mid \mathsf{entries}\, T \\
\text{Subst} & \sigma & \in & \mathsf{Var} \to \mathsf{Path}
\end{array}
\qquad
\begin{array}{rrcl}
\text{Tree} & T & ::= & V \mid \mathsf{file} \mid \mathsf{dirc}_{\varnothing} \mid \bot \\
 & & \mid & T[L] \mid T[L \mapsto T] \\
\text{Shape} & H & ::= & \bot \mid \mathsf{file} \mid \mathsf{dirc} \\
\text{Prop} & \Phi & ::= & \cdots \mid P = Q \\
 & & \mid & P \twoheadrightarrow_T L \mid T \in H \mid S = S' \mid A \in S \\
\text{Forest} & \theta & \in & \mathsf{TreeVar} \to \mathsf{FTFocus}
\end{array}
$$

$$
\boxed{[\![P]\!]_{\sigma,\theta} \in \mathsf{Path}}
\qquad
\begin{array}{rcl}
[\![X]\!]_{\sigma,\theta} & \triangleq & \sigma\, X \\
[\![\epsilon]\!]_{\sigma,\theta} & \triangleq & \varepsilon \\
[\![B]\!]_{\sigma,\theta} & \triangleq & \cdots \\
[\![P \cdot Q]\!]_{\sigma,\theta} & \triangleq & \mathsf{join}\, [\![P]\!]_{\sigma,\theta}\ [\![Q]\!]_{\sigma,\theta} \\
[\![L]\!]_{\sigma,\theta} & \triangleq & \sigma\, L \\
[\![\mathsf{norm}\, P]\!]_{\sigma,\theta} & \triangleq & \mathsf{norm}\, [\![P]\!]_{\sigma,\theta}
\end{array}
$$

$$
\boxed{[\![S]\!]_{\sigma,\theta} = \mathcal{P}_{\mathsf{fin}}(\mathsf{Name})}
\qquad
\begin{array}{rcll}
[\![R]\!]_{\sigma,\theta} & \triangleq & \sigma\, R & \sigma\, R \text{ is a set} \\
[\![\varnothing]\!]_{\sigma,\theta} & \triangleq & \varnothing & \\
[\![S \cup A]\!]_{\sigma,\theta} & \triangleq & [\![S]\!]_{\sigma,\theta} \cup [\![A]\!]_{\sigma,\theta} & \\
[\![\mathsf{entries}\, T]\!]_{\sigma,\theta} & \triangleq & \mathsf{dom}\, \delta & [\![T]\!]_{\sigma,\theta} = \mathsf{dirc}\, \delta \\
[\![\mathsf{entries}\, T]\!]_{\sigma,\theta} & \triangleq & \varnothing & [\![T]\!]_{\sigma,\theta} \in \{\mathsf{file}, \bot\}
\end{array}
$$

$$
\boxed{[\![T]\!]_{\sigma,\theta} \in \mathsf{FTFocus}}
\qquad
\begin{array}{rcl}
[\![V]\!]_{\sigma,\theta} & \triangleq & \theta\, V \\
[\![\mathsf{file}]\!]_{\sigma,\theta} & \triangleq & \mathsf{file} \\
[\![\mathsf{dirc}_{\varnothing}]\!]_{\sigma,\theta} & \triangleq & \mathsf{dirc}\, \varnothing \\
[\![\bot]\!]_{\sigma,\theta} & \triangleq & \bot \\
[\![T[L]]\!]_{\sigma,\theta} & \triangleq & f \\
[\![T]\!]_{\sigma,\theta} = \tau\,;\, \mathsf{resolve}\, [\![L]\!]_{\sigma,\theta}\ \tau = \langle C, f \rangle & & \\
[\![T[L]]\!]_{\sigma,\theta} & \triangleq & \bot \\
[\![T]\!]_{\sigma,\theta} = \bot\,;\, [\![L]\!]_{\sigma,\theta} = \varepsilon & & \\
[\![T[L \mapsto T']]\!]_{\sigma,\theta} & \triangleq & C[[\![T']\!]_{\sigma,\theta}] \\
[\![T]\!]_{\sigma,\theta} = \tau\,;\, \mathsf{resolve}\, [\![L]\!]_{\sigma,\theta}\ \tau = \langle C, _ \rangle & &
\end{array}
$$

$$
\boxed{\mathsf{shape}\, f \in \mathsf{Shape}}
\qquad
\begin{array}{rcl}
\mathsf{shape}\, \bot & \triangleq & \bot \\
\mathsf{shape}\, \mathsf{file} & \triangleq & \mathsf{file} \\
\mathsf{shape}\, (\mathsf{dirc}\, \delta) & \triangleq & \mathsf{dirc}
\end{array}
$$

$$
\boxed{\langle \sigma, \theta \rangle \models \Phi}
\qquad
\begin{array}{lll}
\langle \sigma, \theta \rangle \models P = Q & \text{iff} & [\![P]\!]_{\sigma,\theta} = [\![Q]\!]_{\sigma,\theta} \\
\langle \sigma, \theta \rangle \models P \twoheadrightarrow_T L & \text{iff} & \mathsf{resolve}\, [\![P]\!]_{\sigma,\theta}\ [\![T]\!]_{\sigma,\theta} = \langle C, _ \rangle\,; \\
 & & \mathsf{path}\, C = [\![L]\!]_{\sigma,\theta} \\
\langle \sigma, \theta \rangle \models T \in H & \text{iff} & \mathsf{shape}\, [\![T]\!]_{\sigma,\theta} = H \\
\langle \sigma, \theta \rangle \models S = S' & \text{iff} & [\![S]\!]_{\sigma,\theta} = [\![S']\!]_{\sigma,\theta} \\
\langle \sigma, \theta \rangle \models A \in S & \text{iff} & [\![A]\!]_{\sigma,\theta} \in [\![S]\!]_{\sigma,\theta}
\end{array}
$$

Figure 5. The theory of tree-walking word equations with updates.

whether a system of equations has a satisfying substitution. Since words represent paths in our setting, we will write P and Q in lieu of U and V.

The behavior of many path operators can be specified using word equations.

Example 3 (join, base, and dir). The join operator corresponds exactly to concatenation of words (see ⇓SJoin in figure 6 below). $Y_B \coloneqq \mathsf{base}(P)$ can be expressed as a disjunction, depending on whether or not P is the root path: (1) if $P = \varepsilon$, then $Y_B = \epsilon$; *or* (2) if $P \neq \varepsilon$, then $P = X \cdot Y_B$, where X and Y_B are fresh path and component variables. dir is handled similarly (see ⇓SBase/Dir).

Here, component variables X_B denote single components (instead of sequences of components) and name variables X_A denote names only (and not ϵ, ., or ..). Equations also naturally express prefix conditions:

Example 4 (`isPrefix`). `isPrefix(`P`,` Q`)` from example 1 returns true when $P \cdot X = Q$, *i.e.*, $P \preceq Q$.

4.1.2 Tree reasoning with walks and shapes. When resolved against the file tree, a path induces a walk through the tree which, if successful, resolves to a location in the tree. In the semantics of WhileFS, locations were represented by one-holed contexts C (§3.3). In TWE, we distinguish locations L as canonical paths, equivalent to path C, and permit treatment of them as paths. To model path resolution, a *walk assertion* $P \twoheadrightarrow_T L$ asserts that P resolves to the location L in the tree T. The walk assertion does not claim anything about the node at the *target* location—which may be a file, directory, or absent—only that the path resolves successfully to the location. Recall that every *intermediate* location must, however, be a directory with the necessary children. A separate

shape assertion $T[L] \in H$ states that the location has the shape H, either file, dirc, or $\bot$ for absent. A tree term T is abstract (variables V), a file (file), an empty directory ($\text{dirc}_\varnothing$), absent ($\bot$), or the subtree at a location in another tree $T'[L]$.

Already we can partially characterize a wide range of programs using paths and the file tree.

Example 5 (unlink pre-condition). A successful invocation unlink(P) demands that P resolves to a location L on the current tree T and that the location is occupied by a file. Formally, the pre-condition is $(P \twoheadrightarrow_T L) \wedge (T[L] \in \text{file})$, or, in shorthand, $P \twoheadrightarrow_T L \in \text{file}$.

Whereas solutions for word equations range over substitutions, tree walks additionally quantify existentially over file trees. Precisely, as in figure 5, a model consists of a substitution σ and a *tree assignment* θ mapping each tree variable V to a focus, *i.e.*, a file tree τ or the absent $\bot$. The substitution also maps each location symbol L to a linear path describing a location. The model satisfies a walk assertion $P \twoheadrightarrow_T L$ if and only if the substituted path $[\![P]\!]_{\sigma,\theta}$ resolves successfully to the assigned location σL within the assigned tree $[\![T]\!]_{\sigma,\theta}$. Note that tree terms evaluate to *foci* rather than directly to file trees, since a tree term may stand for an absent focus.

4.1.3 Functional tree updates. To model mutations to file trees, it is necessary to support *updates* to trees. We adopt functional updates similarly to the theory of arrays à la McCarthy [53], generalizing the select/store discipline into tree-shaped versions. A selection $T[L]$ gives the subtree at L in T, and an update $T[L \mapsto T']$ gives a new tree from T by replacing the subtree at L with T'.

Example 6 (unlink update). The update $T[L \mapsto \bot]$ encodes unlink(P)'s deletion when $P \twoheadrightarrow_T L \in \text{file}$. With example 5, this gives a complete specification for successful unlink (see $\rightarrow$SUnlink in figure 6).

4.1.4 Tree entries. To allow reasoning over the entries of a directory, $\text{entries}\, T$ denotes a set S consisting of exactly the entry names at the root of the directory T. A set is abstract (variables R), empty ($\varnothing$), or the union with a name ($S \cup A$). TWE can test set equality $S = S'$, disequality $S \neq S'$, membership $A \in S$, and non-membership $A \notin S$. For simplicity, in models we reuse σ for set values.

Example 7 (opendir). Successful $R \coloneqq \text{opendir}(P)$, which resolves P and sets R to the set of entry names at that directory, is $(P \twoheadrightarrow_T L \in \text{dirc}) \wedge (R = \text{entries}\, T[L])$.

4.1.5 Normalization(s). Observe that with walks and equalities, it is possible to specify semantic normalization. For $Q \coloneqq \text{realpath}(P)$, we have the post-condition $(P \twoheadrightarrow_T L) \wedge (L = Q)$. However, *syntactic* normalization cannot be specified precisely by word equations, nor is it regular. We therefore admit normalization as a separate word operator $\text{norm}\, P$. Then, for $Q \coloneqq \text{norm}(P)$, we have the post-condition $Q = \text{norm}\, P$.

4.2 Symbolic semantics for WhileFS

To demonstrate that the logic of tree-walking word equations is adequate to reason about the file system core and WhileFS, we develop symbolic operational semantics (figure 6) which simulate the concrete semantics given in §3.3. The symbolic semantics generate conditions in the theory of tree-walking word equations and are sound and complete with respect to concrete execution steps (theorems 1 and 2). Using symbolic traces, we demonstrate several of the bugs discussed above.

4.2.1 Program state and judgments. Recall that each program state in concrete executions consists of the variable store ρ and the file tree τ. In the symbolic semantics, we maintain a symbolic variable store $\hat{\rho}$ in the standard way, and TWE assertions accumulated in the path condition $\hat{\pi}$ track the space of possible file trees. Since the tree undergoes mutations, we adopt functional updates via the tree-shaped select and store operators introduced above and track the tree term T corresponding

$$\widehat{\text{Val}} \quad \hat{v} ::= \Phi \mid P \mid T \mid S \qquad \widehat{\text{PC}} \quad \hat{\pi} \in \text{Prop}$$
$$\widehat{\text{Store}} \quad \hat{\rho} \in \text{PVar} \rightharpoonup_{\text{fin}} \widehat{\text{Val}} \qquad \widehat{\text{State}} \quad \hat{\varsigma} ::= \langle \hat{\rho}, T, \hat{\pi} \rangle$$

$e \Downarrow^{\hat{\rho}}_{\Phi} \hat{v}$ *Under* $\hat{\rho}$, e *evaluates to* $\hat{v}$, *generating* Φ. $(c, \hat{\varsigma}) \longrightarrow^1 (c', \hat{\varsigma}')$ *Under* $\hat{\varsigma}$, c *steps to* c' *with new state* $\hat{\varsigma}'$.

$$\Downarrow\text{SEqFileType}\ \frac{e_1 \Downarrow^{\hat{\rho}}_{\Phi_1} T_1 \quad e_2 \Downarrow^{\hat{\rho}}_{\Phi_2} T_2 \quad \Phi' = \left(\bigvee_H T_1 \in H \wedge T_2 \in H\right)}{e_1 = e_2 \Downarrow^{\hat{\rho}}_{\Phi_1 \wedge \Phi_2} \Phi'}$$

$$\Downarrow\text{SEqPath}\ \frac{e_1 \Downarrow^{\hat{\rho}}_{\Phi_1} P_1 \quad e_2 \Downarrow^{\hat{\rho}}_{\Phi_2} P_2}{e_1 = e_2 \Downarrow^{\hat{\rho}}_{\Phi_1 \wedge \Phi_2} P_1 = P_2}$$

$$\Downarrow\text{SJoin}\ \frac{e_1 \Downarrow^{\hat{\rho}}_{\Phi_1} P_1 \quad e_2 \Downarrow^{\hat{\rho}}_{\Phi_2} P_2}{\text{join}(e_1, e_2) \Downarrow^{\hat{\rho}}_{\Phi_1 \wedge \Phi_2} P_1 \cdot P_2}$$

$$\Downarrow\text{SBase}\ \frac{e \Downarrow^{\hat{\rho}}_{\Phi} P \quad \Phi' = (P = X \cdot Y_B) \vee (P = \varepsilon \wedge Y_B = \epsilon)}{\text{base}(e) \Downarrow^{\hat{\rho}}_{\Phi \wedge \Phi'} Y_B}$$

$$\Downarrow\text{SDir}\ \frac{e \Downarrow^{\hat{\rho}}_{\Phi} P \quad \Phi' = (P = X \cdot Y_B) \vee (P = \varepsilon \wedge X = \varepsilon)}{\text{dir}(e) \Downarrow^{\hat{\rho}}_{\Phi \wedge \Phi'} X}$$

$$\Downarrow\text{SNorm}\ \frac{e \Downarrow^{\hat{\rho}}_{\Phi} P}{\text{norm}(e) \Downarrow^{\hat{\rho}}_{\Phi} \text{norm}\, P}$$

$$\rightarrow\text{SCreate}\ \frac{e \Downarrow^{\hat{\rho}}_{\Phi} P \quad \text{SAT}(\hat{\pi}') \quad \hat{\pi}' = \hat{\pi} \wedge \Phi \wedge P \twoheadrightarrow_T L \in \bot}{(\text{create}(e), \langle \hat{\rho}, T, \hat{\pi} \rangle) \longrightarrow^1 (\text{skip}, \langle \hat{\rho}, T[L \mapsto \text{file}], \hat{\pi}' \rangle)}$$

$$\rightarrow\text{SMkdir}\ \frac{e \Downarrow^{\hat{\rho}}_{\Phi} P \quad \text{SAT}(\hat{\pi}') \quad \hat{\pi}' = \hat{\pi} \wedge \Phi \wedge P \twoheadrightarrow_T L \in \bot}{(\text{mkdir}(e), \langle \hat{\rho}, T, \hat{\pi} \rangle) \longrightarrow^1 (\text{skip}, \langle \hat{\rho}, T[L \mapsto \text{dirc}_{\varnothing}], \hat{\pi}' \rangle)}$$

$$\rightarrow\text{SUnlink}\ \frac{e \Downarrow^{\hat{\rho}}_{\Phi} P \quad \text{SAT}(\hat{\pi}') \quad \hat{\pi}' = \hat{\pi} \wedge \Phi \wedge P \twoheadrightarrow_T L \in \text{file}}{(\text{unlink}(e), \langle \hat{\rho}, T, \hat{\pi} \rangle) \longrightarrow^1 (\text{skip}, \langle \hat{\rho}, T[L \mapsto \bot], \hat{\pi}' \rangle)}$$

$$\rightarrow\text{SRmdir}\ \frac{e \Downarrow^{\hat{\rho}}_{\Phi} P \quad \text{SAT}(\hat{\pi}') \quad \hat{\pi}' = \hat{\pi} \wedge \Phi \wedge P \twoheadrightarrow_T L \in \text{dirc} \wedge \text{entries}\, T[L] = \varnothing}{(\text{rmdir}(e), \langle \hat{\rho}, T, \hat{\pi} \rangle) \longrightarrow^1 (\text{skip}, \langle \hat{\rho}, T[L \mapsto \bot], \hat{\pi}' \rangle)}$$

$$\rightarrow\text{SRename}\ \frac{e_1 \Downarrow^{\hat{\rho}}_{\Phi_1} P_1 \quad e_2 \Downarrow^{\hat{\rho}}_{\Phi_2} P_2 \quad \text{SAT}(\hat{\pi}') \quad \hat{\pi}' = \hat{\pi} \wedge \Phi_1 \wedge \Phi_2 \wedge P_1 \twoheadrightarrow_T L_1 \notin \bot \wedge P_2 \twoheadrightarrow_{T[L_1 \mapsto \bot]} L_2 \in \bot}{(\text{rename}(e_1, e_2), \langle \hat{\rho}, T, \hat{\pi} \rangle) \longrightarrow^1 (\text{skip}, \langle \hat{\rho}, T[L_1 \mapsto \bot][L_2 \mapsto T[L_1]], \hat{\pi}' \rangle)}$$

$$\rightarrow\text{SStat}\ \frac{e \Downarrow^{\hat{\rho}}_{\Phi} P \quad \text{SAT}(\hat{\pi}') \quad \hat{\pi}' = \hat{\pi} \wedge \Phi \wedge P \twoheadrightarrow_T L \notin \bot}{(x \coloneqq \text{stat}(e), \langle \hat{\rho}, T, \hat{\pi} \rangle) \longrightarrow^1 (\text{skip}, \langle \hat{\rho}[x \mapsto T[L]], T, \hat{\pi}' \rangle)}$$

$$\rightarrow\text{SRealpath}\ \frac{e \Downarrow^{\hat{\rho}}_{\Phi} P \quad \text{SAT}(\hat{\pi}') \quad \hat{\pi}' = \hat{\pi} \wedge \Phi \wedge P \twoheadrightarrow_T L \notin \bot}{(x \coloneqq \text{realpath}(e), \langle \hat{\rho}, T, \hat{\pi} \rangle) \longrightarrow^1 (\text{skip}, \langle \hat{\rho}[x \mapsto L], T, \hat{\pi}' \rangle)}$$

$$\rightarrow\text{SOpendir}\ \frac{e \Downarrow^{\hat{\rho}}_{\Phi} P \quad \text{SAT}(\hat{\pi}') \quad \hat{\pi}' = \hat{\pi} \wedge \Phi \wedge P \twoheadrightarrow_T L \in \text{dirc} \wedge R = \text{entries}\, T[L]}{(x \coloneqq \text{opendir}(e), \langle \hat{\rho}, T, \hat{\pi} \rangle) \longrightarrow^1 (\text{skip}, \langle \hat{\rho}[x \mapsto R], T, \hat{\pi}' \rangle)}$$

$$\rightarrow\text{SReaddirNil}\ \frac{e \Downarrow^{\hat{\rho}}_{\Phi} S \quad \hat{\pi}' = \hat{\pi} \wedge \Phi \wedge S = \varnothing \quad \text{SAT}(\hat{\pi}')}{(x,\ y \coloneqq \text{readdir}(e), \langle \hat{\rho}, T, \hat{\pi} \rangle) \longrightarrow^1 (\text{skip}, \langle \hat{\rho}[x \mapsto S], T, \hat{\pi}' \rangle)}$$

$$\rightarrow\text{SReaddirCons}\ \frac{e \Downarrow^{\hat{\rho}}_{\Phi} S \quad \hat{\pi}' = \hat{\pi} \wedge \Phi \wedge (S = R \cup X_A) \wedge (X_A \notin R) \quad \text{SAT}(\hat{\pi}')}{(x,\ y \coloneqq \text{readdir}(e), \langle \hat{\rho}, T, \hat{\pi} \rangle) \longrightarrow^1 (\text{skip}, \langle \hat{\rho}[x \mapsto R][y \mapsto X_A], T, \hat{\pi}' \rangle)}$$

Figure 6. **Symbolic operational semantics for WhileFS.** See Appendix G.3 for full rules.

to the state of the file tree. The symbolic program state $\hat{\varsigma}$ is therefore a triple $\langle \hat{\rho}, T, \hat{\pi} \rangle$. Mirroring the concrete semantics, we write $(c, \hat{\varsigma}) \longrightarrow^1 (c', \hat{\varsigma}')$ and $(c, \hat{\varsigma}) \longrightarrow^* (c', \hat{\varsigma}')$.

Consider the →SCreate rule from figure 6. First, we symbolically evaluate the argument e into a path term P alongside additional side constraints Φ to be accumulated into the path condition. Following example 5 for unlink above, the path must resolve to an absent location for create to succeed, *i.e.*, $P \twoheadrightarrow_T L \in \bot$. The step may be taken if the pre-condition is satisfiable. The mutation is then modeled by an update $T[L \mapsto \text{file}]$. The rules for mkdir and stat follow similarly, writing $P \twoheadrightarrow_T L \notin H$ as shorthand for $(P \twoheadrightarrow_T L) \wedge (T[L] \notin H)$. rename combines tree selection and updates to detach the subtree at the source location and graft it on at the target. Moving the subtree into

itself is disallowed since the pre-condition demands that P_2 resolves on an updated $T[L_1 \mapsto \bot]$. opendir returns entries $T[L]$ as mentioned above, and readdir pops from the set S if it is non-empty.

The symbolic semantics are sound and complete with respect to concrete executions. Every symbolic trace denotes only real executions; conversely, all real executions are captured by symbolic ones.

Theorem 1 (Soundness of symbolic semantics).
If $(c, \langle \hat{\rho}, T, \hat{\pi} \rangle) \longrightarrow^* (c', \langle \hat{\rho}', T', \hat{\pi}' \rangle)$, *then for every* $\langle \sigma, \theta \rangle \models \hat{\pi}'$, $\left(c, [\![\langle \hat{\rho}, T, \hat{\pi} \rangle]\!]_{\sigma,\theta}\right) \longrightarrow^* \left(c', [\![\langle \hat{\rho}', T', \hat{\pi}' \rangle]\!]_{\sigma,\theta}\right)$.

$$\begin{array}{ccc} (c, \hat{\varsigma}) & \longrightarrow^* & (c', \hat{\varsigma}') \\ {\scriptstyle [\![\cdot]\!]_{\sigma,\theta}}\downarrow & & \downarrow{\scriptstyle [\![\cdot]\!]_{\sigma,\theta}} \\ (c, [\![\hat{\varsigma}]\!]_{\sigma,\theta}) & \dashrightarrow^* & (c', [\![\hat{\varsigma}']\!]_{\sigma,\theta}) \end{array}$$

Theorem 2 (Completeness of symbolic semantics).
If $\langle \sigma, \theta \rangle \models \hat{\pi}$, *and* $\varsigma = [\![\langle \hat{\rho}, T, \hat{\pi} \rangle]\!]_{\sigma,\theta}$, *and* $(c, \varsigma) \longrightarrow^* (c', \varsigma')$, *then there exist* $\hat{\rho}'$, T', $\hat{\pi}'$, *and* $\langle \sigma', \theta' \rangle$ *such that* $(c, \langle \hat{\rho}, T, \hat{\pi} \rangle) \longrightarrow^* (c', \langle \hat{\rho}', T', \hat{\pi}' \rangle)$, $\langle \sigma', \theta' \rangle \models \hat{\pi}'$, *and* $\varsigma = [\![\langle \hat{\rho}, T, \hat{\pi} \rangle]\!]_{\sigma',\theta'}$, *and* $\varsigma' = [\![\langle \hat{\rho}', T', \hat{\pi}' \rangle]\!]_{\sigma',\theta'}$.

$$\begin{array}{ccc} (c, \hat{\varsigma}) & \dashrightarrow^* & (c', \hat{\varsigma}') \\ {\scriptstyle [\![\cdot]\!]_{\sigma,\theta}}\downarrow \; \top \; {\scriptstyle [\![\cdot]\!]_{\sigma',\theta'}} & & \top \; {\scriptstyle [\![\cdot]\!]_{\sigma',\theta'}}\downarrow \\ (c, \varsigma) & \longrightarrow^* & (c', \varsigma') \end{array}$$

4.2.2 Verification and bug-finding. The symbolic semantics form a basis for verification and bug-finding for WhileFS programs. Concretely, bug-finding involves finding execution traces in which some command fails to execute successfully—in other words, a pre-condition is not satisfied, *i.e.*, the *negation* of the pre-condition is satisfiable—and verification means to prove the absence of such traces. We can define operational rules for these failure cases, where commands step to error, and consider symbolic traces which step to error. For instance, the following pair of concrete and symbolic rules states that stat fails if the input path resolves to an absent location or fails altogether:

$$\rightarrow\textsc{StatFail} \quad \frac{e \Downarrow^{\rho} p \qquad \text{resolve}\, p\, \tau \neq \langle _, _ \rangle}{(x \coloneqq \text{stat}(e), \langle \rho, \tau \rangle) \longrightarrow^1 (\text{error}, \langle \rho, \tau \rangle)}$$

$$\rightarrow\textsc{SStatFail} \quad \frac{e \Downarrow^{\hat{\rho}}_{\Phi} P \qquad \text{SAT}(\hat{\pi} \wedge \Phi \wedge (P \twoheadrightarrow_T L \in \bot \vee P \not\twoheadrightarrow_T))}{(x \coloneqq \text{stat}(e), \langle \hat{\rho}, T, \hat{\pi} \rangle) \longrightarrow^1 (\text{error}, \langle \hat{\rho}, T, \hat{\pi} \rangle)}$$

$P \not\twoheadrightarrow_T$ means that P fails to resolve in T:

$$P \not\twoheadrightarrow_T \triangleq (P = Y \cdot Z) \wedge (Z \neq \varepsilon) \wedge (Y \twoheadrightarrow_T L' \notin \text{dirc}) \qquad Y, Z, L' \text{ fresh}$$

By theorems 1 and 2, a symbolic trace stepping to error always corresponds to a concrete failing execution, and all true failures are exhibited by some symbolic trace.

Example 8 (rmr bug). Recall the failing execution of `rmr(/app/config/..)` from listing 2, in which the command `stat(/app/config/../db)` expected a directory at `/app/config` after it had been deleted by `rmdir(/app/config/../config)`. Using →SStatFail, this execution is witnessed by the symbolic trace from the initial state $\langle \varnothing, V, \top \rangle$ in listing 3. In particular, after `rmdir`$(P \cdot X_A)$, where X_A is the first entry name of the directory at P, we have that $P \cdot X_A \twoheadrightarrow_V L_A$ with the tree $V' \triangleq V[L_A \mapsto \bot]$. The subsequent `readdir` yields the second entry name X_A', which must be different from the previous X_A. Then, `stat`$(P \cdot X_A')$ can fail since the condition of →SStatFail, in this case $\left(P \cdot X_A' \twoheadrightarrow_{V'} L_A' \in \bot\right) \vee P \cdot X_A' \not\twoheadrightarrow_{V'}$, is satisfiable. In particular, the right disjunct is satisfiable if P traverses L_A, *i.e.*, if $P = Y \cdot Z$ and $Y \twoheadrightarrow_{V'} L_A$ for some fresh Y and Z. Taking

$$\sigma\, Y = \texttt{app/config} \qquad \sigma\, Z = \texttt{..} \qquad \sigma\, P = \texttt{app/config/..} \qquad \theta\, V = \text{figure 1}$$

we recover exactly the scenario in listing 2.

Example 9 (Zip Slip). "Zip Slip" vulnerabilities in archive extraction software [64] allow attackers to write arbitrary files on the system. Extracting an attacker-controlled archive entry called Z from the

`rmr`(P)		
$V[L_P]$ = `stat`(P)	$P \twoheadrightarrow_V L_P \notin \bot$	(→SStat)
else if $V[L_P]$ == dirc **then**	$V[L_P] \in \mathrm{dirc}$	(→SIf)
R = `opendir`(P)	$R = \mathrm{entries}\, V[L_P]$	(→SOpendir)
while R != $\varnothing$ **do**	$R \neq \varnothing$	(→SWhileTrue)
X_A, R_1 = `readdir`(R)	$(R = R_1 \cup X_A) \wedge (X_A \notin R_1)$	(→SReaddirCons)
`rmr`$(P \cdot X_A)$		
$V[L_A]$ = `stat`$(P \cdot X_A)$	$P \cdot X_A \twoheadrightarrow_V L_A \notin \bot$	(→SStat)
else if $V[L_A]$ == dirc **then**	$V[L_A] \in \mathrm{dirc}$	(→SIf)
R_2 = `opendir`$(P \cdot X_A)$	$R_2 = \mathrm{entries}\, V[L_A]$	(→SOpendir)
done	$R_2 = \varnothing$	(→SWhileFalse)
`rmdir`$(P \cdot X_A)$	$V' \triangleq V[L_A \mapsto \bot]$	(→SRmdir)
while R_1 != $\varnothing$ **do**	$R_1 \neq \varnothing$	(→SWhileTrue)
$X_A{}', R_1'$ = `readdir`(R_1)	$(R_1 = R_1' \cup X_A{}') \wedge (X_A{}' \notin R_1')$	(→SReaddirCons)
`rmr`$(P \cdot X_A{}')$		
× = `stat`$(P \cdot X_A{}')$	$\left(P \cdot X_A{}' \twoheadrightarrow_{V'} L_A' \in \bot\right) \vee P \cdot X_A{}' \not\twoheadrightarrow_{V'}$	(→SStatFail)

Listing 3. **A symbolic trace for the failing execution of `rmr` in listing 2, from the initial state $\langle \varnothing, V, \top \rangle$ with path input P.** Each line shows the conjuncts introduced to the path condition by the corresponding rule. The condition for →SStatFail is satisfiable if $(P = Y \cdot Z) \wedge (Y \twoheadrightarrow_{V'} L_A)$.

directory Y may be understood as the trace $X \coloneqq \mathrm{join}(Y, Z)\fatsemi\mathrm{create}(X)$, which generates the equation $X = Y \cdot Z$ (from ⇓SJoin) and the assertion $X \twoheadrightarrow_T L$ (from →SCreate). It is possible for L to be outside Y, *i.e.*, $Y \not\preceq L$, since the accumulated condition $(X = Y \cdot Z) \wedge (X \twoheadrightarrow_T L) \wedge \left(Y \not\preceq L\right)$ is satisfiable, *e.g.*, take $Y \mapsto$ `home/me`, $Z \mapsto$ `../../etc/ssh/sshd_config`, and $L \mapsto$ `etc/ssh/sshd_config`. $\mathrm{isPrefix}(Y, X)$ is trivially satisfied and therefore a useless guard before $\mathrm{create}(X)$, but it becomes useful if X is first canonicalized with realpath. In this case, following →SRealpath and example 4, we have $Y \preceq L$, which makes $Y \not\preceq L$ impossible.

5 WE + LEN-HARDNESS OF FRAGMENTS

We have seen that the behavior of WhileFS can be expressed in the theory of TWE. A decision procedure for TWE-satisfiability would enable automatic analysis of WhileFS programs. However, the theory is hard to decide: it encodes the theory of word equations with length constraints (WE + LEN), the decidability of which remains a longstanding open question [13, 28]. In fact, there are several distinct ways it can encode WE + LEN; this section describes these reductions via different fragments of TWE. Figure 7 summarizes the relationships between TWE, WE + LEN, and WE.

We first demonstrate two fragment reductions that use different features of TWE (§5.1 and §5.2). We start with a reduction that only uses the word equations and normalization (norm) features of TWE—*i.e.*, we reduce to a fragment of the full theory, WE + NORM, that is already sufficient to encode WE + LEN. The reduction to TWE, a superset of WE + NORM, therefore follows trivially. We then observe that the same structure of reduction is possible using only word equations and walk assertions ($P \twoheadrightarrow_V L$), thereby establishing that a second fragment (WE + WALK) can also encode WE + LEN. Together, these results map out which features of TWE contribute to its hardness.

The constructions exploit *unbounded but controlled upward traversal* via '..' and a mechanism for *observation of the root directory* to force equality between the *depths* of walks induced by paths, which corresponds to length equality for certain kinds of walks. Drawing on these observations, we describe a fragment that likely sits at the boundary of decidability for TWE (§5.3): word equations

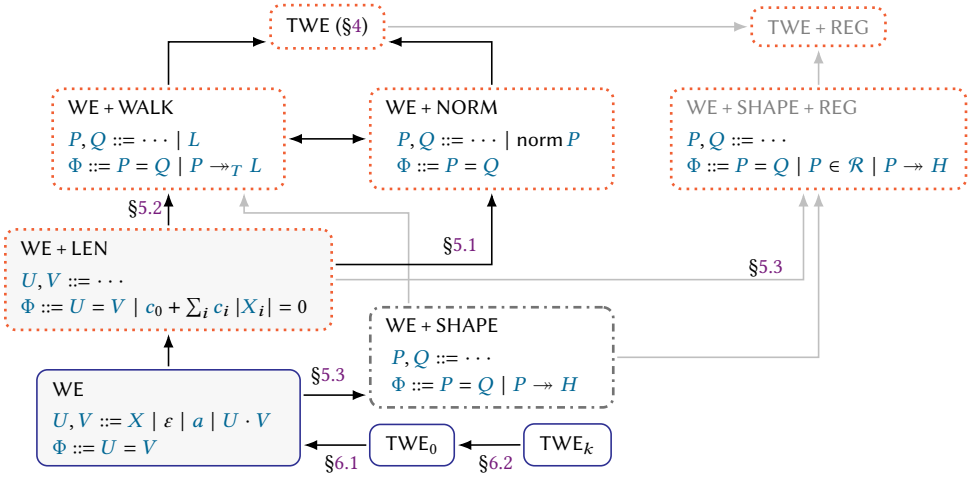


Figure 7. **Relationships between hard (by reduction from WE + LEN) and decidable fragments.** $\mathcal{T}_1 \longrightarrow \mathcal{T}_2$ means that there is a computable, equisatisfiable reduction of $\mathcal{T}_1$ into $\mathcal{T}_2$. §5 and §6 consider chiefly the fragments on the left and center of the figure; those on the right of the figure are not fragments of TWE, but contextualize the discussed fragments. The decidability of WE + SHAPE (dash-dotted frame) is uncertain.

plus shape constraints. This fragment, WE + SHAPE, omits exactly the features of TWE that WE + NORM and WE + WALK demonstrate contribute (independently) to the problem's hardness, making it the next clear candidate fragment. The prior reduction strategy does not work for this fragment. However, adding regular constraints (a standard extension axis for word equation theories) recovers the length equalities, so the resulting theory WE + SHAPE + REG is as hard as WE + LEN. These observations suggest that WE + SHAPE may be the most expressive fragment of TWE whose decidability remains in reach.

Preliminaries. Recall that TWE extends WE, the standard theory of ordinary word equations, *i.e.*, $\varphi ::= U = V$ only, over words U, V. WE + LEN extends WE with arithmetic predicates over *variable* lengths of the form $c_0 + \sum_i c_i |X_i| = 0$:

$$U, V \quad ::= \quad X \mid a \mid \varepsilon \mid U \cdot V \qquad\qquad \Phi ::= U = V \mid (c_0 + \textstyle\sum_i c_i |X_i| = 0)$$

We assume that values are over an infinite alphabet Σ *without* ϵ, ., or .., *i.e.*, purely ordinary names.

5.1 WE + LEN-Hardness of WE + NORM

First, consider the fragment of TWE which extends WE with only syntactic norm. Since there are no trees in this fragment, a model is simply a substitution σ as in ordinary WE. Eliding the standard connectives, WE + NORM has the grammar:

$$P, Q \quad ::= \quad X \mid a \mid \varepsilon \mid P \cdot Q \mid \mathsf{norm}\, P \qquad\qquad \Phi ::= P = Q$$

Theorem 3 (WE + LEN reduces to WE + NORM).

There is a computable, equisatisfiable reduction from WE + LEN *to* WE + NORM*, thus to* TWE.

In other words, there is a computable $(\!|\cdot|\!)$ such that for all WE + LEN formulae Φ,

$$\mathsf{SAT}_{\mathsf{WE+LEN}}(\Phi) \textit{ if and only if } \mathsf{SAT}_{\mathsf{WE+NORM}}((\!|\Phi|\!)).$$

It suffices to reduce *length equalities* $|U_1| = |U_2|$ and prove that WE + LEN models $\sigma_{\mathsf{LEN}} \models |U_1| = |U_2|$ can be transformed into WE + NORM ones $\sigma_{\mathsf{NORM}} \models (\!||U_1| = |U_2|\!)$ and vice versa (see Appendix H.1).

At a high level, the reduction enforces length equality by forcing U_1 and U_2 to represent *anchored walks from the root to the same depth*. Concretely, we reduce U_1 and U_2 to paths P_1 and P_2 which induce net downward walks that would return to the root via the same net upward Q, *i.e.*, $P_1 \cdot Q$ and $P_2 \cdot Q$ each induce *perfectly balanced excursions*. Since P_1 and P_2 are balanced by the same Q, they must induce walks to the same depth. The paths (in WE + NORM) must be net downward but need not be strictly linear; however, for so-called *anchored* paths, equal *depth* means that the *normalized* forms are of equal *length*. By taking the norm of the satisfying values for P_1 and P_2, we can recover the linear values for U_1, U_2 to reconstruct a satisfying model for WE + LEN.

First, we assume that Balanced and Anchored predicates are expressible in WE + NORM and demonstrate the reduction (§5.1.1). Then we define both and show the desired properties (§5.1.2).

5.1.1 Reduction. The idea is to map length-equal words to coordinated, *perfectly balanced excursions* which force their walks to have equal *depth*, and thereby recover equal *length*. In particular, map U_1 and U_2 to paths $P_1 \triangleq U_1$ and $P_2 \triangleq U_2$, each of which is balanced by the same "return" path Q:

$$(\!||U_1| = |U_2|\!) \quad \triangleq \quad \mathsf{Balanced}\,(P_1 \cdot Q) \wedge \mathsf{Balanced}\,(P_2 \cdot Q) \quad \text{fresh } Q$$

By *balanced* we mean that a path P induces a walk into the tree from the root which returns to the root (*e.g.*, figures 8c, 8e and 8f). By *perfectly*, we mean that the walk never attempts to go any further than the root, *i.e.*, never exhibits saturation (figure 8c only). Geometrically, it may be helpful to view such perfectly balanced excursions as loops starting and ending at the root. Balanced $(P_i \cdot Q)$ therefore means that P_i induces a walk into the tree from the root to some location L_i from which Q returns exactly to the root. Since P_1 and P_2 are perfectly balanced by the *same* Q, it must be that L_1 and L_2 are such that following the same sequence of components reaches the root. This does not mean that the locations are identical, but they must at least be at the same *depth* within the tree.

Precisely, perfect balance means an *anchored* path of net-zero depth:

Property 1 *(Perfect balance).* $\sigma \models \mathsf{Balanced}\,P$ *iff* $\sigma \models \mathsf{Anchored}\,P$ *and* $\mathsf{depth}\,[\![P]\!]_\sigma = 0$, *where* $\mathsf{depth}\,p \in \mathsf{Int}$ *counts the (possibly negative) net downward movement induced by* p*:*

$$\begin{array}{lll} \mathsf{depth}\,\varepsilon \triangleq 0 & \mathsf{depth}\,(p/\epsilon) \triangleq \mathsf{depth}\,p & \mathsf{depth}\,(p/a) \triangleq \mathsf{depth}\,p + 1 \\ & \mathsf{depth}\,(p/.) \triangleq \mathsf{depth}\,p & \mathsf{depth}\,(p/..) \triangleq \mathsf{depth}\,p - 1 \end{array}$$

We will return momentarily to the precise definition of Anchored, which disallows saturation to achieve perfect balance. For now, observe that depth distributes over concatenation, and if Balanced $(P_1 \cdot Q)$ and Balanced $(P_2 \cdot Q)$, then $\mathsf{depth}\,[\![P_1]\!]_\sigma = \mathsf{depth}\,[\![P_2]\!]_\sigma$. At this point, the forward direction of theorem 3 (constructing σ_{NORM} from σ_{LEN}) is trivial, since words in WE + LEN are linear paths in WE + NORM. For linear paths, $|p| = \mathsf{depth}\,p$, so P_1 and P_2 of equal length n can be perfectly balanced via $Q \mapsto ..^n$. These linear paths do not saturate, and their downward movement is canceled exactly by the sequence of n ..'s. Hence we may take σ_{NORM} to be σ_{LEN}, additionally mapping Q to $..^n$ where $n = \left|[\![P_1]\!]_{\sigma_{\mathsf{LEN}}}\right| = \left|[\![P_2]\!]_{\sigma_{\mathsf{LEN}}}\right|$.

The backwards direction is harder. Paths in WE + NORM may be non-linear, and we must recover *linear* ones to construct σ_{LEN}. Moreover, the equality of depth does not generally imply equality of length. For instance, $a/../a'$ and a have the same depth of 1 but different lengths. However, the *normalized* forms do have equal length, and this length is the same as the original depth: $\mathsf{norm}\,(a/../a') = a'$ and $\mathsf{norm}\,a = a$ each have length 1. This is a useful fact: if $\mathsf{depth}\,p = |\mathsf{norm}\,p|$, then where σ_{NORM} maps P_1 and P_2 to non-linear paths of equal depth, we may take their linear, equal-length normalizations for σ_{LEN}.

Nonetheless, there sometimes remains a mismatch between depth and normalized length: consider $../a$, which has depth 0 but whose normalized form a has length 1. This mismatch arises because depth always counts .. negatively, but norm swallows up .. in the same way that .. saturates at the root (see figure 8d). Figures 8e and 8f show paths which are "outwardly" balanced (starting and ending at the root) but for which the correspondence to depth is broken due to saturation. But the correspondence holds for paths which do not exhibit this saturation—equivalently, whose depth count never goes negative.

Property 2 *(Anchoring).* $\sigma \models \text{Anchored}\, P$ *iff* $\text{depth}\, p' \geq 0$ *for every prefix* $p' \preceq [\![P]\!]_\sigma$.

For such *anchored* paths, .. *always* cancels, hence normalization agrees with depth-counting in the desired way: if $\sigma \models \text{Anchored}\, P$, then $\text{depth}\, [\![P]\!]_\sigma = |\text{norm}\, [\![P]\!]_\sigma|$. As a result, the reduction gives $|\text{norm}\, [\![P_1]\!]_\sigma| = |\text{norm}\, [\![P_2]\!]_\sigma|$, and to reconstruct equal-length, linear values for U_1 and U_2, we may simply take $\text{norm}\, [\![P_1]\!]_\sigma$ and $\text{norm}\, [\![P_2]\!]_\sigma$. This completes the backwards direction (see Appendix H.1 for the full construction). It remains only to define Balanced and Anchored in WE + NORM.

5.1.2 Defining Balanced *and* Anchored. The two predicates can be expressed using norm:

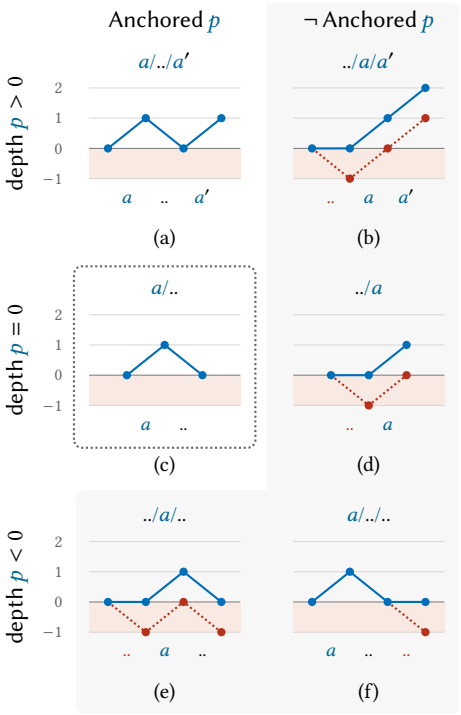


Figure 8. **Anchored and unanchored paths of different depths.** Each plot follows depth across p as resolved (solid blue) and according to depth p (dotted red), drawn only when it departs from resolution (possible only when unanchored). $a/..$ (boxed) alone is perfectly balanced.

$$\begin{aligned} \text{Anchored}\, P &\triangleq \text{norm}\, P \neq \text{norm}\, (a \cdot P) \qquad \text{fresh } a \\ \text{Balanced}\, P &\triangleq \text{Anchored}\, P \wedge \text{norm}\, P = \varepsilon \end{aligned}$$

The second conjunct for Balanced means that P is an excursion which returns to the root, detected via equality with the empty path. Anchored P ensures that prefixes never have depth less than 0 by preventing saturation in P. If P never attempts to rise above the root, then a is untouched by normalization, so $\text{norm}\, (a \cdot P) = a \cdot \text{norm}\, P$ differs from $\text{norm}\, P$. Otherwise, if it does, then $P = P_1 \cdot P_2$ where some P_1 has depth of -1. In this case, the residual .. cancels a, and from that point both sides normalize identically, violating the disequality.

Lemma 1. Balanced *and* Anchored *satisfy properties 1 and 2, respectively.*

5.2 WE + LEN-Hardness of WE + WALK

WE + WALK, which extends bare WE with walk assertions over abstract trees $P \twoheadrightarrow_V L$, can encode WE + LEN similarly, using locations in lieu of norm for Balanced and Anchored (see Appendix H.3):

Theorem 4 (WE + LEN reduces to WE + WALK).

There is a computable, equisatisfiable reduction from WE + LEN *to* WE + WALK, *thus to* TWE.

5.3 WE + LEN-Hardness of WE + SHAPE + REG

The constructions for WE + NORM and WE + WALK observe the depth of a path and transport depth equalities to length equalities via a restriction to anchored paths. This argument relies on reasoning about steps within the walk using normalized forms or locations. As such, the construction fails when the ability to examine normalized forms is removed, as in the fragment WE + SHAPE, which allows walks to *opaque* locations, $P \twoheadrightarrow_V H \triangleq P \twoheadrightarrow_V L \in H$, where L is always fresh and inaccessible:

$$P, Q \quad ::= \quad X \mid a \mid \varepsilon \mid P \cdot Q \qquad\qquad \Phi ::= P = Q \mid P \twoheadrightarrow_V H$$

Without access to norm or L, it is no longer clear how to detect that a walk goes to the root directory or to restrict solutions to anchored paths. Even if precise root isolation could be achieved using shape assertions, it is unclear how non-anchored paths could be eliminated. Interestingly, however, length equalities are recovered if regular constraints $\mathcal{R}$ on paths are added to this fragment, yielding WE + SHAPE + REG:

$$\Phi \quad ::= \quad \cdots \mid P \in \mathcal{R}$$

Theorem 5 (WE + LEN reduces to WE + SHAPE + REG).

There is likewise a reduction from WE + LEN *to* WE + SHAPE + REG, *thus to* TWE + REG.

The regular constraints in WE + SHAPE + REG can be represented by symbolic automata over the infinite alphabet of path components. The regular constraints put paths into a rigid form in which a block of .. components can serve as a counter for the length of a word. Shape assertions then rule out the wrong counter lengths: if the block is too short or too long, the constraints force incompatible file and directory requirements. We leave the precise proof to Appendix H.4.

6 DECIDABILITY OF LINEAR AND BOUNDED THEORY FRAGMENTS

While the previous section shows that the full theory of TWE is hard to decide, the reliance of its reductions on *unbounded upward movement* points to useful decidable fragments. In §6.1 we show decidability of the *linear* fragment TWE_0, which disallows non-linear components in paths and thus only permits downward movement. §6.2 uses this engine to decide a *bounded* non-linear fragment TWE_k in which the number of occurrences of non-linear components in each path is bounded by an arbitrary constant k.

Unlike the fragments in the previous section, these are *semantic restrictions* on models, not on the syntax of TWE-formulae or $\textsc{While}_{\mathsf{FS}}$ programs. They are nonetheless well-motivated and practically realizable: in practice paths are often linear or exhibit a limited number of .. components. File system interaction bugs may be discovered with reasonably low bounds; for instance, the buggy execution of `rmr` in listing 2 requires only $k = 1$. Moreover, the reduction to word equations allows leveraging off-the-shelf solvers which support sequence theories, such as Z3 [21] or cvc5 [8].

6.1 Decidability of linear TWE

First, we consider TWE_0, a fragment of TWE which allows only models which do not exhibit non-linear components in any path. Precisely, a model $\langle \sigma, \theta \rangle$ is within the fragment if, for all P which appear in the formula Φ, the evaluation of P is linear, *i.e.*, $[\![P]\!]_{\sigma,\theta}$ lin. Note that all constant components in Φ must be linear as a result. For instance, for the formula $V[X] \in \mathsf{dirc}$, TWE_0 admits models mapping X to ε or a' but not any mapping it to paths containing ϵ, ., or .. components. Since every path is linear, every walk $P \twoheadrightarrow_T L$ follows ordinary name components downward into the tree; indeed, P is necessarily identical to L. In this setting, tree reasoning can be compiled into pure word constraints.

The target WE theory is over an infinite alphabet and distinguishes between variables standing for words and single letters. If letter equality is decidable in PSPACE, then WE is as well [42, Theorem 3].

> Theorem 6.
>
> *There is an equisatisfiable* PSPACE *reduction from* TWE_0 *to* WE. *Therefore,* TWE_0*-satisfiability is decidable in* PSPACE.

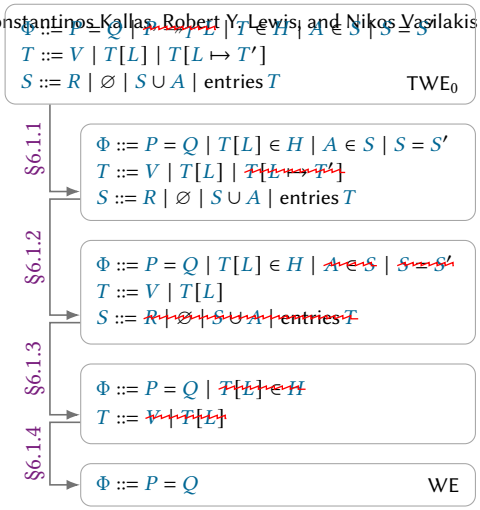


Figure 9. **Reduction of TWE_0 to WE.**

The construction, illustrated in figure 9, progressively rewrites features from the theory until pure word constraints are reached. The key fact is that linear paths collapse to locations; we can rewrite walks $P \twoheadrightarrow_T L$ into shape assertions over tree selections $T[L] \in H$ (§6.1.1).

The remaining features—tree updates, entries sets, and shape consistency—reason *locally* with respect to *ancestorship*: each atom depends only on the ancestor and descendant relationships between locations, which are word-expressible using prefix predicates. Concretely, we can eliminate tree updates $T[L \mapsto T']$, so that all trees are selections over abstract trees $V[L]$ only (§6.1.2). Then, we rewrite entries T sets into words whose elements must be mutually distinct (§6.1.3). This property is not typically expressible over words of unbounded length, but since linear walks traverse only one location at each depth-level in the tree, it is possible to establish a small model property for these sets and express mutual distinctness via a bounded number of pairwise conjunctions. Finally, trees are eliminated entirely: shape assertions over selections are rewritten via a topological analysis into prefix constraints (§6.1.4). See Appendix I for a full proof.

6.1.1 Rewriting walk assertions. Since paths coincide with locations in the linear setting, walks coincide with tree selections. For $P \twoheadrightarrow_T L$, since $P = L$, the walk is equivalent to a selection $T[L]$:

$$P \twoheadrightarrow_T L \;\Leftrightarrow\; (P = L) \wedge \overbrace{(T[\varepsilon] \in \{\mathsf{dirc}, \mathsf{file}\})}^{\text{root exists}} \wedge \overbrace{(T[L] \in \{\mathsf{dirc}, \mathsf{file}, \bot\})}^{\text{selection succeeds}} \quad \text{where} \quad T \in \mathcal{H} \triangleq \bigvee_{H \in \mathcal{H}} T \in H$$

6.1.2 Eliminating tree updates. We eliminate $T[L_{\mathsf{up}} \mapsto T']$ inductively, such that only selections over abstract trees remain. In essence, we can capture the effect of updates on selections based on whether the selection falls inside or outside the updated subtree. For a selection at an arbitrary location L, we split into two cases: (1) if the selection falls within the updated subtree, it proceeds into the subtree; (2) otherwise, it observes the original tree. Writing $L \preceq L'$ to denote that L is a prefix of L', for shape constraints:

$$T[L_{\mathsf{up}} \mapsto T'][L] \in H \iff \overbrace{T[L_{\mathsf{up}}] \in \{\mathsf{dirc}, \mathsf{file}, \bot\}}^{\text{update is defined}} \wedge \left(\begin{array}{l} \overbrace{\big((L = L_{\mathsf{up}} \cdot L') \wedge (T'[L'] \in H)\big)}^{L \text{ is inside } T'} \\ \vee \underbrace{\big((L_{\mathsf{up}} \not\preceq L) \wedge (T[L] \in H)\big)}_{L \text{ is outside } T'} \end{array} \right)$$

Likewise, we must rewrite entries over updates. Again, since the effect of the update is local, and the set of entries concerns only the *immediate children* of a tree, an update at L_{up} may affect it in three

cases depending on the ancestor relationship between the queried and updated locations. (1) If the directory queried at L is inside the updated subtree T' (*i.e.*, $L = L_{\mathsf{up}} \cdot L'$), then the updated directory $T'[L']$ is used. Otherwise, (2) if it is outside the updated subtree, the entry set is unaffected—*except* (3) if the update is at a direct child (*i.e.*, $L \cdot A = L_{\mathsf{up}}$), in which case the update adds or removes that child's name A. Precisely, we replace entries $T[L_{\mathsf{up}} \mapsto T'][L]$ with an abstract set R with a condition given by $T[L_{\mathsf{up}}] \in \{\mathsf{dirc}, \mathsf{file}, \bot\} \wedge (\Psi_{\mathsf{inside}} \vee \Psi_{\mathsf{outside}} \vee \Psi_{\mathsf{child}})$:

$$\Psi_{\mathsf{inside}} \triangleq \overbrace{L_{\mathsf{up}} \cdot L' = L}^{L \text{ is inside } T'} \wedge \overbrace{R = \mathsf{entries}\, T'[L']}^{R \text{ is entries from } T'} \qquad \Psi_{\mathsf{outside}} \triangleq \overbrace{\begin{pmatrix} L_{\mathsf{up}} = L \cdot A \cdot A' \cdot L' \\ \vee\ L \parallel L_{\mathsf{up}} \end{pmatrix}}^{L \text{ is outside parent of } T'} \wedge \overbrace{R = \mathsf{entries}\, T[L]}^{R \text{ is entries from } T}$$

$$\Psi_{\mathsf{child}} \triangleq \overbrace{L_{\mathsf{up}} = L \cdot A}^{L \text{ is parent of } T'} \wedge \underbrace{\begin{pmatrix} \overbrace{(T' \in \{\mathsf{file}, \mathsf{dirc}\} \wedge R = (\mathsf{entries}\, T[L]) \cup A)}^{T' \text{ exists} \Rightarrow R \text{ is entries from } T \textit{ including } A} \\ \vee\ (T' \in \bot \wedge A \notin R \wedge R \cup A = (\mathsf{entries}\, T[L]) \cup A) \end{pmatrix}}_{T' \text{ is absent} \Rightarrow R \text{ is entries from } T \textit{ excluding } A}$$

We write $L \parallel L'$ to mean that neither is a prefix of the other. At this point, all tree terms are nested selections over variables $V[L_1] \cdots [L_k]$. We may safely flatten each of these chains into a single selection $V[L_1 \cdot \cdots \cdot L_k]$.

6.1.3 Reducing entry sets into bounded-length words. After elimination of tree updates, every entries term has the form $\mathsf{entries}\, V[L]$. We eliminate such terms first, observing that the set of names admits finite description based on the selections witnessed in the formula. Then, we encode name sets S into words whose elements must be mutually distinct. This property is not generally word-expressible for words of *unbounded* length, but it is possible to establish a small model property on name sets and enforce mutual distinctness using polynomially many disequalities.

Eliminating entries. Observe that $[\![\mathsf{entries}\, V[L]]\!]_{\sigma,\theta}$ is effectively the set of immediate child names $\{a \in \mathsf{Name} \mid \langle \sigma, \theta \rangle \models V[L \cdot a] \in \{\mathsf{file}, \mathsf{dirc}\}\}$. This set is finite yet unbounded, but it admits a small model property, since only those *names witnessed by some selection on* V affect the satisfaction of the formula. The membership of witnessed names is fixed, while the set is free to include or exclude all other names. Consider for instance the formula $(R = \mathsf{entries}\, V[L]) \wedge (V[L_1] \in H_1) \wedge (V[L_2] \in H_2)$. The two selections require that all the ancestors of L_1 and L_2 exist, hence if L is an ancestor of any of these locations, then the set for R must contain the corresponding names. This is all that is required to satisfy the formula; whether or not other names are also in the set is irrelevant.

Hence it suffices to replace entries terms with abstract name sets R equipped with guarded membership predicates ranging over the finite set of locations appearing in selections, *i.e.*, $L \in \mathcal{L}_V(\Phi) = \{L \mid V[L] \text{ in } \Phi\}$. In particular, we assert $\Psi_{\{\mathsf{file},\bot\}} \vee \Psi_{\mathsf{dirc}}$:

$$\Psi_{\{\mathsf{file},\bot\}} \triangleq \overbrace{V[L] \in \{\mathsf{file}, \bot\}}^{L \text{ is not a directory}} \wedge \overbrace{R = \varnothing}^{R \text{ is empty}} \qquad \Psi_{\mathsf{dirc}} \triangleq \overbrace{V[L] \in \mathsf{dirc}}^{L \text{ is a directory}} \wedge \overbrace{\left(\bigwedge_{L_{\mathsf{sel}} \in \mathcal{L}_V(\Phi)} \Psi_{\mathsf{unrelated}} \vee \Psi_{\mathsf{descendant}} \right)}^{R \text{ is pinned by selections}}$$

In essence, files and absent locations have no entries, but for directories, we again perform prefix-based case analysis to determine if a selection at L_{sel} contributes to pinned names at $V[L]$:

$$\Psi_{\mathsf{unrelated}} \triangleq \overbrace{L_{\mathsf{sel}} = L \vee \left(L \not\preceq L_{\mathsf{sel}}\right)}^{L_{\mathsf{sel}} \text{ is unrelated}}$$

$$\Psi_{\mathsf{descendant}} \triangleq \overbrace{L_{\mathsf{sel}} = L \cdot A \cdot L''}^{L_{\mathsf{sel}} \text{ is a strict descendant}} \wedge \underbrace{\begin{pmatrix} \overbrace{(V[L \cdot A] \in \{\mathsf{file}, \mathsf{dirc}\} \wedge A \in R)}^{L \cdot A \text{ exists} \Rightarrow A \textit{ included}} \\ \vee\ (V[L \cdot A] \in \bot \wedge A \notin R) \end{pmatrix}}_{L \cdot A \text{ is absent} \Rightarrow A \textit{ excluded}}$$

If the selection is unrelated, *i.e.*, not a strict descendant of $V[L]$, then it is outside the region of interest. Otherwise, it is a strict descendant and contributes to R through the child A of L on its way: $A \in R$ if $L \cdot A$ exists, and $A \notin R$ if it is absent. Additionally, to preserve functional consistency, for any pair of entries entries $V[L_1]$ and entries $V[L_2]$, we assert that $L_1 = L_2 \Rightarrow R_1 = R_2$.

Reducing name sets into words. The remaining set atoms are rewritten into word equations. First, each set disequality $S_1 \neq S_2$ is rewritten using a fresh witness name A, becoming

$$\overbrace{(A \in S_1 \wedge A \notin S_2)}^{A \text{ in } S_1, \text{ not in } S_2} \vee \overbrace{(A \notin S_1 \wedge A \in S_2)}^{A \text{ not in } S_1, \text{ in } S_2}.$$

Now, for name sets, only assertions for equality $S_1 = S_2$, membership $A \in S$, and non-membership $A \notin S$ remain, over set terms $S ::= R \mid \varnothing \mid S \cup A$. A standard small model property enables a reduction of sets to words. Because only membership predicates and insertions pin set members, the sets for satisfying models can always be pruned to those with cardinality of at most $|\mathcal{S}(\Phi)|$, where $\mathcal{S}(\Phi)$ is the set $\{A \mid A \in S \text{ or } S \cup A \text{ in } \Phi\}$. Then, each set variable R is replaced by a sequence $\overline{S}$ of fresh name variables $X_{A1} \cdot \cdots \cdot X_{Ak}$, for $0 \leq k \leq |\mathcal{S}(\Phi)|$, that are pairwise distinct:

$$\bigvee_{0 \leq k \leq |\mathcal{S}(\Phi)|} \left(\overline{S} = X_{A1} \cdot \cdots \cdot X_{Ak} \wedge \bigwedge_{1 \leq i < j \leq k} X_{Ai} \neq X_{Aj} \right).$$

6.1.4 Reifying shape lookups into prefix analysis. Once left with word equations and shape assertions over tree selections $V[L] \in H$, the question is one of realizability: is there an arrangement of the locations such that the shapes describe a valid tree? The sole structural invariant is that all files and absent locations must be in leaf positions, and this property is readily expressed in terms of word-expressible ancestorship predicates: if $V[L] \in \{\text{file}, \bot\}$, then for any other selection $V[L'] \in H$, L' must not be a strict descendant of L, *i.e.*, $L \not\prec L'$. As with entries, functional consistency asserts that $L_1 = L_2 \Rightarrow H_1 = H_2$. The resulting formula is in WE, which is decidable in PSPACE.

6.2 Decidability of bounded non-linear TWE

A decision procedure for the linear fragment TWE_0 enables decidability of *bounded* non-linear fragments TWE_k for any $k \geq 0$. In these fragments, each path may exhibit up to k-many non-linear components. The construction is straightforward: since at most k components are non-linear, we may decompose each path into $k + 1$ *linear* paths delimited by up to k non-linear components. Then, each non-linear step of a walk can be handled explicitly via case work and simulated via linear walks and shape assertions in TWE_0. The full construction is found in Appendix I.6.

> Theorem 7.
>
> *There is an equisatisfiable* PSPACE *reduction from* TWE_k *to* TWE_0 *and to* WE. *Therefore,* TWE_k*-satisfiability is decidable in* PSPACE*, uniformly in* k.

For each path P, we know that $P = X_1 \cdot B_1 \cdot X_2 \cdot B_2 \cdot \cdots \cdot B_k \cdot X_{k+1}$ where each X_i is linear and each B_i must be empty ε or one of the non-linear $\epsilon, ., ..$ components. To reduce a walk $P \twoheadrightarrow_T L$ to TWE_0, we introduce fresh linear variables Y_i for $1 \leq i \leq k + 1$ and Z_i for $1 \leq i \leq k$, with the intention that Y_i will correspond to the location reached after resolving the prefix through X_i and

Z_i after resolving B_i as well. Precisely, $Y_1 = X_1$ and, for $1 \le i \le k$ and fresh name variables X_{Ai},

$$\overbrace{Y_{i+1} = Z_i \cdot X_{i+1}}^{Y_{i+1} \text{ follows } X_{i+1} \text{ from } Z_i} \wedge \left(\begin{array}{l} \overbrace{(B_i \in \{\varepsilon, \epsilon, .\} \wedge Z_i = Y_i)}^{B_i \text{ stays put}} \\ \vee \underbrace{(B_i = .. \wedge ((Y_i = \varepsilon \wedge Z_i = \varepsilon) \vee Y_i = Z_i \cdot X_{Ai}))}_{B_i \text{ moves to the parent, or stays at the root}} \end{array} \right)$$

Since P resolves successfully to L when Y_i is a directory for every $1 \le i \le k$ and $Y_{k+1} = L$, we may write $P \twoheadrightarrow_T L$ using linear walks and shape assertions:

$$\overbrace{Y_{k+1} \twoheadrightarrow_T L}^{P \text{ resolves to } L} \wedge \overbrace{\left(\bigwedge_{1 \le i \le k} Y_i \twoheadrightarrow_T L_i \in \mathsf{dirc} \right)}^{\text{each } B_i \text{ from a directory}}.$$

Example 10. Recall $Y \twoheadrightarrow_{V'} L_A$ from example 8, where $V' = V[L_A \mapsto \bot]$. With $k = 1$, the reduction decomposes $Y = X_1 \cdot B_1 \cdot X_2$ and generates the linear walks $Z_1 \cdot X_2 \twoheadrightarrow_{V'} L_A$ and $X_1 \twoheadrightarrow_{V'} L_1 \in \mathsf{dirc}$, together with the case split on B_1. When $B_1 \in \{\varepsilon, \epsilon, .\}$, $Z_1 = X_1$. The reduction to WE (§6.1) obtains $X_1 = L_1 \wedge L_A = L_1 \cdot X_2 \wedge L_A \not\preceq L_1$. Solving these constraints gives a satisfying witness $B_1 \mapsto \varepsilon$, $X_1 \mapsto \texttt{app}$, and $X_2 \mapsto \texttt{config}$, reproducing the witness $\sigma\, Y = \texttt{app/config}$, the same as in example 8.

7 RELATED WORK

Logical foundations. TWE shares ingredients with several areas of prior art—including word equations, tree navigation, and updates—drawing them into a single theory for file system reasoning.

Word equations. TWE includes word equations (WE), which enjoy a rich body of literature. Makanin [51] (see also Diekert [24]) first solved the satisfiability problem, and the proposed algorithm was gradually improved to EXPSPACE [34, 43, 61] and eventually to PSPACE [41, 57]. A number of notable extensions to WE are also well-studied, including the addition of regular constraints [61] (REG, decidable in PSPACE) and length constraints [28, 49] (decidability open), as well as restrictions such as quadratic equations [25], straight-line [48] and chain-free fragments [2, 48], and pattern languages [4]. Word equations over infinite alphabets with constraints have also been studied, with decidability and complexity bounds established for several fragments [42]. TWE fits into this line of work as a family of extensions to WE in which words may also induce walks navigating over finite trees. This is in contrast to context unification (decidable in PSPACE [40]), which directly generalizes words into one-holed tree contexts. The cancellation behavior of .. in walks bears some resemblance to free inverse monoids [23], though the inverse is one-sided and saturates at the root.

Feature logics. The tree elements of TWE share commonalities with several flavors of feature logics. FT [7] is a theory of finite trees with named *features*, which correspond to *names* in our setting. CFT [63] extends FT with arity constraints expressing that a tree has exactly some *constant* set of features; TWE instead supports reasoning about the entry names of a directory as an unbounded, symbolic set. Like EF [67], an extension of CFT with first-class features, TWE treats *names* as first-class values; however, they are also subject to word equations. The CoLiS project [9] explores file system reasoning using a decidable first-order theory of feature trees extended with functional updates [39] similar to those in TWE. But whereas paths in CoLiS are strictly concrete and linear, our framework supports reasoning about paths with variables, non-linear elements, and equational constraints. Backofen [5, 6] introduces path variables and path equality to feature logics, but the decidable fragments considered there do not support equality over arbitrary words. Moreover, the iterated feature selection of path resolution exhibits saturation of parent references at the root.

Tree-walking automata. Tree-walking automata (TWA) [3], finite state machines which recognize tree languages by walking over an input tree in a manner similar to string automata, have been studied extensively [11]. The procedural perspective on path resolution is directly aligned with tree-walking automata: every concrete path can be understood as a deterministic TWA which accepts file trees in which the path successfully resolves. TWE considers not only concrete paths but also symbolic ones—making some part of the automaton unknown (similar to nested TWAs [66])—and moreover adds constraints over those unknown portions via word equations.

XPath logics. Several lines of work use tree logics and and TWA [10, 66] to model XPath [70], a query language for extracting portions of XML trees. XPath queries are more expressive than file system paths, supporting data-value predicates within the path itself (*e.g.*, field X of a node satisfies some condition). Existing logics focus on selection over a fixed tree and do not model the XQuery Update Facility [71], which provides a way to mutate the XML tree. TWE supports updates.

Separation logics. SL [58] offers a framework for compositional reasoning over resource heaps based on splitting heaps into disjoint fragments. Hierarchical file systems can be understood as memory heaps and paths as addresses (§2.1). Gardner et al. [29], drawing on context logic [16], investigate program logics based on structural SL for file system reasoning over *linear* paths. However, non-linear components pose a challenge for disjoint reasoning by introducing non-local walk behavior. To handle non-linear paths, effect fusion logic [56] introduces an effect separating conjunction and effect frame rule in order to propagate updates across heap fragments. Our framework does not aim to develop a program logic; rather, TWE aims to be a basis for first-order automated and semi-automated analyses by isolating the constraint problem generated by path and tree reasoning.

Reasoning about file systems. A complementary body of work focuses on the file system domain, including the correctness of file system implementations and analyses of client programs.

Verified file systems. Verified file systems [17, 18, 62] are file system *implementations* proven correct with respect to a specification, aiming to offer correctness guarantees in the implementation of operations such as file creation or writing. Relatedly, SibylFS [59] provides an executable specification of the POSIX file system interface used to test implementations for conformance. These concerns are complementary to our framework, which targets reasoning about the correctness of *client programs using a file system*, assuming that the implementation is correct.

Static analyses. Several static analysis systems reason about file system interactions to some degree. ShellCheck [36] is a linter that flags *syntactic* patterns corresponding to code smells, some of which involve likely-buggy file system interactions (*e.g.*, deletions of important system paths). In contrast, our framework targets *semantic* reasoning about program execution. KLEE [15] is a symbolic execution engine for LLVM and uses a flat directory structure consisting of a fixed, constant number of symbolic files. CoLiS [9, 39] analyzes Debian package installation scripts but is restricted to concrete and linear paths. SaSh [45, 46] performs symbolic execution to find bugs in shell programs using a coarse, non-hierarchical model of the file system. FiFA [60] uses abstract interpretation over path values to extract file system state pre-conditions required for a program's correct execution, but paths are treated as flat strings rather than as modeling resolution over a hierarchical tree. Smoosh [32] is an executable formal semantics for POSIX shell using a simplified file system model that does not support abstract/symbolic paths or states. PaSh's annotations [35, 69] record per-command I/O effects on files with concrete paths. In contrast to these works, TWE accommodates fully abstract/symbolic paths and richer structural modeling over arbitrarily shaped and sized hierarchical file system states. The TWE framework can enrich these analyses to support hierarchical file systems and symbolic, non-linear paths.

Effect systems. Type-and-effect systems [31, 50, 65], including algebraic effect handlers [47], encode over-approximations of effectful computations into a static typing analysis. The granularity of tracked effects is the kind of effect a program term may produce—*e.g.*, a file write or a file read—not the affected paths. In contrast, effect capability systems [12] with capabilities over file/directory paths [20, 30] treat paths as opaque tokens and do not model path resolution. TWE could serve as a predicate language for such systems to enable reasoning about path relationships and resolution.

DSLs. Forest [27] generates type-safe filestores and traversal APIs over the file system from data descriptions. On the other hand, TWE reasons directly about a program's interactions with paths and the file system itself. TxForest [26] uses zippers to achieve incrementality, locality, and functional updates. We use zippers to model non-linear path resolution along parent references.

8 FUTURE DIRECTIONS

WHILE$_{FS}$ and TWE provide foundations for further investigation in multiple directions. One is to study the decidability of richer fragments of TWE, including the open fragment WE + SHAPE (§5.3), and TWE with standard restrictions on word equation structure such as straight-line [48] and chain-free fragments [2, 48]. Since the decidable fragments in this paper reduce to WE, automated and semi-automated verifiers and bug-finders based on WHILE$_{FS}$'s symbolic semantics could use off-the-shelf sequence theory solvers [8, 21] to reason about real code using the file system.

TWE itself presents several opportunities for extension to support reasoning about node-level enrichments such as hard links and permissions, structural enrichments like soft links and mounts, concurrent and networked file systems, and platform-specific semantics. Handling soft links, for instance, would require changing file trees and path resolution, augmenting WHILE$_{FS}$'s concrete semantics and TWE's models. TWE-formulae and the symbolic semantics would then transfer without modification into this new setting. Such enrichments would likely inherit TWE's hardness while complicating decidability. Since TWE describes file system *states*, it could underlie a concurrent reasoning system over executions. WHILE$_{FS}$ and TWE moreover provide foundations to precisely describe and compare file system features across platforms and implementations, *e.g.*, POSIX's case-sensitive, non-linear paths vs. Win32's case-insensitive and sometimes verbatim paths, in which . and .. are ordinary names and for which norm is identity.

ACKNOWLEDGMENTS

We would like to thank Evangelos Lamprou, Ethan Lavi, Yuchen Lu, Oğuzhan Çölkesen, Ken Mahattanadul, George Kapetanakis, Will Crichton, Gavin Gray, Konstantinos Mamouras, and the anonymous POPL 2027 referees for helpful feedback on earlier versions of this paper. This material is based upon research supported by NSF awards CNS-2541584, CCF-2525351, CNS-2247687, and CNS-2312346, DARPA contract no. HR001124C0486, an Amazon Research Award (Fall 2024), a Google ML-and-Systems Junior Faculty Award, a seed grant from Brown University's Data Science Institute, and a BrownCS Faculty Innovation Award.

DISCLOSURE OF USE OF AI IN RESEARCH

The mechanized proofs contained in the artifact were produced with the assistance of AI tools. The definitions and theorem statements of the formalization were reviewed by the authors, and all proofs are checked by the Lean kernel. The authors take full responsibility for the content of the paper and artifact.

DATA AVAILABILITY STATEMENT

An artifact [72] contains mechanized proofs of all theorems in the paper. The artifact is also available on GitHub at https://github.com/atlas-brown/twe-artifact.

REFERENCES

[1] Michael Gordon Abbott, Thorsten Altenkirch, Conor McBride, and Neil Ghani. 2005. δ for Data: Differentiating Data Structures. *Fundam. Informaticae* 65, 1-2 (2005), 1–28. http://content.iospress.com/articles/fundamenta-informaticae/fi65-1-2-02

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bui Phi Diep, Luk ás Holík, and Petr Janku. 2019. Chain-Free String Constraints. In *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11781)*, Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza (Eds.). Springer, 277–293. https://doi.org/10.1007/978-3-030-31784-3_16

[3] Alfred V. Aho and Jeffrey D. Ullman. 1971. Translations on a Context-Free Grammar. *Inf. Control.* 19, 5 (1971), 439–475. https://doi.org/10.1016/S0019-9958(71)90706-6

[4] Dana Angluin. 1980. Finding Patterns Common to a Set of Strings. *J. Comput. Syst. Sci.* 21, 1 (1980), 46–62. https://doi.org/10.1016/0022-0000(80)90041-0

[5] Rolf Backofen. 1994. Regular Path Expressions in Feature Logic. *J. Symb. Comput.* 17, 5 (1994), 421–455. https://doi.org/10.1006/JSCO.1994.1028

[6] Rolf Backofen. 1996. Controlling Functional Uncertainty. In *12th European Conference on Artificial Intelligence, Budapest, Hungary, August 11-16, 1996, Proceedings*, Wolfgang Wahlster (Ed.). John Wiley and Sons, Chichester, 557–561.

[7] Rolf Backofen and Gert Smolka. 1993. A Complete and Recursive Feature Theory. In *31st Annual Meeting of the Association for Computational Linguistics, 22-26 June 1993, Ohio State University, Columbus, Ohio, USA, Proceedings*, Lenhart K. Schubert (Ed.). ACL, 193–200. https://doi.org/10.3115/981574.981600

[8] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 415–442.

[9] Benedikt F. H. Becker, Nicolas Jeannerod, Claude Marché, Yann Régis-Gianas, Mihaela Sighireanu, and Ralf Treinen. 2022. The CoLiS platform for the analysis of maintainer scripts in Debian software packages. *Int. J. Softw. Tools Technol. Transf.* 24, 5 (2022), 717–733. https://doi.org/10.1007/S10009-022-00671-1

[10] Michael Benedikt and Christoph Koch. 2008. XPath leashed. *ACM Comput. Surv.* 41, 1 (2008), 3:1–3:54. https://doi.org/10.1145/1456650.1456653

[11] Mikolaj Bojanczyk. 2008. Tree-Walking Automata. In *Language and Automata Theory and Applications, Second International Conference, LATA 2008, Tarragona, Spain, March 13-19, 2008. Revised Papers (Lecture Notes in Computer Science, Vol. 5196)*, Carlos Martín-Vide, Friedrich Otto, and Henning Fernau (Eds.). Springer, 1–2. https://doi.org/10.1007/978-3-540-88282-4_1

[12] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 126:1–126:30. https://doi.org/10.1145/3428194

[13] J. Richard Büchi and Steven Senger. 1988. Definability in the Existential Theory of Concatenation and Undecidable Extensions of this Theory. *Math. Log. Q.* 34, 4 (1988), 337–342. https://doi.org/10.1002/MALQ.19880340410

[14] BusyBox. [n. d.]. BusyBox: The Swiss Army Knife of Embedded Linux. https://www.busybox.net/.

[15] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[16] Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. 2005. Context logic and tree update. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, Jens Palsberg and Martín Abadi (Eds.). ACM, 271–282. https://doi.org/10.1145/1040305.1040328

[17] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay Mert Ileri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2017. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles , Shanghai, China, October 28-31, 2017*. ACM, 270–286. https://doi.org/10.1145/3132747.3132776

[18] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems*

Principles , SOSP 2015, Monterey, CA, USA, October 4-7, 2015, Ethan L. Miller and Steven Hand (Eds.). ACM, 18–37. https://doi.org/10.1145/2815400.2815402

[19] Catalin Cimpanu. 2018. Botched Npm Update Crashes Linux Systems, Forces Users to Reinstall. https://www.bleepingcomputer.com/news/linux/botched-npm-update-crashes-linux-systems-forces-users-to-reinstall/.

[20] Aaron Craig, Alex Potanin, Lindsay Groves, and Jonathan Aldrich. 2018. Capabilities: Effects for Free. In *Formal Methods and Software Engineering - 20th International Conference on Formal Engineering Methods, ICFEM 2018, Gold Coast , QLD, Australia, November 12-16, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11232)*, Jing Sun and Meng Sun (Eds.). Springer, 231–247. https://doi.org/10.1007/978-3-030-02450-5_14

[21] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[22] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 625–635. https://doi.org/10.1007/978-3-030-79876-5_37

[23] Timothy Deis, John Meakin, and G. Sénizergues. 2007. Equations in Free Inverse Monoids. *International Journal of Algebra and Computation* 17, 04 (June 2007), 761–795. https://doi.org/10.1142/S0218196707003755

[24] Volker Diekert. 2015. More Than 1700 Years of Word Equations. In *Algebraic Informatics - 6th International Conference, CAI 2015, Stuttgart, Germany, September 1-4, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9270)*, Andreas Maletti (Ed.). Springer, 22–28. https://doi.org/10.1007/978-3-319-23021-4_2

[25] Volker Diekert and John Michael Robson. 1999. Quadratic Word Equations. In *Jewels are Forever, Contributions on Theoretical Computer Science in Honor of Arto Salomaa*, Juhani Karhumäki, Hermann A. Maurer, Gheorghe Paun, and Grzegorz Rozenberg (Eds.). Springer, 314–326.

[26] Jonathan DiLorenzo, Katie Mancini, Kathleen Fisher, and Nate Foster. 2019. TxForest: A DSL for Concurrent Filestores. In *Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11893)*, Anthony Widjaja Lin (Ed.). Springer, 332–354. https://doi.org/10.1007/978-3-030-34175-6_17

[27] Kathleen Fisher, Nate Foster, David Walker, and Kenny Qili Zhu. 2011. Forest: a language and toolkit for programming with filestores. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM, 292–306. https://doi.org/10.1145/2034773.2034814

[28] Vijay Ganesh, Mia Minnes, Armando Solar-Lezama, and Martin C. Rinard. 2012. Word Equations with Length Constraints: What's Decidable?. In *Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers (Lecture Notes in Computer Science, Vol. 7857)*, Armin Biere, Amir Nahir, and Tanja E. J. Vos (Eds.). Springer, 209–226. https://doi.org/10.1007/978-3-642-39611-3_21

[29] Philippa Gardner, Gian Ntzik, and Adam Wright. 2014. Local Reasoning for the POSIX File System. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8410)*, Zhong Shao (Ed.). Springer, 169–188. https://doi.org/10.1007/978-3-642-54833-8_10

[30] Isaac Oscar Gariano, James Noble, and Marco Servetto. 2019. CallE: An Effect System for Method Calls. *CoRR* abs/1907.04934 (2019). arXiv:1907.04934 http://arxiv.org/abs/1907.04934

[31] David K. Gifford and John M. Lucassen. 1986. Integrating Functional and Imperative Programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming, LFP 1986, Cambridge, Massachusetts, USA, August 4-6 , 1986*, William L. Scherlis, John H. Williams, and Richard P. Gabriel (Eds.). ACM, 28–38. https://doi.org/10.1145/319838.319848

[32] Michael Greenberg and Austin J. Blatt. 2020. Executable formal semantics for the POSIX shell. *Proc. ACM Program. Lang.* 4, POPL (2020), 43:1–43:30. https://doi.org/10.1145/3371111

[33] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The Essence of JavaScript. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6183)*, Theo D'Hondt (Ed.). Springer, 126–150. https://doi.org/10.1007/978-3-642-14107-2_7

[34] Claudio Gutierrez. 1998. Satisfiability of Word Equations with Constants is in Exponential Space. In *39th Annual Symposium on Foundations of Computer Science, FOCS 1998, Palo Alto, California, USA, November 8-11, 1998*. IEEE Computer Society, 112–119. https://doi.org/10.1109/SFCS.1998.743434

[35] Shivam Handa, Konstantinos Kallas, Nikos Vasilakis, and Martin C. Rinard. 2021. An order-aware dataflow model for parallel Unix pipelines. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–28. https://doi.org/10.1145/3473570

[36] Vidar Holen. 2026. ShellCheck. https://www.shellcheck.net/.

[37] Gérard P. Huet. 1997. The Zipper. *J. Funct. Program.* 7, 5 (1997), 549–554. https://doi.org/10.1017/S0956796897002864

[38] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (2001), 396–450. https://doi.org/10.1145/503502.503505

[39] Nicolas Jeannerod and Ralf Treinen. 2018. Deciding the First-Order Theory of an Algebra of Feature Trees with Updates. In *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10900)*, Didier Galmiche, Stephan Schulz, and Roberto Sebastiani (Eds.). Springer, 439–454. https://doi.org/10.1007/978-3-319-94205-6_29

[40] Artur Jeż. 2014. Context Unification is in PSPACE. In *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 8573)*, Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias (Eds.). Springer, 244–255. https://doi.org/10.1007/978-3-662-43951-7_21

[41] Artur Jeż. 2016. Recompression: A Simple and Powerful Technique for Word Equations. *J. ACM* 63, 1 (2016), 4:1–4:51. https://doi.org/10.1145/2743014

[42] Artur Jeż, Anthony W Lin, Oliver Markgraf, and Philipp Rü mmer. 2023. Decision Procedures for Sequence Theories. In *International Conference on Computer Aided Verification*. Springer, 18–40. https://doi.org/10.1007/978-3-031-37703-7_2

[43] Antoni Koscielski and Leszek Pacholski. 1996. Complexity of Makanin's Algorithm. *J. ACM* 43, 4 (1996), 670–684. https://doi.org/10.1145/234533.234543

[44] Frank Landymore. 2026. Blundering Husband Asks Claude AI to "Organize" Wife's PC, Accidentally Erases Her Cherished Family Photos. https://futurism.com/artificial-intelligence/claude-wife-photos.

[45] Lukas Lazarek, Seong-Heon Jung, Evangelos Lamprou, Zekai Li, Anirudh Narsipur, Eric Zhao, Michael Greenberg, Konstantinos Kallas, Konstantinos Mamouras, and Nikos Vasilakis. 2025. From Ahead-of- to Just-in-Time and Back Again: Static Analysis for Unix Shell Programs. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems, HotOS 2025, Banff, AB, Canada, May 14-16, 2025*. ACM, 88–95. https://doi.org/10.1145/3713082.3730395

[46] Lukas Lazarek, Evangelos Lamprou, George Kapetenakis, Eric Zhao, Zhiwen Zheng, Michael Greenberg, Konstantinos Kallas, and Nikos Vasilakis. 2026. Ahead-of-time Analysis of Shell Program Effects. In *Proceedings of the ACM SIGOPS 32nd Symposium on Operating Systems Principles, SOSP 2026, Clarion Congress Hotel Prague, Prague, Czechia, September 29-October 2, 2026 (SOSP '26)*. Association for Computing Machinery, New York, NY, USA.

[47] Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014 (EPTCS, Vol. 153)*, Paul Blain Levy and Neel Krishnaswami (Eds.). 100–126. https://doi.org/10.4204/EPTCS.153.8

[48] Anthony Widjaja Lin and Pablo Barceló. 2016. String solving with word equations and transducers: towards a logic for analysing mutation XSS. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 123–136. https://doi.org/10.1145/2837614.2837641

[49] Anthony W. Lin and Rupak Majumdar. 2018. Quadratic Word Equations with Length Constraints, Counter Systems, and Presburger Arithmetic with Divisibility. In *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11138)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer, 352–369. https://doi.org/10.1007/978-3-030-01090-4_21

[50] John M. Lucassen and David K. Gifford. 1988. Polymorphic Effect Systems. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, Jeanne Ferrante and Peter Mager (Eds.). ACM Press, 47–57. https://doi.org/10.1145/73560.73564

[51] G. S. Makanin. 1977. The problem of solvability of equations in a free semigroup. *Mathematics of the USSR-Sbornik* 32, 2 (1977), 129–198.

[52] Conor McBride. 2001. The Derivative of a Regular Type is its Type of One-Hole Contexts. (2001). http://strictlypositive.org/diff.pdf

[53] John McCarthy. 1962. Towards a Mathematical Science of Computation. In *Information Processing, Proceedings of the 2nd IFIP Congress 1962, Munich, Germany, August 27 - September 1, 1962*. North-Holland, 21–28.

[54] MITRE Corporation. 2024. *CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')*. Common Weakness Enumeration. https://cwe.mitre.org/data/definitions/22.html

[55] MITRE Corporation. 2024. *CWE-23: Relative Path Traversal*. Common Weakness Enumeration. https://cwe.mitre.org/data/definitions/23.html

[56] Gian Ntzik and Philippa Gardner. 2015. Reasoning about the POSIX file system: local update and global pathnames. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA , USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 201–220. https://doi.org/10.1145/2814270.2814306

[57] Wojciech Plandowski. 2004. Satisfiability of word equations with constants is in PSPACE. *J. ACM* 51, 3 (2004), 483–496. https://doi.org/10.1145/990308.990312

[58] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

[59] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. 2015. SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems. In *Proceedings of the 25th Symposium on Operating Systems Principles , SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, Ethan L. Miller and Steven Hand (Eds.). ACM, 38–53. https://doi.org/10.1145/2815400.2815411

[60] Rodney Rodriguez and Xiaoyin Wang. 2021. Understanding Execution Environment of File-Manipulation Scripts by Extracting Pre-Conditions. In *29th IEEE/ACM International Conference on Program Comprehension , ICPC 2021, Madrid, Spain, May 20-21, 2021*. IEEE, 406–410. https://doi.org/10.1109/ICPC52881.2021.00048

[61] Klaus U. Schulz. 1990. Makanin's Algorithm for Word Equations - Two Improvements and a Generalization. In *Word Equations and Related Topics, First International Workshop, IWWERT '90, Tübingen, Germany, October 1-3, 1990, Proceedings (Lecture Notes in Computer Science, Vol. 572)*, Klaus U. Schulz (Ed.). Springer, 85–150. https://doi.org/10.1007/3-540-55124-7_4

[62] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Push-Button Verification of File Systems via Crash Refinement. In *Proceedings of the 2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, Dilma Da Silva and Bryan Ford (Eds.). USENIX Association. https://www.usenix.org/conference/atc17/technical-sessions/presentation/sigurbjarnarson

[63] Gert Smolka and Ralf Treinen. 1994. Records for Logic Programming. *J. Log. Program.* 18, 3 (1994), 229–258. https://doi.org/10.1016/0743-1066(94)90044-2

[64] Snyk Security Team. 2018. *Zip Slip: Arbitrary File Write Through Archive Extraction.* Technical Report. Snyk. https://res.cloudinary.com/snyk/image/upload/v1528192501/zip-slip-vulnerability/technical-whitepaper.pdf

[65] Jean-Pierre Talpin and Pierre Jouvelot. 1994. The Type and Effect Discipline. *Inf. Comput.* 111, 2 (1994), 245–296. https://doi.org/10.1006/INCO.1994.1046

[66] Balder ten Cate and Luc Segoufin. 2010. Transitive closure logic, nested tree walking automata, and XPath. *J. ACM* 57, 3 (2010), 18:1–18:41. https://doi.org/10.1145/1706591.1706598

[67] Ralf Treinen. 1993. Feature Constraints with First-Class Features. In *Mathematical Foundations of Computer Science 1993, 18th International Symposium, MFCS'93, Gdansk, Poland, August 30 - September 3, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 711)*, Andrzej M. Borzyszkowski and Stefan Sokolowski (Eds.). Springer, 734–743. https://doi.org/10.1007/3-540-57182-5_64

[68] GitHub user `keyvin`. 2015. Moved ~/.local/share/steam. Ran steam. It deleted everything on system owned by user. #3671. https://github.com/ValveSoftware/steam-for-linux/issues/3671.

[69] 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

[70] W3C. 2017. XML Path Language (XPath) 3.1. (2017). https://www.w3.org/TR/2017/REC-xpath-31-20170321/

[71] W3C. 2017. XQuery Update Facility 3.0. (2017). https://www.w3.org/TR/2017/NOTE-xquery-update-30-20170124/

[72] Eric Zhao, Zekai Li, Feiyang Wang, Lukas Lazarek, Konstantinos Kallas, Robert Lewis, and Nikos Vasilakis. 2026. *Artifact for Trees, Walks, and Words: A Logic for File System Interactions.* https://doi.org/10.5281/zenodo.23332805