Local Reasoning about POSIX File Systems

Philippa Gardner, Gian Ntzik, and Adam Wright

Imperial College London
p.gardner@imperial.ac.uk
gian.ntzik08@imperial.ac.uk
adam.wright07@imperial.ac.uk

Abstract. We provide a program logic for specifying a core subset of the POSIX file system and for reasoning about client programs.

Keywords: file systems, POSIX, local reasoning, separation logic

1 Introduction

Local reasoning was originally introduced in separation logic [20], where the goal was to provide scalable, modular reasoning about low-level programs for manipulating heaps defined as partial functions from addresses to values. Since then, there has been much work on abstraction. We have separation logic with abstract predicates for reasoning locally about e.g. linked lists [19], concurrent abstract predicates for more abstract reasoning about e.g. sequential and concurrent sets [7, 22], context logic for reasoning abstractly about complex structured data such as the W3C DOM library for XML update [13, 5, 21], and structural separation logic for reasoning about concurrent trees [23, 12].

Despite these recent advances in abstract local reasoning, we still have insufficient technology for reasoning *abstractly* about programs that manipulate complex data by *globally* traversing the data structure to a place where they *locally* update. Such programs are common place. For example, the POSIX file system has an English specification which naturally describes commands which globally follow directory paths to locally update files or directories[1]. We provide a formal specification of a core subset of the POSIX file system, with the aim to be as close to the POSIX English specification as we can. We achieve this by extending structural separation logic with *path promises* that require stable information about where a directory exists within a complete file system. It enables us to provide integrated reasoning about the directory structure and the standard heap within the same logic. We demonstrate this integrated reasoning by verifying safety properties of a client software installer.

File systems are essential for manipulating persistent storage. The most commonly used file-system abstraction standard is POSIX [2]. We spent considerable effort identifying a core subset of sequential POSIX, which is both faithful to the

[1] Both files and directories are called 'files' in POSIX. We use the term 'entries' to denote either directories or files.

standard and a natural subset with which to introduce our reasoning. Various structures of the file-system state are naturally modelled as heaps: file heaps mapping file identifiers (inodes) to bytes; file-descriptor heaps mapping file descriptors to input/output related data. Separation logic can reason about these heap structures. Our challenge is to reason about the directory structure, which we regard as a tree-shaped hierarchy[2], where directory tree nodes are identified by global path resolution and their updates are naturally local.

We reason about this directory tree hierarchy, extending structural separation logic with *path promises*. Structural separation logic (SSL) is a program logic for reasoning locally about complex structured data. With SSL, we can for example reason about DOM commands which have direct access to tree nodes using DOM identifiers [23, 12]. SSL allows fine-grained reasoning about tree fragments using *abstract heaps* which provide instrumentation for separating and joining tree fragments. SSL is a substantial improvement over previous work on context-logic reasoning about DOM. However, basic SSL cannot reason about global paths. We extend SSL with *promises* which declare stable global properties about data structures. The general theory is in Wright's thesis [23]. Here we introduce the extension by studying POSIX, to ground our theory on real application. We provide small axioms which naturally correspond to the POSIX English standard. We provide integrated reasoning about the directory tree structure and the standard heap structures. Although we concentrate on sequential POSIX, our results immediately extend to a disjoint concurrent POSIX since our reasoning is underpinned by the views framework. In future, we will explore shared-memory concurrent POSIX as defined by the standard. We use our reasoning to verify that a client software installer behaves sensibly: if the installer fails, the file system will be unchanged; if it succeeds, the program will be successfully installed; and there will be no other possible outcomes.

Related Work There has been substantial work on the formal specification of file systems [14, 17, 9, 4], even leading to a verification challenge by Joshi and Holzmann [16, 10]. It is not feasible to give a comprehensive account of this work in the space available; such an account will be in Ntzik's thesis [18]. Here, we concentrate on demonstrating the advantages of our local directory tree reasoning approach compared to reasoning about global directory trees and heap structures with paths as addresses.

A natural question is whether we can use first-order reasoning about a global tree directory, as in [8]. However, this leads to scalability problems. Consider the incorrect specification of the `rename(p/a, p'/b)` command:

$$\{\, resolve(p, d[t + a[t']]) \wedge resolve(p', d'[t'' \wedge \neg exists(b)]) \,\}$$
`rename(p/a, p'/b)`
$$\{\, resolve(p, d[t]) \wedge resolve(p', d'[t'' + b[t']]) \,\}$$

[2] In general, files and directories can be hard linked more than once. Most implementations only allow files to be linked more than once. This is a sensible choice as, for example, cycles generated by directory hard links are not detected by recursive traversal programs. We therefore regard POSIX as a tree-shaped hierarchy.

$resolve(p, d[t + a[t']])$ states that path p resolves to the directory d containing an arbitrary forest t and the directory a; also, $resolve(p', d'[t'' \wedge \neg exists(b)])$ states that p' resolves to directory d' with no b entry. This precondition is incorrect because POSIX specifies an error when p' is a descendant of p/a: that is, p' is $p/a/p''$ for some path p''. To correct the specification we require a syntactic path check to identify this error case. Now consider the command `rename(p/a, p'/b) ; rename(p''/c, p'''/d)`. In this case, we need syntactic checks that p' is not a descendant of p/a, p''' is not a descendant of p''/c, and also p''/c is not a descendant of p/a. It is evident that this style of reasoning does not scale. Those familar with separation logic might recognise that this example is analogous to Reynolds' list example for justifying separation logic.

A completely different approach, used in much of the work on the formal specification of file systems using Z [17] and other methods as [14], is to treat paths as heap addresses. Define the set of heaps as $\textsc{Paths} \xrightarrow{\text{fin}} \textsc{Bytes} \cup \mathcal{P}(\textsc{FNames})$, mapping paths to byte sequences in the file case, or sets of names in the directory case. This approach requires significant global constraints: for example, for every path in the domain mapping to a directory, paths to every descendant must also be in the domain and must be consistent with each other. This makes the specification of `rename(p/a, p'/b)` even more complicated, as it not only involves updating the heap cell of path p/a, but also the heap cells of all the descendants to ensure the validity of their path addresses.

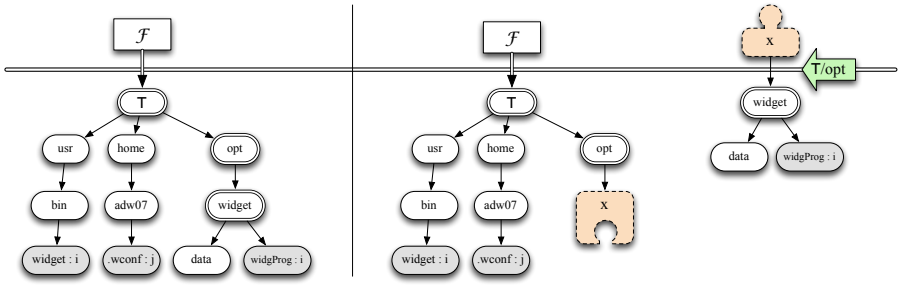

Fig. 1. The left-hand diagram represents a complete directory; the right, the same directory instrumented with abstract addresses.

2 Example Specifications

A substantial part of the POSIX standard describes the file system. We focus on a core fragment of 16 commands in this paper. Although small, it includes most primitive commands that manipulate the structure and perform input-output (IO). In fact, a large proportion of the file system commands can be implemented using our core commands. This means that we can derive specifications of these other commands without additional machinery, as we shall demonstrate.
First consider the English description of the `rmdir` command[3]:

[3] This description only presents the case when the operation succeeds. When the command fails, for example if `path` does not identify an existing file or directory, the

[`r := rmdir(path)`] Remove the directory identified by `path` and set `r` to 0. The directory must be empty.

Intuitively, this command acts *globally* by traversing the global path `path` to identify the location of the update, and it acts *locally* by removing the empty directory whilst leaving the rest of the file system unchanged. We capture this combination of global and local behaviour using structural separation logic extended with *promises*. Consider figure 1. The left hand side illustrates part of a structured heap, consisting of a heap cell $\mathcal{F}$ whose structured value is a complete directory tree. The right hand side illustrates part of an abstract heap, consisting of the heap cell $\mathcal{F}$ whose value is now an incomplete directory tree with *body address* x and an abstract heap cell with *abstract cell address* x whose value is a pair consisting of a complete subdirectory and a *path promise* '$\top/opt$' providing the stability condition that body address x must be at the end of '$\top/opt$'. This promise to x allows us to reason locally about the abstract heap cell x whilst retaining the knowledge of its location in the global structure. Notice that the two heaps illustrated in figure 1 describe the same state, in that they differ only in the instrumentation added by abstract addresses. When we move from the left to the right hand side view of the state we apply *abstract allocation* in that we allocate a new abstract heap cell, while the converse is *abstract deallocation*.

With SSL and path promises, we can reason about such abstract heap cells. Our assertion $\alpha^{\mathrm{P}} \mapsto \mathrm{A}[\varnothing]$ describes the ownership of the abstract heap cell denoted by logical variable α containing an empty directory $\mathrm{A}[\varnothing]$ and assocaited with path promise P. Using it, we are able to provide an axiomatic specification of the `rmdir` command:

$$\left\{\texttt{path} \Rightarrow \mathrm{P/A} \wedge \texttt{r} \Rightarrow - \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \mathrm{A}[\varnothing]\right\} \texttt{r:=rmdir(path)} \left\{\texttt{r} \Rightarrow 0 \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \varnothing\right\}$$

As well as the assertion for the abstract heap cell at α, the precondition contains assertions about the store, derived from the standard variables as resource approach [3]. The pure *expression assertion* $\texttt{path} \Rightarrow \mathrm{P/A}$ states that expression `path` has logical expression P/A as its value. This logical expression describes an arbitrary path P followed by the directory or file name A. Similarly, the *variable assertion* $\texttt{r} \Rightarrow -$ states that the program variable `r` has some arbitrary value. The postcondition states that variable `r` now has value 0, whilst abstract heap cell α is empty with the path promise P. The additional variable resource predicate $\mathcal{E}$ captures any additional variable resource needed to evaluate the expression `path`, and is unchanged between the pre and postconditions.

This specification is *small* in that the precondition intuitively describes local ownership of the minimum resource needed to safely run the command: the variables `r` and those needed to evaluate `path` given by $\mathcal{E}$; and the abstract cell address α with the directory being updated. It also describes the global information that only directories satisfying path promise P associated with α can be framed on. To illustrate this, consider the directory $\mathcal{F} \mapsto \top[\mathrm{C} + \mathrm{D}[\mathrm{A}[\varnothing]]]$, the path $\mathrm{P} = \top/\mathrm{D}/\mathrm{A}$ and the proof derivation:

---

result is to assign -1 to `r` and set the global variable `errno` to ENOENT. We discuss error specifications in the appendix.

$\{\texttt{path} \Rightarrow \top/\mathrm{D}/\mathrm{A} \star \texttt{r} \Rightarrow - \star \mathcal{E} \star \mathcal{F} \mapsto \top[\mathrm{C} + \mathrm{D}[\mathrm{A}[\varnothing]]]\}$
// abstract allocation
$\{\texttt{path} \Rightarrow \top/\mathrm{D}/\mathrm{A} \star \mathcal{E} \star \texttt{r} \Rightarrow - \star \exists\alpha.\ (\mathcal{F} \mapsto \top[\mathrm{C} + \mathrm{D}[\alpha]] \star \alpha^{\top/\mathrm{D}} \mapsto \mathrm{A}[\varnothing])\}$
// existential elimination and frame rule
// and apply the axiom
$\{\texttt{path} \Rightarrow \top/\mathrm{D}/\mathrm{A} \star \texttt{r} \Rightarrow - \star \mathcal{E} \star \alpha^{\top/\mathrm{D}} \mapsto \mathrm{A}[\varnothing]\}$
`r := rmdir(path)`
$\{\texttt{path} \Rightarrow \top/\mathrm{D}/\mathrm{A} \star \texttt{r} \Rightarrow 0 \star \mathcal{E} \star \alpha^{\top/\mathrm{D}} \mapsto \varnothing\}$
// existential, frame rule reapplication
$\{\texttt{path} \Rightarrow \top/\mathrm{D}/\mathrm{A} \star \texttt{r} \Rightarrow 0 \star \mathcal{E} \star \exists\alpha.\ (\mathcal{F} \mapsto \top[\mathrm{C} + \mathrm{D}[\alpha]] \star \alpha^{\top/\mathrm{D}} \mapsto \varnothing)\}$
// abstract deallocation
$\{\texttt{r} \Rightarrow - \star \mathcal{E} \star \mathcal{F} \mapsto \top[\mathrm{C} + \mathrm{D}[\varnothing]]\}$

The initial precondition contains a directory hierarchy beginning at the file system root $\top$ with arbitrary contents captured by the logical variable C and a directory named D that contains just the empty directory A. This does not match the precondition of `rmdir`, and so we take the following steps. First, we abstractly allocate a new abstract heap cell containing the A directory. Then, we apply the standard Hoare logic existential elimination to set aside the existential binding of α, and use the frame rule to set aside what `rmdir` does not need. We now match `rmdir`'s precondition, where $\top/\mathrm{D}$ is P. Afer applying the axiom we can reintroduce the resource and binding set aside with frame and existential elimination, and abstractly deallocate the cell with address α.
Now consider the `unlink` command and its English specification:

[`r := unlink(path)`] Remove the link to the file identified by `path`.

Again, using SSL we can formalise the English specification with the following small axiom:

$$\{\texttt{path} \Rightarrow \mathrm{P}/\mathrm{A} \wedge \texttt{r} \Rightarrow - \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I}\}\ \texttt{r:=unlink(path)}\ \{\mathcal{E} \star \texttt{r} \Rightarrow 0 \star \alpha^{\mathrm{P}} \mapsto \varnothing\}$$

In the precondition, $\alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I}$ states that a file named A is found at abstract cell address α at the end of path P. Note that the contents of the file are not included in the precondition. When the last link to a file is removed, the file will no longer be accessible by any path, and we assume a garbage collection step will remove any associated file data. Notice that we do *not* remove the abstract heap cell α. If the axiom destroyed this cell, the associated α body address (which must exist in some data in the frame) would have no matching cell address. This would break the stability of the system, where a cell address always match with a body address at the appropriate path promise.

Finally, consider the `stat` command, which returns meta-data about the file or directory identified by the path argument. In this paper, we take that meta-data to be just the file type, D for directory and F for file. There is one axiom for each file type; the directory case is:

$$\{\texttt{path} \Rightarrow \mathrm{P}/\mathrm{A} \wedge \texttt{t} \Rightarrow - \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \mathrm{A}[\beta]\}\ \texttt{t:=stat(path)}\ \{\texttt{t} \Rightarrow \mathrm{D} \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \mathrm{A}[\beta]\}$$

Notice that the specification uses *body address* β in $\alpha^{\mathrm{P}} \mapsto \mathrm{A}[\beta]$ to specify that the content of A is not changed by the command. It does not need more de-

tailed knowledge of the contents of A since the command does not require this knowledge to determine that the entry is a directory.

The commands discussed in this section are enough to derive another POSIX command: `r := remove(path)`. According to its POSIX description, this command removes the file or empty directory identified by the `path` argument. In figure 2 we implement `remove` and derive its specification using the `stat`, `unlink` and `rmdir` commands discussed earlier. The specification that we derive is:

$$\{\, \mathtt{path} \Rightarrow \mathrm{P/A} \wedge \mathtt{r} \Rightarrow - \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto (\mathrm{A} : \mathrm{I} \vee \mathrm{A}[\varnothing]) \,\}$$
$$\mathtt{r := remove(path)}$$
$$\{\, \mathtt{r} \Rightarrow 0 \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \varnothing \,\}$$

Note that this matches exactly the English description obtained from the POSIX standard. Following the same process, we can use the core fragment of this paper to "discover" formal specifications of many more complex commands of POSIX.

$\{\, \mathtt{path} \Rightarrow \mathrm{P/A} \wedge \mathtt{r} \Rightarrow - \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto (\mathrm{A} : \mathrm{I} \vee \mathrm{A}[\varnothing]) \,\}$

```
r := remove(path) ≜ local t {
  t := stat(path);
```

$\{\, \exists \mathrm{T}.\ \mathtt{path} \Rightarrow \mathrm{P/A} \wedge \mathtt{r} \Rightarrow - \star \mathtt{t} \Rightarrow \mathrm{T} \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto (\mathrm{A} : \mathrm{I} \wedge \mathrm{T} = \mathrm{F}) \vee (\mathrm{A}[\varnothing] \wedge \mathrm{T} = \mathrm{D}) \,\}$

```
  if t = F
```

$\{\, \mathtt{path} \Rightarrow \mathrm{P/A} \wedge \mathtt{r} \Rightarrow - \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I} \,\}$

```
    r := unlink(path);
```

$\{\, \mathtt{path} \Rightarrow \mathrm{P/A} \wedge \mathtt{r} \Rightarrow 0 \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \varnothing \,\}$

```
  else if t = D
```

$\{\, \mathtt{path} \Rightarrow \mathrm{P/A} \wedge \mathtt{r} \Rightarrow - \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \mathrm{A}[\varnothing] \,\}$

```
    r := rmdir(path);
```

$\{\, \mathtt{path} \Rightarrow \mathrm{P/A} \wedge \mathtt{r} \Rightarrow 0 \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \varnothing \,\}$

```
  else r := -1;
```

$\{\, \mathtt{r} \Rightarrow 0 \star \mathtt{t} \Rightarrow - \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \varnothing \,\}$

```
}
```

$\{\, \mathtt{path} \Rightarrow \mathrm{P/A} \star \mathtt{r} \Rightarrow 0 \star \alpha^{\mathrm{P}} \mapsto \varnothing \,\}$

Fig. 2. Implementation of `remove` and proof that it meets its specification.

3 File System Specification

We provide an axiomatic specification of our POSIX commands, based on structural separation logic extended with naturally stable promises for describing update on abstract heaps. An abstract program state comprises: an abstract *file-system heap*, which represents the directory tree and associated stored files, as might intuitively reside on a hard disk; a *process heap*, which represents the contents of computer memory during execution; and a *variable store*, which represents the values of program variables.

3.1 File-system Heap

Abstract file-system heaps are abstract heaps whose cells contain partial directories. Directories are defined using a set of *inodes* INODES, ranged over by

$\iota, \kappa, \cdots$, and a set of file names FNAMES, ranged over by $\mathrm{A}, \mathrm{B}, \cdots$, for naming directories and files. Both sets are defined as in POSIX. Our partial directories are instrumented by body addresses (context holes), drawn from the countably infinite set of abstract addresses ABSADDRS, ranged over by $\mathbf{x}, \mathbf{y}, \mathbf{z}, \cdots$, with $(\text{FNAMES} \cup \{\mathcal{F}\} \cup \text{INODES}) \cap \text{ABSADDRS} = \varnothing$ where $\mathcal{F}$ is the distinguished address of the root directory.

Definition 1 (Directories). *The set of* unrooted directories, UDIRS, *is:*

$$ud ::= \varnothing \mid a : \iota \mid a[ud] \mid ud + ud \mid \mathbf{x}$$

where $\varnothing$ is the empty list of entries, $a : \iota$ is a file link associating file name $a \in$ FNAMES with inode $\iota \in$ INODES, $a[ud]$ is a directory named a containing unrooted abstract directory ud, $+$ is directory composition and $\mathbf{x} \in$ ABSADDRS is a body address. The directories have sibling-unique names, body addresses are unique, and $+$ is commutative and associative with identity $\varnothing$.

There is a distinguished $\top \notin$ FNAMES representing the root directory of the file-system tree. The set of rooted directories, RDIRS, *is defined as* $\text{RDIRS} \triangleq \{\top[ud] \mid ud \in \text{UDIRS}\}$. *The set of* directories, $d \in$ DIRS, *is defined by* $\text{DIRS} \triangleq \text{UDIRS} \cup \text{RDIRS}$. *Each directory entry has a type* $\text{DETYPES} \triangleq \{\mathrm{F}, \mathrm{D}\}$*: the* F *denotes a hard link to a file, and* D *a directory.*

Each body address can be replaced with a directory via *context application*.

Definition 2 (Context application). *The* addresses function, *addrs* : DIRS $\to \mathcal{P}(\text{ABSADDRS})$ *describes the set of body addresses in a directory. Context application is the function* $\circ$: ABSADDRS $\to$ (DIRS $\to$ UDIRS) $\rightharpoonup$ DIRS *defined by:*

$$d_1 \circ_{\mathbf{x}} ud_2 = \begin{cases} d_1[ud_2/\mathbf{x}] & \mathbf{x} \in \text{addrs}(d_1) \wedge \text{addrs}(d_1) \cap \text{addrs}(ud_2) \subseteq \{\mathbf{x}\} \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

where $d_1[ud_2/\mathbf{x}]$ is the substitution of ud_2 for $\mathbf{x}$ in d_1. The function is only defined only if the result is in DIRS.

Many POSIX commands refer to entries in the file system tree by *absolute (or rooted) paths* through the directory tree. In general, POSIX paths are complex with e.g. backwards paths (..) and symbollic links. In this paper, we work with simple *linear paths*, just called paths for this paper, where the path structure matches the inductive directory structure. In fact, general POSIX paths will be very interesting for our overall agenda of local reasoning about complex data structures, since they walk right across the structure. We should be able to handle general paths using a combination of promises and *obligations* discussed in the conclusions: the abstract address $\mathbf{x}$ will have the promise that the part of the path in the context is stable, and the obligation to keep the part of the path in the context stable.

Definition 3 (Paths and Resolution). *The set of* relative paths, RELPATHS, *is defined by:*

$$rp ::= \epsilon \mid a \mid rp/rp$$

where $a \in$ FNAMES and the path composition $/$ is associative with identity ϵ. The set of absolute paths *is* $\text{ABPATHS} \triangleq \{\top\} \cup \{\top/rp \mid rp \in \text{RELPATHS}\}$. *The set of*

abstract paths *is* $\textsc{AbsPaths} = \{p/\mathbf{x} \mid p \in \textsc{AbPaths}, \mathbf{x} \in \textsc{AbsAddrs}\}$. *The set of* paths, $p \in \textsc{Paths}$, *is* $\textsc{Paths} \triangleq \textsc{RelPaths} \cup \textsc{AbPaths} \cup \textsc{AbsPaths}$.

The path resolution *function* $\text{resolve} : \textsc{Paths} \times \textsc{Dirs} \rightharpoonup \textsc{Dirs}$ *is defined by:*

$$\begin{array}{lll} \text{resolve}(a, d + a : \iota) & = a : \iota & \text{resolve}(a/rp, d_1 + a[d_2]) = \text{resolve}(rp, d_2) \quad \textit{if } rp \notin \epsilon \\ \text{resolve}(a, d_1 + a[d_2]) & = a[d_2] & \text{resolve}(\top, \top[d]) = \top[d] \\ \text{resolve}(\mathbf{x}, \mathbf{x} + a[d_2]) & = \mathbf{x} & \text{resolve}(\top/rp, \top[d]) = \text{resolve}(rp, d) \quad \textit{if } rp \notin \epsilon \end{array}$$

In all other cases, the result is undefined.

A file-system heap is the union of three finite partial functions: from distinguished address $\mathcal{F}$ to the root directory which might be partial; from abstract addresses to absolute paths (expressing where the corresponding body address lies) and directories; and from inodes to byte sequences representing file contents. We construct file-system heaps in two phases: first, we define *pre-file-system heaps*; then, we define well-formedness conditions to give the full definition.

Definition 4 (Pre-file-system Heap). *Let* Bytes *be the set of finite byte sequences. A* pre-file-system heap, $pfs \in \textsc{PreFS}$, *is a function in the set*

$$(\{\mathcal{F}\} \rightharpoonup \{\epsilon\} \times \textsc{RDirs}) \sqcup (\textsc{AbsAddrs} \overset{fin}{\rightharpoonup} \textsc{AbPaths} \times \textsc{Dirs}) \sqcup (\textsc{Inodes} \overset{fin}{\rightharpoonup} \textsc{Bytes})$$

Let $inodes(d)$ *denote the set of all inodes occurring in directory* d. *A pre-file-system-heap, pfs, is* complete *if:* $dom(pfs) \cap \textsc{AbsAddrs} = \varnothing$; $pfs(\mathcal{F}) = (\epsilon, rd)$; $addrs(rd) = \varnothing$; *and* $inodes(rd) \subseteq \mathbf{dom}(pfs)$[4].

Pre-file-system heaps may use abstract addresses incorrectly. For example, two separate partial directories at different addresses may contain the same body address, or the path promises may not correctly identify the location of the directory. We define a *collapse relation*, with which we give a well-formedness condition that ensures addresses are used correctly. The collapse relation intuitively states that we can connect a cell address to the matching body address with context application, if the paths match, as illustrated in figure 3.

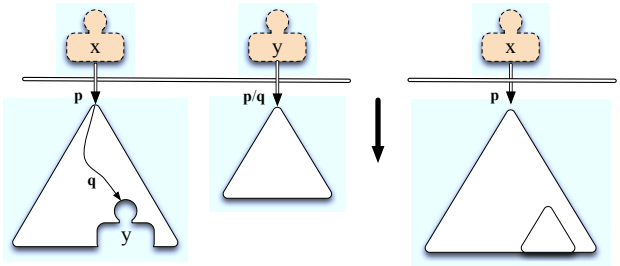

Fig. 3. Collapse relation

Definition 5 (Collapse Relation). *The one-step collapse relation,* $\downarrow \subseteq \textsc{PreFS} \times \textsc{PreFS}$, *relates* $pfs_1 \downarrow pfs_2$ *if and only if there is some address* $\mathbf{addr} \in \textsc{AbsAddrs} \cup \{\mathcal{F}\}$ *and unique* $\mathbf{y} \in \textsc{AbsAddrs}$ *such that:*

1. $pfs_1(\mathbf{addr}) = (p, d)$ *and* $pfs_1(\mathbf{y}) = (p_{\mathbf{y}}, d_{\mathbf{y}})$;

[4] Complete pre-file-system-heaps are thus simple DAGs, with sharing occurring only at the leaves in the sense that two separate file names can point to the same inode.

2. $\mathbf{y} \in addrs(d)$;
3. *there is some* $q \in$ PATHS *such that* $p_{\mathbf{y}} = p/q$;
4. $\text{resolve}(q, d) = \mathbf{y}$;
5. $pfs_2 = pfs_1[\mathbf{addr} \mapsto (p, d \circ_{\mathbf{y}} d_{\mathbf{y}})]/\mathbf{y}$ [5].

Let $\downarrow^*$ *be the reflexive, transitive closure of* $\downarrow$.

Using collapse, we can detect all pre-file-system heaps that use invalid addressing. Given *pfs*, the correct use of abstract addressing falls into three cases:

1. *pfs* is complete, and is thus trivially uses abstract addresses correctly.
2. *pfs* uses abstract addresses, but is related via collapse to a complete abstract file system. In this case, the complete system it is related to must be unique (see [23] for details).
3. *pfs* uses abstract addresses, but is not immediately related to a complete file system. However, at least one other pre-abstract file system pfs' can be found such that the union of the two *does* collapse to a complete file system (as in case 2). In this case, *pfs* is a *partial* file-system heap, missing some data, but still using abstract addressing in a consistent way.

With the collapse relation, we can define file-system heaps.

Definition 6 (File-system Heaps). *The set of* file-system heaps, FS *ranged by fs, is defined as:*

$$\text{FS} = \left\{ pfs \in \text{PREFS} \;\middle|\; \exists pfs', pfs'' \in \text{PREFS}.\ pfs \sqcup pfs' \downarrow^* pfs'' \wedge pfs'' \text{ is complete} \right\}$$

3.2 Process Heap

The *process heap* represents the contents of the heap during program execution. It contains structures used for controlling access to files and directories: *open file descriptions* and *directory streams*. An open file description is a record holding information that controls file accesses: the inode and current offset of an open file. It is used to support the commands `read`, `write`, `lseek` and `close`. The heap addresses of open file descriptions, in POSIX terminology called *file descriptors*, are given by the set OFADDRS and ranged by $f, g, \ldots$.

A directory stream is an abstract data structure that captures the set of the entries in a given directory and supports the `opendir`, `readdir` and `closedir` commands. For example, when `opendir(p)` is used, a fresh *directory stream address* from the set DSADDRS is allocated and mapped to a directory stream, which provides a snapshot of the entry names in the directory given by path `p`. Here, we deviate from POSIX. `readdir` returns the names of entries contained within a directory. POSIX allows a high degree of non-determinism when using `readdir` on a directory whilst modifying its contents. One may see some changes, all changes, or none. We adopt a *snapshot semantics*. A copy of the set of entries is taken when `opendir` is used. Calls to `readdir` return elements of that set in a non-deterministic order, but will observe no future changes.

Definition 7 (Process Heaps). *A process heap, denoted* $\text{ph} \in \text{PH}$*, is a partial function in the set* $(\text{DSADDRS} \xrightarrow{fin} \mathcal{P}(\text{FNAMES})) \sqcup (\text{OFADDRS} \xrightarrow{fin} (\text{INODES} \times \mathbb{N}))$

[5] That is, pfs_2 is equal to the function obtained from pfs_1 by removing $\mathbf{y}$ from the domain, mapping **addr** to $(p, d \circ_{\mathbf{y}} d_{\mathbf{y}})$, and leaving the other mappings the same.

3.3 Programming Language

To write programs that use our POSIX subset, we define a standard imperative sequential WHILE language. Variables are assigned values through a *variable store*, $\sigma : \textsc{Vars} \rightharpoonup \textsc{Values}$, with the set of variable stores denoted Σ. Variables are dynamically typed, with values drawn from the set:

$$\textsc{Values} \triangleq \begin{array}{c} \mathbb{Z} \uplus \{\textbf{true}, \textbf{false}\} \uplus \textsc{RelPaths} \uplus \textsc{AbPaths} \uplus \textsc{Bytes} \uplus \textsc{Inodes} \\ \uplus\ \textsc{OFAddrs} \uplus \textsc{DSAddrs} \uplus \textsc{DETypes} \end{array}$$

Program expressions are used as the rvalue of assignments and as parameters to control-flow commands. They consist of the standard literals, variable lookup, arithmatic and boolean operations, and three file-system specific expressions: path concatenation, Expr/Expr; base name extraction `base`(Expr); and directory name extraction `dir`Expr. Expression evaluation $[\![\cdot]\!]_\sigma : \textsc{Expr} \to \Sigma \rightharpoonup \textsc{Values}$ is mostly standard, with the file system specific evaluations being:

$$\begin{array}{rll} [\![\text{Expr/Expr}']\!]_\sigma \triangleq [\![\text{Expr}]\!]_\sigma / [\![\text{Expr}']\!]_\sigma & \text{iff } [\![\text{Expr}']\!]_\sigma \notin \textsc{AbPaths} \\ [\![\texttt{base}(\text{Expr})]\!]_\sigma \triangleq a & \text{iff } [\![\text{Expr}]\!]_\sigma \equiv \top/rp/a \wedge a \notin \epsilon \\ [\![\texttt{dir}(\text{Expr})]\!]_\sigma \triangleq rp & \text{iff } [\![\text{Expr}]\!]_\sigma \equiv \top/rp/a \wedge a \notin \epsilon \\ [\![\texttt{base}(\text{Expr})]\!]_\sigma \triangleq \top & \text{iff } [\![\text{Expr}]\!]_\sigma \equiv \top \\ [\![\texttt{dir}(\text{Expr})]\!]_\sigma \triangleq \top & \text{iff } [\![\text{Expr}]\!]_\sigma \equiv \top \end{array}$$

Together, file system heaps, process heaps and variable stores form the *abstract program states* of file system programs.

Definition 8 (Abstract Program States). *The set of* abstract program states*, as* $\in \textsc{AStates}$*, is defined as:* $\textsc{AStates} \triangleq \text{FS} \times \textsc{Ph} \times \Sigma$

Our core POSIX commands can be classified into *structural* commands that manipulate the file system structure, *primitive IO* commands that read and write the contents of files, and *state* commands for querying the type of files.

Definition 9 (Core Fragment & Programming Language). *The core POSIX fragment consists of structural commands* $\mathbb{C}_{Str} \in \textsc{Comm}_{Str}$*, IO commands* $\mathbb{C}_{IO} \in \textsc{Comm}_{IO}$*, and state commands* $\mathbb{C}_{Stat} \in \textsc{Comm}_{Stat}$*:*

$$\begin{array}{rl} \mathbb{C}_{Str} ::= & \texttt{r := mkdir(path)} \mid \texttt{r := rmdir(path)} \mid \texttt{r := link(existing, new)} \\ & \mid \texttt{r := unlink(path)} \mid \texttt{r := rename(old, new)} \\ \mathbb{C}_{IO} ::= & \texttt{dir := opendir(path)} \mid \texttt{fn := readdir(dir)} \mid \texttt{closedir(dir)} \\ & \mid \texttt{fd := open(path, flags)} \mid \texttt{buffer := read(fd, size)} \\ & \mid \texttt{size := write(fd, buffer)} \\ & \mid \texttt{offset' := lseek(fd, offset, whence)} \\ & \mid \texttt{close(fd)} \\ \mathbb{C}_{Stat} ::= & \texttt{t := stat(path)} \end{array}$$

The commands, $\mathbb{C} \in \textsc{Comm}$*, of the programming language are:*

$$\begin{array}{rl} \mathbb{C} ::= & \texttt{var :=} \text{ Expr} \mid \texttt{local var in } \mathbb{C} \mid \texttt{if} \text{ Expr } \texttt{then } \mathbb{C} \texttt{ else } \mathbb{C} \\ & \mid \texttt{while} \text{ Expr } \texttt{do } \mathbb{C} \mid \texttt{skip} \mid \mathbb{C} \texttt{ ; } \mathbb{C} \mid \mathbb{C}_{Str} \mid \mathbb{C}_{IO} \mid \mathbb{C}_{Stat} \end{array}$$

In POSIX, the commands are specified as C function interfaces. Here, we adapt them to a simple imperative programming style for simplicity. Details relating to the semantics of C are thus abstracted.

3.4 Program Logic

We describe our *program logic* for reasoning about POSIX programs, extending structural separation logic [23, 12] with naturally stable promises for describing update on abstract heaps.[6] Analogous to programs using variables and expressions, assertions use logical variables and expressions. Logical variables are mapped to values by a logical environment, $e \in \textsc{LEnv}$, extending program values with directories, paths, abstract addresses, and sets of these values. Logical expressions, denoted by E, E', are defined and evaluated similarly to program expressions, disallowing program variables. We denote logical variables with block capitals $\mathrm{A}, \mathrm{B}, \mathrm{X}, \mathrm{Y}, \ldots$, except for abstract address variables denoted $\alpha, \beta, \ldots$.

Abstract Directory Cell	$\alpha^E \mapsto \phi$	Empty Entry	$\varnothing$
File Cell	$\mathrm{I} \overset{\mathrm{F}}{\mapsto} E$	File Type Entry	$E : I$
Open File Descriptor Cell	$\mathrm{X} \overset{\mathrm{OF}}{\mapsto} (\mathrm{I}, E)$	Directory Type Entry	$E[\phi]$
Directory Stream Cell	$\mathrm{X} \overset{\mathrm{DS}}{\mapsto} E$	File System Root	$\top[\phi]$
Normal Heap Cell	$E \overset{\mathrm{H}}{\mapsto} E$	Logical Expression	E
Variable Value	$\mathtt{var} \Rightarrow E$	Entry List	$\phi + \phi$
Expression Value	$\mathrm{Expr} \Mapsto E$	Context Application	$\phi \circ_\alpha \phi$
		Path Resolution	$@E$

Fig. 4. Assertion language

Assertions, $P, Q \in \textsc{Asrts}$, are constructed from: standard first order logic connectives and quantifiers; the *separating conjunction* of separation logic, $P \star Q$, and its unit, emp; and the specific assertions of figure 4 describing file-system heaps, process heaps and variable stores. The key assertion is the directory cell assertion, $\alpha^E \mapsto \phi$, which combines *local* information ϕ about the partial directory at α, and *global* information about the environment using path promise E. It states that, at abstract cell address given by α, there is a partial subdirectory satisfying directory assertion ϕ (to be explained) which can be rejoined with the main directory using body address α which must be at the end of path expression E. The splitting and joining of partial directories gives rise to novel allocation and deallocation assertion axioms, which we discuss after introducing the remaining assertions.

The file cell assertion, $\mathrm{I} \overset{\mathrm{F}}{\mapsto} E$, describes the file with inode address given by the logical variable I and contents given by the byte sequence described by logical expression E. The next three cell assertions describe elements of the process heap and are directly lifted from definition 7. The final two describe the contents of the variable store. The assertion $\mathtt{var} \Rightarrow E$, as described previously in section 2, maps the program variable `var` to the value given by the logical expression E. Our core program commands accept parameters given by program expressions. The assertion $\mathrm{Expr} \Mapsto E$ states that the program expression Expr evaluates to

[6] We have been asked whether ramified separation logic for reasoning about dags might be worth exploring [15]. It uses the sepish connective to say that there is possibly some shared dag structure, but where it is not determined. Here, the dag structure is fully determined at the leaves, so ramified separation logic is not appropriate.

the value of the logical expression E. This expression evaluation requires that we own all the variables used in the expression. Since in an arbitrary expression the variables are unknown, we will typically use this assertion in conjunction with an *exact* assertion $\mathcal{E}$, leading to assertions of the form $\mathrm{Expr} \Mapsto E \wedge \mathcal{E}$, where $\mathcal{E}$ captures all the variable resource required to evaluate Expr.

Directory assertions, $\phi, \psi \in \textsc{DirAsrts}$, are constructed from standard first-order connectives and quantifiers, and the assertions of figure 4 describing the structure of directories, context application and path resolution. Most have been directly lifted from the structure of directories (definition 1). The *context application* assertion, $\phi \circ_\alpha \psi$, taken from context logic, describes a directory tree that can be separated into an partial directory satisfying ϕ and containing abstract body address α, and a partial directory satisfying ψ. The assertion $@E$ describes directories in which the path given by E resolves.

Definition 10 (Derived Assertions). *The standard first-order logic assertions are derived from* $\Rightarrow$ *and* **false**. *Additionally, we define the following:*

$$\Diamond\phi \triangleq \exists\alpha.\ (\mathbf{true} \circ_\alpha \phi) \qquad \oplus\phi \triangleq \mathbf{true} + \phi$$
$$\mathit{complete} \triangleq \neg\exists\alpha.\ \Diamond\alpha \qquad \mathit{top\_complete} \triangleq \neg\exists\alpha.\ \oplus\alpha$$
$$\mathit{entry}(\mathrm{A}) \triangleq \mathrm{A}[\mathbf{true}] \vee \exists \mathrm{I}.\ (\mathrm{A} : \mathrm{I}) \qquad \mathit{top}(\phi) \triangleq \phi \wedge \mathit{top\_complete}$$
$$\mathit{can\_create}(\mathrm{A}) \triangleq (\neg \oplus \mathit{entry}(\mathrm{A})) \wedge \mathit{top\_complete}$$
$$\mathit{names}(\mathrm{S}) \triangleq \forall \mathrm{A}.\ (\mathrm{A} \in \mathrm{S} \iff \oplus\mathit{entry}(\mathrm{A})) \wedge \mathit{top\_complete}$$

The assertion $\Diamond\phi$ is read "somewhere ϕ", and describes directories containing some directory satisfying ϕ. The assertion $\oplus\phi$ is similar, restricted to siblings. The assertion *complete* describes directories that do not contain any abstract body addresses and thus no subdirectory is missing; *top_complete* is similar, but restricted to siblings. The assertion $\mathit{top}(\phi)$ states that the directory entries satisfy ϕ, and that no sibling entries have been split away. The assertion $\mathit{can\_create}(\mathrm{A})$ states that an entry named A can be safely created at the current sibling level (used for commands that create new entries such as `mkdir`). Finally, $\mathit{names}(\mathrm{S})$ states that every name in the set S is present as an entry.

A crucial part of the reasoning is the *abstract allocation and deallocation* axioms. They are similar to normal heap allocation and deallocation axioms, but instead of introducing and deleting fresh heap cells, they introduce and delete abstract heap cells in order to split and recombine partial directories. They are essential for our local reasoning about directories, and are only possible due to the recent technological advances of the views framework [6]. For uniformity, we give these as axioms over the *id* command, which has no operational effect. It is a technical device to enable axioms to be used whenever required.

Definition 11 (Abstract allocation and deallocation axioms). *The axioms for* abstract allocation *and* abstract deallocation *are, respectively:*

$$\left\{\alpha^{\mathrm{P}} \mapsto \left((\phi_1 \wedge @_{\mathrm{Q}}/\beta) \circ_\beta \phi_2\right)\right\}\ id\left\{\exists\gamma.\left(\alpha^{\mathrm{P}} \mapsto (\phi_1 \wedge @_{\mathrm{Q}}/\beta) \circ_\beta \gamma * \gamma^{\mathrm{P/Q}} \mapsto \phi_2\right)\right\}$$

$$\left\{\exists\gamma.\left(\alpha^{\mathrm{P}} \mapsto (\phi_1 \wedge @_{\mathrm{Q}}/\beta) \circ_\beta \gamma * \gamma^{\mathrm{P/Q}} \mapsto \phi_2\right)\right\}\ id\left\{\alpha^{\mathrm{P}} \mapsto \left((\phi_1 \wedge @_{\mathrm{Q}}/\beta) \circ_\beta \phi_2\right)\right\}$$

The first axiom is *abstract allocation*. The precondition states that there is a partial directory at cell α with path promise P. This partial directory can

be viewed as two separate parts: the context directory described by ϕ_1 which contains a relative path Q ending in body address β; and the subdirectory described by ϕ_2. The postcondition states that directory really can be separated into its two subparts: the subdirectory satisfying ϕ_2 can by "allocated" into its own abstract heap cell γ whose corresponding body address is at rooted path P/Q; and the context directory satisfying ϕ_1 still at α. *Abstract dealloaction* is the converse: if we know that γ is at the end of path Q in a directory that is itself at the end of path P, it is safe to combine the two using context application.

We justify the abstract allocation and deallocation axioms by refering to the collapse relation in Definition 5. Abstract allocation is the assertion equivalent of "expanding" by one step, in that the result introduces one additional abstract address, but still collapses to the same complete heap. Deallocation is the equivalent of a single collapse step, and will still result in the same complete file system. Therefore, whilst abstract (de)allocation changes the abstract addressing in use by a file system, it does not change the underlying file system.

3.5 Axiomatic Specification

Figure 5 provides the axioms for the commands used in our software installer example, plus the axioms for `rename` as it is the most challenging command. The rest are IO commands for regular files and their axioms are given in the appendix. Each axiom must be *stable* with respect to both abstract addresses and path promises. Axioms cannot introduce or remove abstract addresses, and must not invalidate any path promises that have been issued. Commands that alter paths (for example, `rename`) ensure this later point by requiring pre-conditions contain no abstract body addresses.

Consider the `mkdir(path)` command. According to its POSIX description, it creates a new empty directory identified by the `path` argument. An existing entry with the same name must not already exist. In our specification's precondition, the `path` argument evaluates to a path of the form P/B/A, and the abstract heap cell $\alpha^{\mathrm{P}} \mapsto \mathrm{B}[\mathrm{C} \wedge \mathit{can\_create}(\mathrm{A})]$ states that the directory B must be at abstract address α found at the end of path P with contents C where the predicate $\mathit{can\_create}(\mathrm{A})$ (definition 10) states that it is safe to create a new entry A. Indeed, in the postcondition, a new empty directory named A is created. Note that in the case we create a new directory directly under the root, in the path expression P will be an empty path and B will be $\top$.

Next, consider `link(existing, new)`, which creates a new hard link with path `new` to the file identified by the path `existing`. Its first axiom is similar to that of `mkdir` except that it involves two paths with one abstract heap cell for each path. The α heap cell simply states that the `existing` path P/A identifies a file link named A to the file with inode address I. The β heap cell states as in `mkdir` that an entry with the name we want to create does not already exist. In the postcondition this new entry is created with another file link to the same file. Note that, in the first `link` axiom, the update takes place between two different directories, while in the second they take place within the same directory.

The `rename` command moves and/or renames the entry identified by the path `old` to that identified by `new`. Consider the first axiom case, where `old`

$$\left\{\begin{array}{c}\texttt{path} \Mapsto \mathrm{P/B/A} \wedge \texttt{r} \Rightarrow - \star \mathcal{E} \star \\ \alpha^{\mathrm{P}} \mapsto \mathrm{B}[\mathrm{C} \wedge \mathit{can\_create}(\mathrm{A})]\end{array}\right\}$$
`r := mkdir(path)`
$$\left\{\texttt{r} \Rightarrow 0 \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \mathrm{B}[\mathrm{C} + \mathrm{A}[\varnothing]]\right\}$$

$$\left\{\begin{array}{c}\texttt{path} \Mapsto \mathrm{P/A} \wedge \texttt{r} \Rightarrow - \star \\ \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \mathrm{A}[\varnothing]\end{array}\right\}$$
`r := rmdir(path)`
$$\left\{\texttt{r} \Rightarrow 0 \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \varnothing\right\}$$

$$\left\{\begin{array}{c}\texttt{existing} \Mapsto \mathrm{P/A} \wedge \texttt{new} \Mapsto \mathrm{P'/D/B} \wedge \texttt{r} \Rightarrow - \star \mathcal{E} \star \\ \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I} \star \ \beta^{\mathrm{P}'} \mapsto \mathrm{D}[\mathrm{C} \wedge \mathit{can\_create}(B)]\end{array}\right\}$$
`r := link(existing, new)`
$$\left\{\texttt{r} \Rightarrow 0 \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I} \star \beta^{\mathrm{P}'} \mapsto \mathrm{D}[\mathrm{C} + \mathrm{B} : \mathrm{I}]\right\}$$

$$\left\{\begin{array}{c}\texttt{existing} \Mapsto \mathrm{P/D/A} \wedge \texttt{new} \Mapsto \mathrm{P/D/B} \wedge \texttt{r} \Rightarrow - \star \mathcal{E} \star \\ \alpha^{\mathrm{P}} \mapsto \mathrm{D}[(\mathrm{C} + \mathrm{A} : \mathrm{I}) \wedge \mathit{can\_create}(\mathrm{B})]\end{array}\right\}$$
`r := link(existing, new)`
$$\left\{\texttt{r} \Rightarrow 0 \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \mathrm{D}[\mathrm{C} + \mathrm{A} : \mathrm{I} + \mathrm{B} : \mathrm{I}]\right\}$$

$$\left\{\begin{array}{c}\texttt{path} \Mapsto \mathrm{P/A} \wedge \texttt{r} \Rightarrow - \star \\ \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I}\end{array}\right\}$$
`r := unlink(path)`
$$\left\{\texttt{r} \Rightarrow 0 \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \varnothing\right\}$$

$$\left\{\begin{array}{c}\texttt{old} \Mapsto \mathrm{P/A} \wedge \texttt{new} \Mapsto \mathrm{P'/D/B} \wedge \texttt{r} \Rightarrow - \star \mathcal{E} \star \\ \alpha^{\mathrm{P}} \mapsto \mathrm{A}[\mathrm{C} \wedge \mathit{complete}] \star \\ \beta^{\mathrm{P}'} \mapsto \mathrm{D}\left[\mathrm{C}' \wedge \mathit{can\_create}(\mathrm{B})\right]\end{array}\right\}$$
`r := rename(old, new)`
$$\left\{\texttt{r} \Rightarrow 0 \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \varnothing \star \beta^{\mathrm{P}'} \mapsto \mathrm{D}[\mathrm{C}' + \mathrm{B}[\mathrm{C}]]\right\}$$

$$\left\{\begin{array}{c}\texttt{old} \Mapsto \mathrm{P/D/A} \wedge \texttt{new} \Mapsto \mathrm{P/D/B} \wedge \texttt{r} \Rightarrow - \star \mathcal{E} \star \\ \alpha^{\mathrm{P}} \mapsto \mathrm{D}[(\mathrm{C} + \mathrm{A}[\mathrm{C}' \wedge \mathit{complete}]) \wedge \mathit{can\_create}(B)]\end{array}\right\}$$
`r := rename(old, new)`
$$\left\{\texttt{r} \Rightarrow 0 \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \mathrm{D}[\mathrm{C} + \mathrm{B}[\mathrm{C}']]\right\}$$

$$\left\{\begin{array}{c}\texttt{old} \Mapsto \mathrm{P/A} \wedge \texttt{new} \Mapsto \mathrm{P'/B} \wedge \texttt{r} \Rightarrow - \star \mathcal{E} \\ \star \\ \alpha^{\mathrm{P}} \mapsto \mathrm{A}[\mathrm{C} \wedge \mathit{complete}] \star \beta^{\mathrm{P}'} \mapsto \mathrm{B}[\varnothing]\end{array}\right\}$$
`r := rename(old, new)`
$$\left\{\texttt{r} \Rightarrow 0 \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \varnothing \star \beta^{\mathrm{P}'} \mapsto \mathrm{B}[\mathrm{C}]\right\}$$

$$\left\{\begin{array}{c}\texttt{old} \Mapsto \mathrm{P/A} \wedge \texttt{new} \Mapsto \mathrm{P'/B} \wedge \texttt{r} \Rightarrow - \star \mathcal{E} \star \\ \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I} \star \beta^{\mathrm{P}'} \mapsto \mathrm{B} : \mathrm{I}'\end{array}\right\}$$
`r := rename(old, new)`
$$\left\{\texttt{r} \Rightarrow 0 \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \varnothing \star \beta^{\mathrm{P}'} \mapsto \mathrm{B} : \mathrm{I}\right\}$$

$$\left\{\begin{array}{c}\texttt{old} \Mapsto \mathrm{P/D/A} \wedge \texttt{new} \Mapsto \mathrm{P/D/B} \wedge \texttt{r} \Rightarrow - \star \mathcal{E} \star \\ \alpha^{\mathrm{P}} \mapsto \mathrm{D}[(\mathrm{C} + \mathrm{A} : \mathrm{I}) \wedge \mathit{can\_create}(\mathrm{B})]\end{array}\right\}$$
`r := rename(old, new)`
$$\left\{\texttt{r} \Rightarrow 0 \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \mathrm{D}[\mathrm{C} + \mathrm{B} : \mathrm{I}]\right\}$$

$$\left\{\begin{array}{c}\texttt{old} \Mapsto \mathrm{P/A} \wedge \texttt{new} \Mapsto \mathrm{P'/D/B} \wedge \texttt{r} \Rightarrow - \star \mathcal{E} \star \\ \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I} \star \beta^{\mathrm{P}'} \mapsto \mathrm{D}\left[\mathrm{C} \wedge \mathit{can\_create}(\mathrm{B})\right]\end{array}\right\}$$
`r := rename(old, new)`
$$\left\{\texttt{r} \Rightarrow 0 \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \varnothing \star \beta^{\mathrm{P}'} \mapsto \mathrm{D}[\mathrm{C} + \mathrm{B} : \mathrm{I}]\right\}$$

$$\left\{\begin{array}{c}\texttt{old} \Mapsto \mathrm{P/A} \wedge \texttt{new} \Mapsto \mathrm{P/A} \wedge \texttt{r} \Rightarrow - \star \mathcal{E} \star \\ \alpha^{\mathrm{P}} \mapsto \mathrm{C} \wedge \mathit{entry}(\mathrm{A})\end{array}\right\}$$
`r := rename(old, new)`
$$\left\{\texttt{r} \Rightarrow 0 \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \mathrm{C}\right\}$$

$$\left\{\begin{array}{c}\texttt{path} \Mapsto \mathrm{P/A} \wedge \texttt{t} \Rightarrow - \star \mathcal{E} \star \\ \alpha^{\mathrm{P}} \mapsto \mathrm{A}[\beta]\end{array}\right\}$$
`t := stat(path)`
$$\left\{\texttt{t} \Rightarrow \mathrm{D} \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \mathrm{A}[\beta]\right\}$$

$$\left\{\begin{array}{c}\texttt{path} \Mapsto \mathrm{P/A} \wedge \texttt{t} \Rightarrow - \star \\ \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I}\end{array}\right\}$$
`t := stat(path)`
$$\left\{\texttt{t} \Rightarrow \mathrm{F} \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I}\right\}$$

$$\left\{\begin{array}{c}\texttt{path} \Mapsto \mathrm{P/D} \star \texttt{dir} \Rightarrow - \star \mathcal{E} \star \\ \alpha^{\mathrm{P}} \mapsto \mathrm{D}[\mathit{top}(\mathrm{C})]\end{array}\right\}$$
`dir := opendir(path)`
$$\left\{\exists \mathrm{H}. \left(\begin{array}{c}\texttt{dir} \Rightarrow \mathrm{H} \star \mathcal{E} \star \\ \alpha^{\mathrm{P}} \mapsto \mathrm{D}[\mathrm{C} \wedge \mathit{names}(\mathrm{A})] \star \mathrm{H} \overset{\mathrm{DS}}{\mapsto} \mathrm{A}\end{array}\right)\right\}$$

$$\left\{\begin{array}{c}\texttt{dir} \Rightarrow \mathrm{H} \star \texttt{fn} \Rightarrow - \star \mathcal{E} \star \\ \mathrm{H} \overset{\mathrm{DS}}{\mapsto} \mathrm{A} \wedge \mathrm{A} \neq \{\}\end{array}\right\}$$
`fn := readdir(dir)`
$$\left\{\begin{array}{c}\texttt{fn} \Rightarrow \mathrm{B} \star \texttt{dir} \Rightarrow \mathrm{H} \star \mathcal{E} \star \\ \mathrm{H} \overset{\mathrm{DS}}{\mapsto} (\mathrm{A} \setminus \{\ \mathrm{B}\}) \wedge \mathrm{B} \in \mathrm{A}\end{array}\right\}$$

$$\left\{\begin{array}{c}\texttt{dir} \Rightarrow \mathrm{H} \star \texttt{fn} \Rightarrow - \star \mathcal{E} \star \\ \mathrm{H} \overset{\mathrm{DS}}{\mapsto} \{\}\end{array}\right\}$$
`fn := readdir(dir)`
$$\left\{\texttt{fn} \Rightarrow \epsilon \star \texttt{dir} \Rightarrow \mathrm{H} \star \mathcal{E} \star \mathrm{H} \overset{\mathrm{DS}}{\mapsto} \{\}\right\}$$

$$\left\{\texttt{dir} \Rightarrow \mathrm{H} \star \mathrm{H} \overset{\mathrm{DS}}{\mapsto} \mathrm{A}\right\}$$
`closedir(dir)`
$$\left\{\mathrm{emp}\right\}$$

Fig. 5. Some of the POSIX axiomatic specifications.

is a directory and `new` does not exist. This captured by the two abstract heap cells α and β in the precondition. Note that in the α heap cell, we require the directory A to be complete: we capture the entire subtree. This means that the target of the operation is not the same as or a descendant of this directory. In the postcondition, the entire contents of the directory, captured by C have been moved to the target location. As in `link` we distinguish the cases of different and same directory updates with distinct axioms.

Finally, the `dir:=opendir(path)` command allocates a new directory stream for the directory identified by `path` and assigns its address to `dir`. In the precondition we require the contents of the directory to be complete at the top level with the $top(\mathrm{C})$ predicate (definition 10). In the allocated directory stream, $\mathrm{H} \overset{\mathrm{DS}}{\mapsto} \mathrm{A}$, the set of the directory's entry names A is given by the $names(\mathrm{A})$ predicate. Elements of the directory stream are obtained via the `readdir` command, for which we have two cases: one when the directory stream is not empty and one where it is. Note that `readdir` non-deterministically selects which entry name to return and remove from the A set. This mirrors the fact that in POSIX the order of directory entries is implementation defined. The `closedir` command simply deallocates the directory stream given as argument.

Definition 12 (Hoare Logic). *The Hoare judgements* $\vdash \{P\}\,\mathbb{C}\,\{Q\}$*, are constructed by the POSIX command axioms and the standard inference rules of separation logic (including the frame rule)*[7]*.*

Soundness We believe it is enough to justify our axiomatic specification by comparing it with the POSIX English standard, since the descriptions are naturally close. However, this is perhaps a controversial point. In the appendix, we give a theoretical soundness result, providing an operational semantics and proving soundness in the style of the views framework [6]. More interestingly, with Tom Ridge, we are currently exploring a more practical soundness result. Ridge has an ML implementation of the POSIX file system. Together, we are working on a mechansied HOL soundness result with respect to his imlementation, plus testing and comparision of his implementation with a real-world implementation.

4 Software Installer

We now demonstrate our reasoning by considering a *software installer*. Newly obtained software is typically provided as a bundle, either downloaded onto the users file system or provided on some media containing a file system. The goal is to take this bundle and place the contents at the correct points in the users' file system such that the software can run. This may involve other tasks such as removing any previous installations and dealing with incompatible user files. Software installers are a common class of client file system programs that perform complex manipulatation of file system structure.

[7] The angelic behaviour of abstract allocation means we do not have the rule of conjunction. This is typical of angelic behaviours, as is well-known from concurrency theory.

Here, we develop an installer for the fictional software "Widget Version 2". It supersedes "Widget Version 1", but is incompatible with any V1 user configuration files. Widget V2 consists of a program executable, '*widgProg*' and a data file, '*widgData*'. We follow common conventions for placing these files in the file system [1]. The two files will be placed in the directory '$\top$/*opt*/*widget*/'. The program will be made usable by creating a hard link from '$\top$/*usr*/*bin*/*widget*' to the file '$\top$/*opt*/*widget*/*widgProg*'. An example situation the installer may encounter is that in figure 1, where Widget V1 exists and the user '*adw*07' has a configuration file.

Even though our installer is fictional, it follows a common workflow found in real practice. In our example, this workflow translates to the following steps:

① Test if entries already exist at the locations we wish to place Widget V2 files. If they exist, we expect '$\top$/*usr*/*bin*/*widget*' to be a file and '$\top$/*opt*/*widget*' to be a directory. If this is the case, we remove them. If it is not, the installer aborts without modifying the system to avoid damaging other components.
② Check for V1 configuration files in home directories, and remove them where they exist as they are assumed to be incompatible.
③ Copy Widget V2 files to the target location on the file system.
④ Make a link to the Widget V2 executable, so the user can run it.

Before implementing the installer we need to consider errors. So far, our specifications describe only when commands succeed. However, commands can also fail with an error result. Our installer relies on the `stat` command returning an error when the path does not exist. We discuss error specifications for all of the commands in the appendix. Here, we discuss only the ENOENT error for `stat`, triggered when a path argument does not resolve to a file or directory. To describe a file system tree in which a path cannot resolve, we define the following:

$$\text{ENOENT}(\mathrm{P}) \triangleq \mathrm{P} \equiv \epsilon \vee (\exists \mathrm{P}', \mathrm{A}, \mathrm{B}, \mathrm{P}''.\ \mathrm{P} \equiv \mathrm{P}'/\mathrm{A}/\mathrm{B}/\mathrm{P}''.\ \alpha^{\mathrm{P}'} \mapsto \mathrm{A}[\mathit{can\_create}(\mathrm{B})])$$

This predicate states that the path P has a prefix which can be resolved, but a suffix which *cannot*. All paths which do not resolve will satisfy this specification and with it, we can give the following error axiom for `stat`:

$$\{\, \texttt{path} \Mapsto \mathrm{P} \wedge \texttt{t} \Longrightarrow - \star \texttt{errno} \Longrightarrow - \star \mathcal{E} \star \mathrm{S} \wedge \text{ENOENT}(\mathrm{P}) \,\}$$
$$\texttt{t := stat(path)}$$
$$\{\, \texttt{t} \Longrightarrow -1 \star \texttt{errno} \Longrightarrow \text{ENOENT} \star \mathcal{E} \star \mathrm{S} \,\}$$

In the precondition we use the predicate on the value of `path` to assert that we are in the error case. Note that we capture the state satisfying the predicate in the logical variable S. In the postcondition, this state is preserved and the global variable `errno` is assigned the error value, for which we use the same name as the predicate for convenience.

To remove an existing Widget installation (point ②) we need to be able to remove non-empty directories. The command `rmdir` only removes empty directories. To remove a non-empty directory we can implement a program `rmdirRec` that recursively removes all of a directories entries before removing the directory itself. The specification is:

$$\left\{\texttt{path} \Mapsto \mathrm{P}/\mathrm{A} \wedge \texttt{r} \Rightarrow - \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \mathrm{A}[\mathit{complete}]\right\}$$
```
r := rmdirRec(path)
```
$$\left\{\texttt{r} \Rightarrow 0 \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \varnothing\right\}$$

Finally, to copy files (point ④), we can implement the program `fileCopy` with the following specification:

$$\left\{\begin{array}{c}\texttt{source} \Mapsto \mathrm{P}/\mathrm{A} \wedge \texttt{target} \Mapsto \mathrm{P}'/\mathrm{D} \wedge \texttt{r} \Rightarrow - \star \mathcal{E} \\ \star\ \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I} \star \beta^{\mathrm{P}'} \mapsto \mathrm{D}[\mathrm{C} \wedge \mathit{can\_create}(\mathrm{A})] \star \mathrm{I} \overset{\mathrm{F}}{\mapsto} \textsc{Sd}\end{array}\right\}$$
```
r := fileCopy(source, target)
```
$$\left\{\exists \mathrm{I}'.\ \texttt{r} \Rightarrow 0 \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I} \star \beta^{\mathrm{P}'} \mapsto \mathrm{D}[\mathrm{C} + \mathrm{A} : \mathrm{I}'] \star \mathrm{I} \overset{\mathrm{F}}{\mapsto} \textsc{Sd} \star \mathrm{I}' \overset{\mathrm{F}}{\mapsto} \textsc{Sd}\right\}$$

We have implemented both `rmdirRec` and `fileCopy` and derived their specifications in the appendix.

The installation of a simple, two file program is a surprisingly complex task. We therefore specify our intuitions about good behaviour and prove that our installer matches them. First, we develop abstractions to assist us in the specifications. We use the following predicates to assert that entries may or may not exist within a given directory resource:

$$\mathit{out}(\mathrm{C}, \mathrm{A}) \triangleq \mathit{top}(\mathrm{C}) \wedge \neg \oplus \mathit{entry}(\mathrm{A}) \qquad \mathit{in}(\mathrm{C}, \mathrm{A}) \triangleq \mathrm{C} + \mathit{entry}(\mathrm{A})$$
$$\mathit{infile}(\mathrm{C}, \mathrm{A}) \triangleq \mathrm{C} + \exists \mathrm{I}.\ \mathrm{A} : \mathrm{I}$$

The first predicate describes directory entries C in which an entry named A does not exist, whereas the second describes entries C in which it does. $\mathit{infile}(\mathrm{C}, \mathrm{A})$ is more specific and describes directory entries C with an A file entry.

We build a precondition for our installer out of several sub-assertions with the help of the above predicates. In these, the assertion $+_{\mathrm{X} \epsilon \mathrm{E}}\, \phi$ is the iterated version of +, interpreted as $\phi_1 + \cdots + \phi_{|\mathrm{E}|}$ where each ϕ_i has X bound to a distinct member of E.

$$\mathit{instSrc}_{\mathrm{Pre}} \triangleq \delta^{\mathrm{IL}} \mapsto (\mathrm{widgProg} : \mathrm{J} + \mathrm{widgData} : \mathrm{K}) \star \mathrm{J} \overset{\mathrm{F}}{\mapsto} \textsc{Prog} \star \mathrm{K} \overset{\mathrm{F}}{\mapsto} \textsc{Dat}$$
$$\mathit{home}_{\mathrm{Pre}} \triangleq \alpha^{\top} \mapsto \mathrm{home}\left[+_{(\mathrm{N},\mathrm{C}) \epsilon \mathrm{H}}\, \mathrm{N}\left[\mathit{infile}(\mathrm{C}, .\mathrm{wconf}) \vee \mathit{out}(\mathrm{C}, .\mathrm{wconf})\right]\right]$$
$$\mathit{bin}_{\mathrm{Pre}} \triangleq \gamma^{\top/\mathit{usr}} \mapsto \mathrm{bin}\left[\mathit{in}(\mathrm{B}, \mathrm{widget}) \vee \mathit{out}(\mathrm{B}, \mathrm{widget})\right]$$
$$\mathit{opt}_{\mathrm{Pre}} \triangleq \beta^{\top} \mapsto \mathrm{opt}\left[\mathit{top}(\mathrm{T}) + (\varnothing \vee \mathrm{widget}[\mathrm{T}_w \wedge \mathit{complete}]\right]$$

Each of these describes the states that parts of the file system may be in for the Widget installer to run safely. The directory entries and file data that make up the Widget version 2 installation sources are described by $\mathit{instSrc}_{\mathrm{Pre}}$. We require them to be in a location given by the variable `iloc`. $\mathit{home}_{\mathrm{Pre}}$ captures all the home directories of the system, along with the fact that some of them will contain a Widget V1 '*.wconf*' configuration file. The $\mathit{bin}_{\mathrm{Pre}}$ resource captures the UNIX executables directory, that may contain a '*widget*' entry. Finally, $\mathit{opt}_{\mathrm{Pre}}$ describes the target installation directory, which may already contain a previous installation, which we require to be complete, as it will be deleted.

We combine these descriptions into a precondition, where we also snapshot the initial state in the logical variable W, to show that nothing changes in the event of an error.

$$P_{ins} \triangleq \texttt{iloc} \Rightarrow \mathrm{IL} \star \texttt{r} \Rightarrow - \star \texttt{errno} \Rightarrow - \wedge \mathrm{W} \wedge \mathit{instSrc}_{\mathrm{Pre}} \star \mathit{home}_{\mathrm{Pre}} \star \mathit{bin}_{\mathrm{Pre}} \star \mathit{opt}_{\mathrm{Pre}}$$

If the installer errors, we expect the file system to be unchanged. If it succeeds, we expect Widget V1 to be installed successfully. There should be no other outcome. We describe a successful installation with the following sub-assertions:

$$
\begin{aligned}
v2Files_{\mathrm{Post}}(\mathrm{J},\mathrm{K}) &\triangleq \mathrm{J} \overset{\mathrm{F}}{\mapsto} \textsc{Prog} \star \mathrm{K} \overset{\mathrm{F}}{\mapsto} \textsc{Dat} \\
home_{\mathrm{Post}} &\triangleq \alpha^{\top} \mapsto \mathrm{home}\left[+_{(\mathrm{N},\mathrm{C})\in \mathrm{H}}\, \mathrm{N}\,[\mathrm{C} \wedge \mathit{can\_create}(.\mathrm{wconf})]\right] \\
bin_{\mathrm{Post}}(\mathrm{J}) &\triangleq \gamma^{\top/usr} \mapsto \mathrm{bin}[\mathrm{B} + \mathrm{widget} : \mathrm{J}] \\
opt_{\mathrm{Post}}(\mathrm{J},\mathrm{K}) &\triangleq \beta^{\top} \mapsto \mathrm{opt}[\mathrm{T} + \mathrm{widget}[\mathrm{widgProg} : \mathrm{J} + \mathrm{widgDat} : \mathrm{K}]]
\end{aligned}
$$

The postcondition is built from these sub-assertions.

$$
Q_{ins} \triangleq \begin{array}{c} \exists \mathrm{R}, \mathrm{J}', \mathrm{K}'.\ \mathtt{iloc} \Rightarrow \textsc{Il} \star \mathtt{r} \Rightarrow \mathrm{R} \star \mathtt{errno} \Rightarrow - \wedge (\mathrm{R} = -1 \Rightarrow \mathrm{W}) \\ \wedge\ \mathrm{R} = 0 \Rightarrow \left(\begin{array}{c} \mathit{instSrc}_{\mathrm{Pre}} \star \mathit{home}_{\mathrm{Post}} \star \mathit{opt}_{\mathrm{Post}}(\mathrm{J}', \mathrm{K}') \star \mathit{bin}_{\mathrm{Post}}(\mathrm{J}') \\ \star\ v2\mathit{Files}_{\mathrm{Post}}(\mathrm{J}', \mathrm{K}') \end{array} \right) \end{array}
$$

Note that if the installer fails, the return variable `r` has value -1 and the state of the file system is the same as in the precondition, captured by the logical variable W. Otherwise, `r` is 0 and the state changes according to the sub-assertions that we have defined.

Our installer implementation, along with a proof that it meets its specification, $\{P_{ins}\}$`installWidgetV2`$\{Q_{ins}\}$, is given in figure 6. Throughout the proof we make implicit use of the frame rule to temporarily discard irrelevant state, and at the points of axiom application we implicitly use abstract allocation/deallocation as described in section 2.

5 Conclusions and Future Work

The POSIX file system provides an interesting challenge for local reasoning: complex abstract data update with global rooted paths for identifying the place to do local update. We have extended structural separation logic with *path promises*, to provide a natural axiomatic specification of the POSIX file system; the general theory is in [23]. We are able to verify properties of client programs such as the software installer, demonstrating integrated reasoning for the file system and the heap. Our POSIX reasoning provides an illustrative example of this combination of abstract global and local reasoning; others include reasoning about the ith element of a list [23] and the combination of JQuery and DOM.

The promises in our POSIX reasoning are naturally stable. Wright has also explored the combination of promises and *obligations*: promises on abstract heap cells give information about what can be relied upon by the environment; obligations gives information about what data fragments must guarantee; sometimes both are needed for stability. In this paper, the only obligations are that the abstract cell and body addresses must be preserved. In general, understanding obligations is hard. A natural test example would be to extend the core POSIX fragment presented here with non-linear paths (.. and symbollic links), where the paths can move back and forth over the structure. We also believe obligations will be useful for file-access permissions and shared-memory concurrency.

$\{ P_{ins} \}$

```
r := installWidgetV2 ≜
local t1, t2, hDir, user {
```

$\{ \mathtt{t1} \Rightarrow - \star \mathtt{t2} \Rightarrow - \star \mathtt{errno} \Rightarrow - \star bin_{\mathrm{Pre}} \star opt_{\mathrm{Pre}} \}$

```
  // Check for preexisting files (point ①). The installer expects T/usr/bin/widget
  // to be a file and T/opt/widget to be a directory, if they exist.
  t1 := stat('T/usr/bin/widget');  t2 := stat('T/opt/widget');
```

$$\left\{ \begin{array}{c} \mathtt{t1} \Rightarrow \mathrm{T1} \wedge (\mathrm{T1} = \mathrm{F} \vee \mathrm{D} \vee -1) \star \mathtt{t2} \Rightarrow \mathrm{T2} \wedge (\mathrm{T2} = \mathrm{F} \vee \mathrm{D} \vee -1) \\ \star\ \mathtt{errno} \Rightarrow \mathrm{E} \wedge ((\mathrm{T1} = -1 \vee \mathrm{T2} = -1) \Rightarrow \mathrm{E} = \textsc{enoent}) \star bin_{\mathrm{Pre}} \star opt_{\mathrm{Pre}} \end{array} \right\}$$

```
  if t1 = D ∨ t2 = F
    // There are preexisting entries, but not of a previous installation.
    // The installer ends here without any modifications.
    r = -1;
  else
    // Either previous entries do not exist, or they are of a previous installation.
```

$\{ \mathtt{r} \Rightarrow - \star \mathtt{t1} \Rightarrow \mathrm{F} \vee \mathtt{t1} \Rightarrow -1 \star \mathtt{t2} \Rightarrow \mathrm{D} \vee \mathtt{t2} \Rightarrow -1 \star bin_{\mathrm{Pre}} \star opt_{\mathrm{Pre}} \}$

```
    if t1 = F
      // Remove previous installation executable.
      r := unlink('T/usr/bin/widget');
    if t2 = D
      // Remove previous installation directory. We apply the rmdirRec specification.
```

$\{ \mathtt{t2} \Rightarrow \mathrm{D} \star \mathtt{r} \Rightarrow - \star \beta^{\mathsf{T}} \mapsto \mathrm{opt}[\mathrm{T} + \mathrm{widget}[\mathrm{T}_w \wedge complete]] \}$

```
      r := rmdirRec('T/opt/widget');
```

$\{ \mathtt{t2} \Rightarrow \mathrm{D} \star \mathtt{r} \Rightarrow 0 \star \beta^{\mathsf{T}} \mapsto \mathrm{opt}[\mathrm{T}] \}$

$\{ \mathtt{r} \Rightarrow 0 \star \mathtt{t1} \Rightarrow \mathrm{F} \vee \mathtt{t1} \Rightarrow -1 \star \mathtt{t2} \Rightarrow \mathrm{D} \vee \mathtt{t2} \Rightarrow -1 \star \gamma^{\mathsf{T}/usr} \mapsto \mathrm{bin}[\mathrm{B}] \star \beta^{\mathsf{T}} \mapsto \mathrm{opt}[\mathrm{T}] \}$

```
    // Remove any stale Widget configuration files (point ②)
```

$\{ \mathtt{hDir} \Rightarrow - \star \mathtt{user} \Rightarrow - \star \alpha^{\mathsf{T}} \mapsto \mathrm{home}\left[+_{(\mathrm{N},\mathrm{C}) \in \mathrm{H}} \mathrm{N}\left[infile(\mathrm{C}, .\mathrm{wconf}) \vee out(\mathrm{C}, .\mathrm{wconf}) \right]\right] \}$

```
    hDir := opendir('T/home');
    user := readdir(hDir);
```

$$\left\{ \begin{array}{c} \exists \mathrm{H_D}, \mathrm{U}, \mathrm{U_S}.\ \mathtt{hDir} \Rightarrow \mathrm{H_D} \star \mathtt{user} \Rightarrow \mathrm{U} \star \mathrm{H_D} \overset{\mathrm{DS}}{\mapsto} \mathrm{U_S} \\ \star\ \alpha^{\mathsf{T}} \mapsto \mathrm{home}\left[+_{(\mathrm{N},\mathrm{C}) \in \mathrm{H}} \mathrm{N}\left[out(\mathrm{C}, .\mathrm{wconf}) + \mathrm{N} \in \mathrm{U_S} \Rightarrow (\varnothing \vee \exists \mathrm{I}.\ .\mathrm{wconf} : \mathrm{I}) \wedge \mathrm{N} \notin \mathrm{U_S} \Rightarrow \varnothing \right]\right] \end{array} \right\}$$

```
  while user ≠ ϵ
    // We iterate over every user's home directory and delete the file.
    // If the file does not exist, then unlink returns -1 as in stat.
    r := unlink('T/home'/user/'.wconf');  user := readdir(hDir);
  closedir(hDir);
  // In the end, there are no Widget V1 configuration files.
```

$\{ \alpha^{\mathsf{T}} \mapsto \mathrm{home}\left[+_{(\mathrm{N},\mathrm{C}) \in \mathrm{H}} \mathrm{N}\left[\mathrm{C} \wedge can\_create(.\mathrm{wconf})\right]\right] \}$

```
  // Now we create the new installation, copy the new Widget files
  // and link the executable (Points ③ and ④)
```

$$\left\{ \begin{array}{c} \mathtt{r} \Rightarrow - \star \mathtt{iloc} \Rightarrow \mathrm{I_L} \star \delta^{\mathrm{IL}} \mapsto v2Dir_{\mathrm{Pre}} \star \alpha^{\mathsf{T}} \mapsto home_{\mathrm{Post}} \\ \star\ \gamma^{\mathsf{T}/usr} \mapsto \mathrm{bin}[\mathrm{B}] \star \beta^{\mathsf{T}} \mapsto \mathrm{opt}[\mathrm{T}] \star \mathrm{J} \overset{\mathrm{F}}{\mapsto} \textsc{Prog} \star \mathrm{K} \overset{\mathrm{F}}{\mapsto} \textsc{Dat} \end{array} \right\}$$

```
  r := mkdir('T/opt/widget');
  r := fileCopy(instLoc/'widgProg', 'T/opt/widget');
  r := fileCopy(instLoc/'widgData', 'T/opt/widget');
  r := link('T/opt/widget/widgProg', 'T/usr/bin/widget');  r := 0
```

$$\left\{ \begin{array}{c} \exists \mathrm{J'}, \mathrm{K'}.\ \mathtt{r} \Rightarrow 0 \star \mathtt{iloc} \Rightarrow \mathrm{I_L} \star instSrc_{\mathrm{Pre}} \star home_{\mathrm{Post}} \\ \star\ \beta^{\mathsf{T}} \mapsto \mathrm{opt}\left[\mathrm{T} + \mathrm{widget}\left[\mathrm{widgProg} : \mathrm{J'} + \mathrm{widgData} : \mathrm{K'}\right]\right] \\ \star\ \gamma^{\mathsf{T}/usr} \mapsto \mathrm{bin}[\mathrm{B} + \mathrm{widget} : \mathrm{J'}] \star \mathrm{J'} \overset{\mathrm{F}}{\mapsto} \textsc{Prog} \star \mathrm{K'} \overset{\mathrm{F}}{\mapsto} \textsc{Dat} \end{array} \right\}$$

```
}
```

$$\left\{ \begin{array}{c} \exists \mathrm{R}, \mathrm{J'}, \mathrm{K'}.\ \mathtt{iloc} \Rightarrow \mathrm{I_L} \star \mathtt{r} \Rightarrow \mathrm{R} \star \mathtt{errno} \Rightarrow - \wedge (\mathrm{R} = -1 \Rightarrow \mathrm{W}) \\ \wedge\ \mathrm{R} = 0 \Rightarrow \left(instSrc_{\mathrm{Pre}} \star home_{\mathrm{Post}} \star opt_{\mathrm{Post}}(\mathrm{J'}, \mathrm{K'}) \star bin_{\mathrm{Post}}(\mathrm{K'}) \star v2Files_{\mathrm{Post}}(\mathrm{J'}, \mathrm{K'}) \right) \end{array} \right\}$$

$\{ Q_{ins} \}$

Fig. 6. Widget V2 software installer.

References

[1] Filesystem Hierarchy Standard Group. Filesystem hierarchy standard.

[2] POSIX.1-2008, IEEE 1003.1-2008, The Open Group Base Specifications Issue 7.

[3] Variables as resource in separation logic. *Electronic Notes in Theoretical Computer Science*, 155:247 – 276, 2006.

[4] K. Arkoudas, K. Zee, V. Kuncak, and M. Rinard. Verifying a File System Implementation. In *LNCS: Formal Methods and Software Engineering*. 2004.

[5] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic and tree update. *SIGPLAN Not.*, 2005.

[6] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: compositional reasoning for concurrent programs. POPL, 2013.

[7] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP*. 2010.

[8] K. Fisher, N. Foster, D. Walker, and K. Q. Zhu. Forest: a language and toolkit for programming with filestores. ICFP, 2011.

[9] L. Freitas, Z. Fu, and J. Woodcock. POSIX file store in Z/Eves: an experiment in the verified software repository. *Engineering of Complex Computer Systems, IEEE International Conference*, 2007.

[10] L. Freitas, J. Woodcock, and A. Butterfield. POSIX and the verification grand challenge: A roadmap. In *ICECCS*, 2008.

[11] P. Gardner, G. Ntzik, and A. Wright. Local Reasoning about File Systems: Technical Report. Technical report, Imperial College London, 2012 `http://www.doc.ic.ac.uk/~gn408/POSIXFS/`.

[12] P. Gardner, A. Raad, M. Wheelhouse, and A. Wright. Abstract Local Reasoning for Concurrent Libraries, Submitted, 2013.

[13] P. Gardner, G. Smith, M. Wheelhouse, and U. Zarfaty. Local Hoare reasoning about DOM. In *PODS*, 2008.

[14] W. H. Hesselink and M. Lali. Formalizing a hierarchical file system. *REFINE*, 2009.

[15] A. Hobor and J. Villard. The ramifications of sharing in data structures. POPL, 2013.

[16] R. Joshi and G. J. Holzmann. A mini challenge: build a verifiable filesystem. *Form. Asp. Comput.*, 2007.

[17] C. Morgan and B. Sufrin. Specification of the UNIX Filing System. *Software Engineering, IEEE Transactions on*, 1984.

[18] G. Ntzik. *Local Reasoning about File Systems*. PhD thesis, 2014 (Expected).

[19] M. Parkinson and G. Bierman. Separation logic and abstraction. POPL, 2005.

[20] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[21] G. Smith. *Local Reasoning about Web Programs*. PhD thesis, 2011.

[22] A. Turon, D. Dreyer, and L. Birkedal. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. ICFP, 2013.

[23] A. Wright. *Structural Separation Logic*. PhD thesis, Imperial College London, 2013.

Appendix

Satisfaction Relation

Definition 13 (Satisfaction Relation). *The satisfaction relations,* $\vDash_P$: LEnv → AStates × Asrts *and* $\vDash_\phi$: LEnv → Dirs × DirAsrts*, for the abstract state and directory assertions of figure 4, parameterised by a logical environment, are defined as:*

$$
\begin{array}{lcl}
e, (fs, \mathrm{ph}, \sigma) \vDash_P \alpha^E \mapsto \phi & \iff & \exists d.\ fs((\!|\alpha|\!)_e) = ((\!|E|\!)_e, d) \wedge e, d \vDash_\phi \phi \\
e, (fs, \mathrm{ph}, \sigma) \vDash_P \mathrm{X} \overset{\mathrm{F}}{\mapsto} E & \iff & fs((\!|\mathrm{X}|\!)_e) = (\!|E|\!)_e \\
e, (fs, \mathrm{ph}, \sigma) \vDash_P \mathrm{X} \overset{\mathrm{OF}}{\mapsto} (I, E) & \iff & \exists f.\ (\!|\mathrm{X}|\!)_e = f \wedge f \in \mathrm{OFAddrs} \wedge \mathrm{ph}(f) = ((\!|I|\!)_e, (\!|E|\!)_e) \\
e, (fs, \mathrm{ph}, \sigma) \vDash_P \mathrm{X} \overset{\mathrm{DS}}{\mapsto} E & \iff & \exists n.\ (\!|\mathrm{X}|\!)_e = n \wedge n \in \mathrm{DSAddrs} \wedge \mathrm{ph}(n) = (\!|E|\!)_e \\
e, (fs, \mathrm{ph}, \sigma) \vDash_P E \overset{\mathrm{H}}{\mapsto} E' & \iff & \exists n.\ (\!|E|\!)_e = n \wedge n \in \mathbb{N}^+ \wedge \mathrm{ph}(n) = (\!|E'|\!)_e \\
e, (fs, \mathrm{ph}, \sigma) \vDash_P \mathtt{var} \Rrightarrow E & \iff & \sigma(\mathtt{var}) = (\!|E|\!)_e \\
e, (fs, \mathrm{ph}, \sigma) \vDash_P \mathrm{Expr} \Mapsto E & \iff & [\![\mathrm{Expr}]\!]_\sigma = (\!|E|\!)_e \\
e, d \vDash_\phi \varnothing & \iff & d = \varnothing \\
e, d \vDash_\phi E_1 : E_2 & \iff & d = (\!|E_1|\!)_e : (\!|E_2|\!)_e \\
e, d \vDash_\phi E[\phi] & \iff & \exists d'.\ d = (\!|E|\!)_e[d'] \wedge e, d' \vDash_\phi \phi \\
e, d \vDash_\phi \top[\phi] & \iff & \exists d'.\ d = \top[d'] \wedge e, d' \vDash_\phi \phi \\
e, d \vDash_\phi E & \iff & d = (\!|E|\!)_e \\
e, d \vDash_\phi \phi_1 + \phi_2 & \iff & \exists d', d''.\ d = d' + d'' \wedge e, d' \vDash_\phi \phi_1 \wedge e, d'' \vDash_\phi \phi_2 \\
e, d \vDash_\phi \phi_1 \circ_\alpha \phi_2 & \iff & \exists d', \mathbf{x}, d''.\ d = d' \circ_{\mathbf{x}} d'' \wedge e[\alpha \mapsto \mathbf{x}], d' \vDash_\phi \phi_1 \wedge e, d'' \vDash_\phi \phi_2 \\
e, d \vDash_\phi @E & \iff & \mathrm{resolve}((\!|E|\!)_e, d)\ \textit{defined}
\end{array}
$$

IO Command Axioms

Here we include axioms for IO commands on regular files in figure 7. These supplement the axioms of figure 5. Together, these provide the specification for commands when they succeed. Error specifications are discussed in the next section.

Errors

So far, in figure 5 we have given specifications of when the POSIX commands succeed. In any other case they fail. In POSIX this generally means that an *error code* is returned to explain the cause of the failure. Programmers should examine this error code, and take remedial action if the command failed. Errors are also sometimes required to perform elementary tasks. For example, to see if a path `p` exists in the file system, one typically performs `stat(p)` and checks if an error is returned. No error indicates that the path resolves.

We model the following errors that are relevant to our POSIX fragment:

1. ENOENT: A component of a path does not resolve to an existing file or the path is an empty string. For example, this error occurs when resolving $\top/a/b/c/d$, but the directory $\top/a/b$ does not exist in the file system.

2. ENOTDIR: A component of the path prefix is not a directory. For example, this error occurs when resolving $\top/a/b/c/d$, but $\top/a/b$ identifies a file rather than a directory.
3. EEXIST: The path resolves to an existing file or directory. For example, the `link(existing, new)` command returns this error as it expects no file to exist at the `new` path.
4. ENOTEMPTY: The path argument names a directory that is not an empty directory. This error occurs when the path identifies a directory that has entries, but the command expected an empty directory. For example, the `rmdir` command returns this error.
5. EPERM: The intended operation is not supported by the POSIX implementation. This error occurs when you attempt to use a command, but the POSIX implementation running the program does not support using the command on the resource identified by the path. For example, as we do not allow hard links to directories, our `link` command returns this error when it is applied to directories.
6. EINVAL: The arguments supplied to the intended operation are of the wrong form. For example, in order to avoid creating cycles, `rename(old, new)` returns this error if `new` is a descendant `old`.

Commands communicate their success or failure via their return value. Most commands return 0 when they succeed — `open` and `opendir` return non integer values when they succeed — and -1 otherwise. If -1 is returned, a constant integer value representing the error is assigned to the global variable `errno`. The types of errors each command can fail with are given in table 1.

Command	**Possible errors**
`r := mkdir(path)`	ENOENT, ENOTDIR, EEXIST
`r := rmdir(path)`	ENOENT, ENOTDIR$_*$, ENOTEMPTY
`r := link(to, from)`	ENOENT$_*$, ENOTDIR, EPERM
	EEXIST (only for the `from` parameter)
`r := unlink(path)`	ENOENT$_*$, ENOTDIR, EPERM
`r := rename(old, new)`	ENOENT$_*$, ENOTDIR$_*$, ENOTEMPTY,
	EISDIR, EINVAL
`t := stat(path)`	ENOENT, ENOTDIR
`fd := open(path, flags)`	ENOENT$_*$, ENOTDIR, EISDIR
`o := lseek(fd, off, wh)`	No errors
`sz := write(fd, str)`	No errors
`str := read(fd, size)`	No errors
`close(fd)`	No errors
`h := opendir(path)`	ENOENT, ENOTDIR
`s := readdir(h)`	No errors
`closedir(h)`	No errors

Table 1. Possible errors that each command can return

However, we still consider certain types of command failure to be program faults. Calling `wrire(fd)` with an `fd` that does not correspond to a valid file descriptor still faults in our specifications, despite POSIX considering this an EBADF error. We make this distinction based on an intuition of *ownership*. File descriptors are owned and managed by the programs that create them. Writing a program that could encounter EBADF is always going to be a programmer mistake. We can use our program logic to prove that a program does not fault: it would never cause EBADF to occur. Error checking for EBADF could then be elided as unnecessary.

On the other hand, the file system is a shared resource, not owned by any one program. Even in our non-concurrent system, there is nothing to stop someone mutating the file system against our expectations before our program runs. Thus, it is sensible and expected that programmers use errors as a form of "defensive programming", to cater for unanticipated external changes to the file system structure. We can use our reasoning to determine if a program detects all possible error cases, and more importantly how it handles them.

For each command, and each error it can produce, we provide a small axiom that describes the states on which the command will produce the error. Each additional small axiom describes all the states that cause a specific error code.

The properties of each error code are independent of its use by any one command. For example, failing to resolve a path correctly is the same whether the command that failed is `open` or `mkdir`. As such, in figure 8 we provide predicates that describe the states on which resolving a given path would result in each error. We name the predicates identically to the error they represent. There are two variants of ENOENT: $\text{ENOENT}_*(\texttt{path})$ is satisfied when `path` or any prefix of `path` do not resolve, and $\text{ENOENT}(\texttt{path})$ is satisfied simply when any prefix of `path` does not resolve. Similarly there are two variants of ENOTDIR.

With these error predicates, we can axiomatize commands that may return errors. Here, we consider the `stat` command:

$$\frac{\{\,\texttt{path} \Mapsto \mathrm{P/A} \wedge \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I}\,\}}{\begin{array}{c}\texttt{t, r := stat(path)}\\ \{\,\texttt{t} \Mapsto F \wedge \texttt{r} \Mapsto 0 \wedge \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I}\,\}\end{array}} \qquad \frac{\{\,\texttt{path} \Mapsto \mathrm{P/A} \wedge \alpha^{\mathrm{P}} \mapsto \mathrm{A}[\beta]\,\}}{\begin{array}{c}\texttt{t, r := stat(path)}\\ \{\,\texttt{t} \Mapsto D \wedge \texttt{r} \Mapsto 0 \wedge \alpha^{\mathrm{P}} \mapsto \mathrm{A}[\beta]\,\}\end{array}}$$

$$\frac{\{\,\mathrm{S} \wedge \text{ENOENT}_*(\texttt{path})\,\}}{\begin{array}{c}\texttt{t, r := stat(path)}\\ \{\,\texttt{r} \Mapsto -1 \wedge \texttt{errno} \Mapsto \text{ENOENT} \wedge \mathrm{S}\,\}\end{array}} \qquad \frac{\{\,\mathrm{S} \wedge \text{ENOTDIR}(\texttt{path})\,\}}{\begin{array}{c}\texttt{t, r := stat(path)}\\ \{\,\texttt{r} \Mapsto -1 \wedge \texttt{errno} \Mapsto \text{ENOTDIR} \wedge \mathrm{S}\,\}\end{array}}$$

Error cases for the rest of the commands of table 1 are defined similarly.

Example Proofs

Here we give the implementations and proofs for the removal of a non-empty directory and file copying.

`rmdirRec`

Recall the intuitive specification of `rmdirRec` from the software installer example.

$$\left\{ \texttt{path} \Mapsto \mathrm{P}/\mathrm{A} \wedge \texttt{r} \Rightarrow - \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \mathrm{A}[\mathit{complete}] \right\}$$
```
r := rmdirRec(path)
```
$$\left\{ \texttt{r} \Rightarrow 0 \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \varnothing \right\}$$

Our implementation of this command and a proof sketch of the specification is given in figure 9. Notice that at each loop iteration, we use abstract allocation to address the entry we are going delete. When we remove, e.g. with `unlink`, we implicitly apply the Frame Rule to isolate only that resource. Specifications of the cases where `rmdirRec` fails with an error can be easily derived.

`fileCopy`

Finally, following is the specification of `fileCopy`.

$$\left\{ \begin{array}{c} \texttt{source} \Mapsto \mathrm{P}/\mathrm{A} \wedge \texttt{target} \Mapsto \mathrm{P}'/\mathrm{D} \wedge \texttt{r} \Rightarrow - \star \mathcal{E} \\ \star\ \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I} \star \beta^{\mathrm{P}'} \mapsto \mathrm{D}[\mathrm{C} \wedge \mathit{can\_create}(\mathrm{A})] \star \mathrm{I} \overset{\mathrm{F}}{\mapsto} \textsc{Sd} \end{array} \right\}$$
```
r := fileCopy(source, target)
```
$$\left\{ \exists \mathrm{I}'.\ \texttt{r} \Rightarrow 0 \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I} \star \beta^{\mathrm{P}'} \mapsto \mathrm{D}[\mathrm{C} + \mathrm{A} : \mathrm{I}'] \star \mathrm{I} \overset{\mathrm{F}}{\mapsto} \textsc{Sd} \star \mathrm{I}' \overset{\mathrm{F}}{\mapsto} \textsc{Sd} \right\}$$

Because file sizes can exceed available process memory, they are copied chunk by chunk. In figure 10 we show an implementation of a chunked file copy command and prove it's correct with respect to the above specification.

Soundness

We reason about programs manipulating abstract program states (definition 8) that contain instrumentation in the form of abstract addresses and path promises. Operationally, commands operate on *machine states* where the instrumentation is erased.

Definition 14 (Machine States). Machine states, $s \in \textsc{States}$, *are those abstract program states of definition 8 where the file system heap is complete.*

$$\textsc{States} \triangleq \{ s \in \textsc{AStates} \mid s\downarrow_1 \ \ \mathit{is\ complete} \}$$

We relate abstract states to machine states via *reification*. Reification accounts for the (possibly) partial file system by using completions. Process heaps are not instrumented, so are reused unchanged.

Definition 15 (Reification). *The* reification *function,* $\lfloor \cdot \rfloor : \textsc{AStates} \to \mathcal{P}(\textsc{States})$, *is defined by:*

$$\lfloor as \rfloor = \{ (fs, as\downarrow_2, as\downarrow_3) \mid fs \in \mathrm{completion}(as\downarrow_1) \}$$

where $\mathrm{completion}(fs) \triangleq \{ fs_c \in \mathrm{FS} \mid fs' \in \mathrm{FS} \wedge fs_c = fs \sqcup fs' \wedge fs_c \ \mathit{is\ complete} \}$ *where* $\downarrow_i$ *denotes the* i*th projection. We lift reification to sets of instrumented states pointwise.*

The operational semantics of our programming language are directly derived from the operational semantics of the Views framework.

Definition 16 (Program Evaluation). *The evaluation relation, $\Downarrow$, relates programs and their initial states to either a terminal state or a distinguished* fault *state $\lightning$, where $\lightning \notin$ Values. Command/state pairs not in the relation are deemed to be divergent.*

$$\Downarrow \subset (\textsc{Comm} \times \textsc{States}) \times (\textsc{States} \uplus \{\lightning\})$$

We derive the evaluation relation from the operational semantics framework in Views.

We use the standard partial correctness fault avoiding interpretation of Hoare triples.

Definition 17 (Hoare Triple Interpretation).

$$\vDash \{P\}\,\mathbb{C}\,\{Q\} \iff \begin{array}{c} \forall e \in \textsc{LEnv}, s \in \lfloor [\![P]\!]_e \rfloor .\ \mathbb{C}, s \not\Downarrow \lightning \wedge \\ (\exists s' . \mathbb{C}, s \Downarrow s' \Rightarrow s' \in \lfloor [\![Q]\!]_e \rfloor) \end{array}$$

Theorem 1 (Soundness). *If $\vdash \{P\}\,\mathbb{C}\,\{Q\}$, then $\vDash \{P\}\,\mathbb{C}\,\{Q\}$.*

We prove theorem 1 via the *Views* framework of [6]. Readers interested in the details of Views are refered to [6], which gives a detailed account of the parameters and properties required to prove the soundness theorem. Here we simply instantiate the parameters of the Views framework and prove the required properties.

Definition 18 (File Systems Atomic Commands). *The atomic commands are those of definition 9.*

Definition 19 (File Systems View Monoid). *The File Systems View Monoid is:*

$$\textsc{FSView} \triangleq (\mathcal{P}(\textsc{AStates}), \circ, u)$$

where

1. $(ifs_1, \mathrm{ph}_1, \sigma_1) \circ' (ifs_2, \mathrm{ph}_2, \sigma_2) = (ifs_1 \sqcup ifs_2, \mathrm{ph}_1 \sqcup \mathrm{ph}_2, \sigma_1 \sqcup \sigma_2)$
2. $\circ$ *is the lift of* $\circ'$ *to sets*
3. $u = \{(\varnothing, \varnothing, \varnothing)\}$.

Definition 20 (File Systems Axiomatisation). *The File System Axiomatisation is given in figures 5 and 7.*

Definition 21 (File System Machine States). *The machine states are the program states of definition 14.*

Definition 22 (File System Interpretation of Atomic Commands). *The File System interpretation of Atomic Commands are given in figures 11 and 12.*

Definition 23 (File Systems View Reification). *The File System View Reification is the lift of the Instrumented State Reification of definition 15 to sets.*

Definition 24 (File Systems View Disjunction). *The File Systems View Disjunction function is set union, $\cup$.*

We now prove each of the required properties. We begin with some additional lemmas. The Views framework assumes that the $*$ of Views is the separating conjunction. As we work with logical environments, we have "baked" * into the assertion language. Assertions are then transformed into views by interpretation through a logical environment. However, the $*$ of our assertion language is essentially identical to the $*$ of our Views monoid.

Lemma 1 ($*$ equivalence). *Let $*$ be the separating conjunction of our assertion language, figure . Recall that the separating conjunction of our views monoid is $\circ$. For all $p, q \in \mathcal{P}(\text{ASTATES}), e \in \text{LENV}$, it is the case that*

$$[\![p * q]\!]_e = [\![p]\!]_e \circ [\![q]\!]_e$$

Proof. Calculating using definitions 13 and 19

$$\begin{aligned}
[\![p * q]\!]_e &= \{as \in \text{ASTATES} \mid e, as \vDash p * q\} \\
&= \{(ifs_1 \bullet ifs_2, \text{ph}_1 \bullet \text{ph}_2, \sigma) \mid ifs_1, \text{ph}_1, \sigma \vDash p, ifs_2, \text{ph}_2, \sigma, \vDash q\} \\
&= \{(ifs_1, \text{ph}_1, \sigma) \circ' (ifs_2, \text{ph}_2, \sigma) \mid ifs_1, \text{ph}_1, \sigma \vDash p, ifs_2, \text{ph}_2, \sigma, \vDash q\} \\
&= \{P \circ' Q \mid P \in [\![p]\!]_e, Q \in [\![q]\!]_e\} \\
&= [\![p]\!]_e \circ [\![q]\!]_e
\end{aligned}$$

Lemma 2 (Atomic Soundness of Commands).

Proof. We will cover only characteristic examples of commands in our fragment. Atomic soundness for the remaining command axioms is proven similarly.

First, consider the `mkdir` *command axiom in figure 5. By the satisfaction relation (definition 13) we have:*

$$\begin{array}{c}
[\![\, \texttt{path} \Rightarrow \text{P/B/A} \wedge \alpha^{\text{P}} \mapsto \text{B}[\text{C} \wedge \neg top_entry(\text{A}) \wedge \neg top_address] \star r \,]\!]_e \\
= \\
\left\{ \begin{array}{c}
(ifs, \text{ph}, \sigma) \in \text{ASTATES} \mid \\
\exists ifs_1, ifs_2, \text{ph}_1, \text{ph}_2 \,.\; ifs = ifs_1 \bullet ifs_2 \wedge \text{ph} = \text{ph}_1 \bullet \text{ph}_2 \\
\wedge\; [\![\texttt{path}]\!]_\sigma = (\!|\text{P}|\!)_e/(\!|\text{B}|\!)_e/(\!|\text{A}|\!)_e \\
\wedge\; \exists id.\; ifs_1((\!|\alpha|\!)_e) = ((\!|\text{P}|\!)_e, (\!|\text{B}|\!)_e[id]) \vee ifs_1(\mathcal{F}) = \top[id] \wedge id = (\!|\text{C}|\!)_e \\
\wedge\; \neg(\exists id', id'', \iota.\; id = id' + ((\!|\text{A}|\!)_e[id''] \vee (\!|\text{A}|\!)_e : \iota)) \\
\wedge\; \neg(\exists \mathbf{x}, id' .\; id = id' + \mathbf{x}) \\
\wedge\; e, (ifs_2, \text{ph}_2, \sigma) \vDash' r
\end{array} \right\} \\
= \; ass
\end{array}$$

the set of instrumented pre-states. By reification, the pre-states are:

$$\begin{array}{c}
\lfloor ass \rfloor \\
= \\
\left\{ \begin{array}{c}
(fs, \text{ph}, \sigma) | fs \in \text{completion}(as\downarrow_1) \wedge \text{ph} = as\downarrow_2 \wedge \sigma = as\downarrow_3 \\
\wedge\; [\![\texttt{path}]\!]_\sigma = (\!|\text{P}|\!)_e/(\!|\text{B}|\!)_e/(\!|\text{A}|\!)_e \\
\wedge\; \exists id, id', \mathbf{x}. fs(\mathcal{F}) = id \circ_{\mathbf{x}} (\!|\text{B}|\!)_e[id'] \\
\wedge\; \text{resolve}((\!|\text{P}|\!)_e/\mathbf{x}, id) = \mathbf{x} \wedge noent(id', (\!|\text{A}|\!)_e)
\end{array} \right\}
\end{array}$$

Then, by the command interpretation

$$
\begin{array}{c}
[\![\texttt{mkdir(path)}]\!](\lfloor ass \rfloor) \\
= \\
\left\{ \begin{array}{c}
(fs, \mathrm{ph}, \sigma) \mid \mathrm{ph} = as\downarrow_2 \wedge \sigma = as\downarrow_3 \\
\wedge\ [\![\texttt{path}]\!]_\sigma = (\!|\mathrm{P}|\!)_e/(\!|\mathrm{B}|\!)_e/(\!|\mathrm{A}|\!)_e \\
\wedge\ \exists id, id', \mathbf{x}. fs(\mathcal{F}) = id \circ_{\mathbf{x}} (\!|\mathrm{B}|\!)_e[id' + (\!|\mathrm{A}|\!)_e[\varnothing]] \\
\wedge\ \mathrm{resolve}((\!|\mathrm{P}|\!)_e/\mathbf{x}, id) = \mathbf{x}
\end{array} \right\}
\end{array}
$$

we obtain the post-states.

By the satisfaction relation we get the instrumented post-states from the command axiom's postcondition:

$$
\begin{array}{c}
\left[\!\!\left[\alpha^{\mathrm{P}} \mapsto \mathrm{B}[\mathrm{C} + \mathrm{A}[\varnothing]] \star r \right]\!\!\right]_e \\
= \\
\left\{ \begin{array}{c}
(ifs, \mathrm{ph}, \sigma) \in \textsc{AStates} \mid \\
\exists ifs_1, ifs_2, \mathrm{ph}_1, \mathrm{ph}_2 .\ ifs = ifs_1 \bullet ifs_2 \wedge \mathrm{ph} = \mathrm{ph}_1 \bullet \mathrm{ph}_2 \\
\wedge\ [\![\texttt{path}]\!]_\sigma = (\!|\mathrm{P}|\!)_e/(\!|\mathrm{B}|\!)_e/(\!|\mathrm{A}|\!)_e \\
\wedge\ (\exists id.\ ifs_1((\!|\alpha|\!)_e) = ((\!|\mathrm{P}|\!)_e, (\!|\mathrm{B}|\!)_e[id + (\!|\mathrm{A}|\!)_e[\varnothing]]) \\
\wedge\ e, (ifs_2, \mathrm{ph}_2, \sigma) \vDash' r \\
=\ ass'
\end{array} \right\}
\end{array}
$$

where facts about the variable store are derived from logical variable immutability. The instrumented pre and post-states have the same completing pre-instrumented file systems $pifs_c$ in their completions. By this, it is easy to see that:

$$[\![\texttt{mkdir(path)}]\!](\lfloor ass \rfloor) = \lfloor ass' \rfloor$$

Structural and state commands are proven in a similar way.

Now consider the third axiom of the `open` *command. By the satisfaction relation we have:*

$$
\begin{array}{c}
\left[\!\!\left[\texttt{path} \Rightarrow \mathrm{P/A} \wedge \texttt{flags} \Rightarrow \texttt{O\_TRUNC} \wedge \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I} \star \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{S} \star emp \star r \right]\!\!\right]_e \\
= \\
\left\{ \begin{array}{c}
(ifs, \mathrm{ph}, \sigma) \in \textsc{AStates} \mid \\
\exists ifs_1, ifs_2, \mathrm{ph}_1, \mathrm{ph}_2 .\ ifs = ifs_1 \bullet ifs_2 \wedge \mathrm{ph} = \mathrm{ph}_1 \bullet \mathrm{ph}_2 \\
\wedge\ [\![\texttt{path}]\!]_\sigma = (\!|\mathrm{P}|\!)_e/(\!|\mathrm{A}|\!)_e \wedge [\![\texttt{flags}]\!]_\sigma = \texttt{O\_TRUNC} \\
\wedge\ ifs_1((\!|\alpha|\!)_e) = ((\!|\mathrm{P}|\!)_e, (\!|\mathrm{A}|\!)_e : (\!|\mathrm{I}|\!)_e) \wedge ifs_1((\!|\mathrm{I}|\!)_e) = (\!|\mathrm{S}|\!)_e \\
\wedge\ e, (ifs_2, \mathrm{ph}_2, \sigma) \vDash' r \\
=\ ass
\end{array} \right\}
\end{array}
$$

the set of instrumented pre-states. We reify these to pre-states:

$$
\begin{array}{c}
\lfloor ass \rfloor \\
= \\
\left\{ \begin{array}{c}
(fs, \mathrm{ph}, \sigma) | fs \in \mathrm{completion}(as\downarrow_1) \wedge \mathrm{ph} = as\downarrow_2 \wedge \sigma = as\downarrow_3 \\
\wedge\ [\![\texttt{path}]\!]_\sigma = (\!|\mathrm{P}|\!)_e/(\!|\mathrm{A}|\!)_e \wedge [\![\texttt{flags}]\!]_\sigma = \texttt{O\_TRUNC} \\
\wedge\ \exists id.\ fs(\mathcal{F}) = id \circ_{\mathbf{x}} (\!|\mathrm{A}|\!)_e : (\!|\mathrm{I}|\!)_e \\
\wedge\ \mathrm{resolve}((\!|\mathrm{P}|\!)_e/\mathbf{x}, id) = \mathbf{x} \wedge fs(\mathrm{I}) = (\!|\mathrm{S}|\!)_e
\end{array} \right\}
\end{array}
$$

We use the pre-states as input to the command interpretation

$$
\begin{array}{c}
[\![\texttt{fd := open(path, flags)}]\!](\lfloor ass \rfloor) \\
= \\
\left\{ \begin{array}{c}
(fs', \mathrm{ph}', \sigma') \mid \mathrm{ph} = as\downarrow_2 \wedge \sigma = as\downarrow_3 \\
\wedge\ \sigma' = \sigma[\texttt{fd} \mapsto f] \wedge \mathrm{ph}' = \mathrm{ph}[f \mapsto ((\!|\mathrm{I}|\!)_e, 0)] \\
\wedge fs' = fs[\mathrm{I} \mapsto \epsilon] \wedge f \notin dom(\mathrm{ph})
\end{array} \right\}
\end{array}
$$

to obtain the post-states. Again, by the satisfaction relation we get the instrumented post-states.

$$\left[\!\left[\exists \mathrm{FD}.\begin{pmatrix}\mathtt{fd} \Rightarrow \mathrm{FD} \wedge \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I} \\ \star\ \mathrm{I} \overset{\mathrm{F}}{\mapsto} \epsilon \star \mathrm{FD} \overset{\mathrm{OF}}{\mapsto} (\mathrm{I}, 0)\end{pmatrix} \star r\right]\!\right]_e$$
$$=$$
$$\left\{\begin{array}{c}(ifs, \mathrm{ph}, \sigma) \in \textsc{AStates} \mid \\ \exists ifs_1, ifs_2, \mathrm{ph}_1, \mathrm{ph}_2 .\ ifs = ifs_1 \bullet ifs_2 \wedge \mathrm{ph} = \mathrm{ph}_1 \bullet \mathrm{ph}_2 \\ \wedge\ [\![\mathtt{fd}]\!]_\sigma = (\!|\mathrm{FD}|\!)_e \\ \wedge\ ifs_1((\!|\alpha|\!)_e) = ((\!|\mathrm{P}|\!)_e, (\!|\mathrm{A}|\!)_e : (\!|\mathrm{I}|\!)_e) \wedge ifs_1((\!|\mathrm{I}|\!)_e) = \epsilon \\ \wedge\ \mathrm{ph}_1((\!|\mathrm{FD}|\!)_e) = ((\!|\mathrm{I}|\!)_e, 0) \wedge e, (ifs_2, \mathrm{ph}_2, \sigma) \vDash' r \end{array}\right\}$$
$$= ass'$$

Again, the completions of the instrumented pre and post-states are the same and thus it is easy to see that:

$$[\![\texttt{fd := open(path, flags}]\!](\lfloor ass \rfloor) = \lfloor ass' \rfloor$$

Lemma 3 (Entailment Properties). *File System Entailment satisfies Entailment Locality.*

Proof. *Let $p \vDash q$. By the definition of $\vDash$, we have that for all $e \in \textsc{LEnv}$, $[\![p]\!]_e \subseteq [\![q]\!]_e$. We must show that $p \preccurlyeq q$. Recall the definition of $\preccurlyeq$: $p \preccurlyeq q \iff \forall r \in \mathcal{P}(\textsc{AStates}), e \in \textsc{LEnv}, \lfloor p * r \rfloor \subseteq \lfloor q * r \rfloor$. Thus, the lemma requires us to show that if $[\![p]\!]_e \subseteq [\![q]\!]_e$, then $\lfloor [\![p * r]\!]_e \rfloor \subseteq \lfloor [\![q * r]\!]_e \rfloor$, for all choices of logical environments and views r. But, considering lemma 1, this follows immediately as entailment is just subset inclusion.*

Lemma 4 (Disjunction Properties). *File System Disjunction has Join Distributivity and is a Join morphism.*

Proof. *Our Disjunction function is set union. Therefore, for Join Distributivity, we must show $p, q_i \in \mathcal{P}(\textsc{AStates})$ and index set I, $p * \bigcup_{i \in I}\{q_i\} = \bigcup_{i \in I}\{p * q_i\}$. For the Join Morphism property, we need to show $\lfloor \bigcup_{i \in I}\{p_i\} \rfloor = \bigcup_{i \in I}\{\lfloor p_i \rfloor\}$. Both properties follow from the set lifting definition of reification.*

Finally, the soundness of the abstract allocation/deallacotion axioms (definition 11) is justified by the fact that they only change the instrumentation. The reifications of pre- and post-condition are the same. With these results, we can conclude our reasoning is sound.

Proof (Proof of Soundness Theorem). By lemmas 2, 3, 4 according to the Views framework, the partial correctness result follows. The fault avoiding result follows by the fact that $\lightning \notin \textsc{States}$.

$$\{\,\texttt{path} \Mapsto \mathrm{P/B/A} \wedge \texttt{fd} \Rightarrow - \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \mathrm{B}[\mathrm{C} \wedge \mathit{can\_create}(A)]\,\}$$
$$\texttt{fd := open(path, flags)}$$
$$\left\{\,\exists \mathrm{F_D}, \mathrm{I}.\ \begin{pmatrix} \texttt{fd} \Rightarrow \mathrm{F_D} \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \mathrm{B}[\mathrm{C} + \mathrm{A} : \mathrm{I}] \\ \star\ \mathrm{I} \overset{\mathrm{F}}{\mapsto} \varnothing_b \star \mathrm{F_D} \overset{\mathrm{OF}}{\mapsto} (\mathrm{I}, 0) \end{pmatrix}\,\right\}$$

$$\{\,\texttt{path} \Mapsto \mathrm{P/A} \wedge \texttt{flags} \Mapsto 0 \wedge \texttt{fd} \Rightarrow - \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I} \star \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{S}\,\}$$
$$\texttt{fd := open(path, flags)}$$
$$\left\{\,\exists \mathrm{F_D}.\ \begin{pmatrix} \texttt{fd} \Rightarrow \mathrm{F_D} \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I} \\ \star\ \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{S} \star \mathrm{F_D} \overset{\mathrm{OF}}{\mapsto} (\mathrm{I}, 0) \end{pmatrix}\,\right\}$$

$$\{\,\texttt{path} \Mapsto \mathrm{P/A} \wedge \texttt{flags} \Mapsto \texttt{O\_TRUNC} \wedge \texttt{fd} \Rightarrow - \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I} \star \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{S}\,\}$$
$$\texttt{fd := open(path, flags)}$$
$$\left\{\,\exists \mathrm{F_D}.\ \begin{pmatrix} \texttt{fd} \Rightarrow \mathrm{F_D} \star \mathcal{E} \star \alpha^{\mathrm{P}} \mapsto \mathrm{A} : \mathrm{I} \\ \star\ \mathrm{I} \overset{\mathrm{F}}{\mapsto} \varnothing_b \star \mathrm{F_D} \overset{\mathrm{OF}}{\mapsto} (\mathrm{I}, 0) \end{pmatrix}\,\right\}$$
$$\{\,\texttt{fd} \Rightarrow \mathrm{F_D} \star \mathrm{F_D} \overset{\mathrm{OF}}{\mapsto} (\mathrm{I}, \textsc{Offset})\,\}$$
$$\texttt{close(fd)}$$
$$\{\,\mathrm{emp}\,\}$$

$$\{\,\texttt{fd} \Rightarrow \mathrm{F_D} \wedge \texttt{to} \Mapsto \mathrm{O}' \wedge \texttt{wh} \Mapsto \texttt{SEEK\_SET} \wedge \mathrm{O}' \geq 0 \star \texttt{off} \Rightarrow - \star \mathcal{E} \star \mathrm{F_D} \overset{\mathrm{OF}}{\mapsto} (\mathrm{I}, \mathrm{O})\,\}$$
$$\texttt{off := lseek(fd, to, wh)}$$
$$\{\,\texttt{off} \Rightarrow \mathrm{O}' \star \texttt{fd} \Rightarrow \mathrm{F_D} \star \mathcal{E} \star \mathrm{F_D} \overset{\mathrm{OF}}{\mapsto} (\mathrm{I}, \mathrm{O}')\,\}$$

$$\{\,\texttt{fd} \Rightarrow \mathrm{F_D} \wedge \texttt{to} \Mapsto \mathrm{O}' \wedge \texttt{wh} \Mapsto \texttt{SEEK\_CUR} \wedge (\mathrm{O} + \mathrm{O}' \geq 0) \star \texttt{off} \Rightarrow - \star \mathcal{E} \star \mathrm{F_D} \overset{\mathrm{OF}}{\mapsto} (\mathrm{I}, \mathrm{O})\,\}$$
$$\texttt{off := lseek(fd, to, wh)}$$
$$\{\,\texttt{off} \Rightarrow \mathrm{O} + \mathrm{O}' \star \texttt{fd} \Rightarrow \mathrm{F_D} \star \mathcal{E} \star \mathrm{F_D} \overset{\mathrm{OF}}{\mapsto} (\mathrm{I}, \mathrm{O} + \mathrm{O}')\,\}$$

$$\left\{\,\begin{array}{c} \texttt{fd} \Rightarrow \mathrm{F_D} \wedge \texttt{to} \Mapsto \mathrm{O}' \wedge \texttt{wh} \Mapsto \texttt{SEEK\_END} \wedge |\mathrm{S}| + \mathrm{O}' \geq 0 \\ \texttt{off} \Rightarrow - \star \mathcal{E} \star \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{S} \star \mathrm{F_D} \overset{\mathrm{OF}}{\mapsto} (\mathrm{I}, \mathrm{O}) \end{array}\,\right\}$$
$$\texttt{off := lseek(fd, to, wh)}$$
$$\{\,\texttt{off} \Rightarrow |\mathrm{S}| + \mathrm{O}' \star \texttt{fd} \Rightarrow \mathrm{F_D} \star \mathcal{E} \star \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{S} \star \mathrm{F_D} \overset{\mathrm{OF}}{\mapsto} (\mathrm{I}, |\mathrm{S}| + \mathrm{O}')\,\}$$

$$\left\{\,\begin{array}{c} \texttt{fd} \Rightarrow \mathrm{F_D} \wedge \texttt{buf} \Mapsto \mathrm{S} \wedge |\mathrm{S}'| = \mathrm{O} \wedge |\mathrm{S}''| = |\mathrm{S}| \\ \star\ \texttt{sz} \Rightarrow - \star \mathcal{E} \star \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{S}' \cdot \mathrm{S}'' \cdot \mathrm{S}''' \star \mathrm{F_D} \overset{\mathrm{OF}}{\mapsto} (\mathrm{I}, \mathrm{O}) \end{array}\,\right\}$$
$$\texttt{sz := write(fd, buf)}$$
$$\{\,\texttt{sz} \Rightarrow |\mathrm{S}| \star \texttt{fd} \Rightarrow \mathrm{F_D} \star \mathcal{E} \star \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{S}' \cdot \mathrm{S} \cdot \mathrm{S}''' \star \mathrm{F_D} \overset{\mathrm{OF}}{\mapsto} (\mathrm{I}, \mathrm{O} + |\mathrm{S}|)\,\}$$

$$\left\{\,\begin{array}{c} \texttt{fd} \Rightarrow \mathrm{F_D} \wedge \texttt{buf} \Mapsto \mathrm{S} \wedge |\mathrm{S}'| = \mathrm{O} \wedge |\mathrm{S}''| < |\mathrm{S}| \\ \star\ \texttt{sz} \Rightarrow - \star \mathcal{E} \star \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{S}' \cdot \mathrm{S}'' \star \mathrm{F_D} \overset{\mathrm{OF}}{\mapsto} (\mathrm{I}, \mathrm{O}) \end{array}\,\right\}$$
$$\texttt{sz := write(fd, buf)}$$
$$\{\,\texttt{sz} \Rightarrow |\mathrm{S}| \star \texttt{fd} \Rightarrow \mathrm{F_D} \star \mathcal{E} \star \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{S}' \cdot \mathrm{S} \star \mathrm{F_D} \overset{\mathrm{OF}}{\mapsto} (\mathrm{I}, \mathrm{O} + |\mathrm{S}|)\,\}$$

$$\{\,\texttt{fd} \Rightarrow \mathrm{F_D} \wedge \texttt{buf} \Mapsto \mathrm{S} \wedge \mathrm{O} \geq |\mathrm{S}'| \wedge \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{S}' \star \texttt{sz} \Rightarrow - \star \mathcal{E} \star \mathrm{F_D} \overset{\mathrm{OF}}{\mapsto} (\mathrm{I}, \mathrm{O})\,\}$$
$$\texttt{sz := write(fd, buf)}$$
$$\left\{\,\texttt{sz} \Rightarrow |\mathrm{S}| \star \texttt{fd} \Rightarrow \mathrm{F_D} \star \mathcal{E} \star \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{S}' \underset{\mathrm{O}-|\mathrm{S}'|}{\odot} 0 \cdot \mathrm{S} \star \mathrm{F_D} \overset{\mathrm{OF}}{\mapsto} (\mathrm{I}, \mathrm{O} + |\mathrm{S}|)\,\right\}$$

$$\left\{\,\begin{array}{c} \texttt{fd} \Rightarrow \mathrm{F_D} \wedge \texttt{sz} \Mapsto \textsc{Len} \wedge |\mathrm{S}| = \mathrm{O} \wedge |\mathrm{S}'| = \textsc{Len} \\ \star\ \texttt{buf} \Rightarrow - \star \mathcal{E} \star \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{S} \cdot \mathrm{S}' \cdot \mathrm{S}'' \star \mathrm{F_D} \overset{\mathrm{OF}}{\mapsto} (\mathrm{I}, \mathrm{O}) \end{array}\,\right\}$$
$$\texttt{buf := read(fd, sz)}$$
$$\{\,\texttt{buf} \Rightarrow \mathrm{S}' \star \texttt{fd} \Rightarrow \mathrm{F_D} \star \mathcal{E} \star \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{S} \cdot \mathrm{S}' \cdot \mathrm{S}'' \star \mathrm{F_D} \overset{\mathrm{OF}}{\mapsto} (\mathrm{I}, \mathrm{O} + \textsc{Len})\,\}$$

$$\left\{\,\begin{array}{c} \texttt{fd} \Rightarrow \mathrm{F_D} \wedge \texttt{sz} \Mapsto \textsc{Len} \wedge |\mathrm{S}| = \mathrm{O} \wedge \textsc{len} > |\mathrm{S}'| \\ \star\ \texttt{buf} \Rightarrow - \star \mathcal{E} \star \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{S} \cdot \mathrm{S}' \star \mathrm{F_D} \overset{\mathrm{OF}}{\mapsto} (\mathrm{I}, \mathrm{O}) \end{array}\,\right\}$$
$$\texttt{buf := read(fd, sz)}$$
$$\{\,\texttt{buf} \Rightarrow \mathrm{S}' \star \texttt{fd} \Rightarrow \mathrm{F_D} \star \mathcal{E} \star \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{S} \cdot \mathrm{S}' \star \mathrm{F_D} \overset{\mathrm{OF}}{\mapsto} (\mathrm{I}, \mathrm{O} + |\mathrm{S}'|)\,\}$$

$$\{\,\texttt{fd} \Rightarrow \mathrm{F_D} \wedge \texttt{sz} \Mapsto \textsc{Len} \wedge \mathrm{O} > |\mathrm{S}| \star \texttt{buf} \Rightarrow - \star \mathcal{E} \star \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{S} \star \mathrm{F_D} \overset{\mathrm{OF}}{\mapsto} (\mathrm{I}, \mathrm{O})\,\}$$
$$\texttt{buf := read(fd, sz)}$$
$$\{\,\texttt{buf} \Rightarrow \varnothing_b \star \texttt{fd} \Rightarrow \mathrm{F_D} \star \mathcal{E} \star \mathrm{I} \overset{\mathrm{F}}{\mapsto} \mathrm{S} \star \mathrm{F_D} \overset{\mathrm{OF}}{\mapsto} (\mathrm{I}, \mathrm{O})\,\}$$

Fig. 7. Axiomatic specifications for IO commands.

$$
\begin{aligned}
\textsc{enoent}_*(\texttt{path}) &\triangleq (\texttt{path} \Mapsto \epsilon) \vee \left(\begin{array}{c} \exists \mathrm{P}, \mathrm{A}, \mathrm{B}, \mathrm{P}'.\ \texttt{path} \Mapsto \mathrm{P/A/B/P}' \\ \wedge\ \alpha^{\mathrm{P}} \mapsto \mathrm{A}\,[\mathbf{true} \wedge \neg \mathit{top\_entry}(\mathrm{B}) \wedge \neg \mathit{top\_address}] \end{array} \right) \\
\textsc{enoent}(\texttt{path}) &\triangleq (\texttt{path} \Mapsto \epsilon) \vee \left(\begin{array}{c} \exists \mathrm{P}, \mathrm{A}, \mathrm{B}, \mathrm{P}'.\ \texttt{path} \Mapsto \mathrm{P/A/B/P}' \wedge \mathrm{P}' \neq \epsilon \\ \wedge\ \alpha^{\mathrm{P}} \mapsto \mathrm{A}\,[\mathbf{true} \wedge \neg \mathit{top\_entry}(\mathrm{B}) \wedge \neg \mathit{top\_address}] \end{array} \right) \\
\textsc{enotdir}_*(\texttt{path}) &\triangleq \exists \mathrm{P}, \mathrm{A}, \mathrm{P}'.\ \texttt{path} \Mapsto \mathrm{P/A/P}' \wedge \alpha^{\mathrm{P}} \mapsto \exists \mathrm{I}.\ \mathrm{A} : \mathrm{I} \\
\textsc{enotdir}(\texttt{path}) &\triangleq \exists \mathrm{P}, \mathrm{A}, \mathrm{P}'.\ \texttt{path} \Mapsto \mathrm{P/A/P}' \wedge \alpha^{\mathrm{P}} \mapsto \exists \mathrm{I}.\ \mathrm{A} : \mathrm{I} \\
\textsc{eexist}(\texttt{path}) &\triangleq \exists \mathrm{P}, \mathrm{A}.\ \texttt{path} \Mapsto \mathrm{P/A}\ \ \wedge \alpha^{\mathrm{P}} \mapsto \mathit{entry}(\mathrm{A}) \\
\textsc{enotempty}(\texttt{path}) &\triangleq \exists \mathrm{P}, \mathrm{A}.\ \texttt{path} \Mapsto \mathrm{P}\ \ \wedge \alpha^{\mathrm{P}} \mapsto \mathit{entry}(\mathrm{A}) \\
\textsc{eperm}(\texttt{path}) &\triangleq \exists \mathrm{P}, \mathrm{A}.\ \texttt{path} \Mapsto \mathrm{P/A} \wedge \alpha^{\mathrm{P}} \mapsto \mathrm{A}[\mathbf{true}] \\
\textsc{eisdir}(\texttt{path}) &\triangleq \textsc{eperm}(\texttt{path}) \\
\textsc{einval}(\texttt{old}, \texttt{new}) &\triangleq \exists \mathrm{P}, \mathrm{P}', \mathrm{A}.\ \texttt{old} \Mapsto \mathrm{P/A} \wedge \texttt{new} \Mapsto \mathrm{P}' \wedge \alpha^{\mathrm{P}} \mapsto (\mathrm{A}[\mathrm{C}] \wedge \exists \mathrm{P}''.\ @\mathrm{A/P}''/\beta) \star\ \beta^{\mathrm{P}'} \mapsto \mathbf{true}
\end{aligned}
$$

Fig. 8. Definitions of the error predicates used in command specifications.

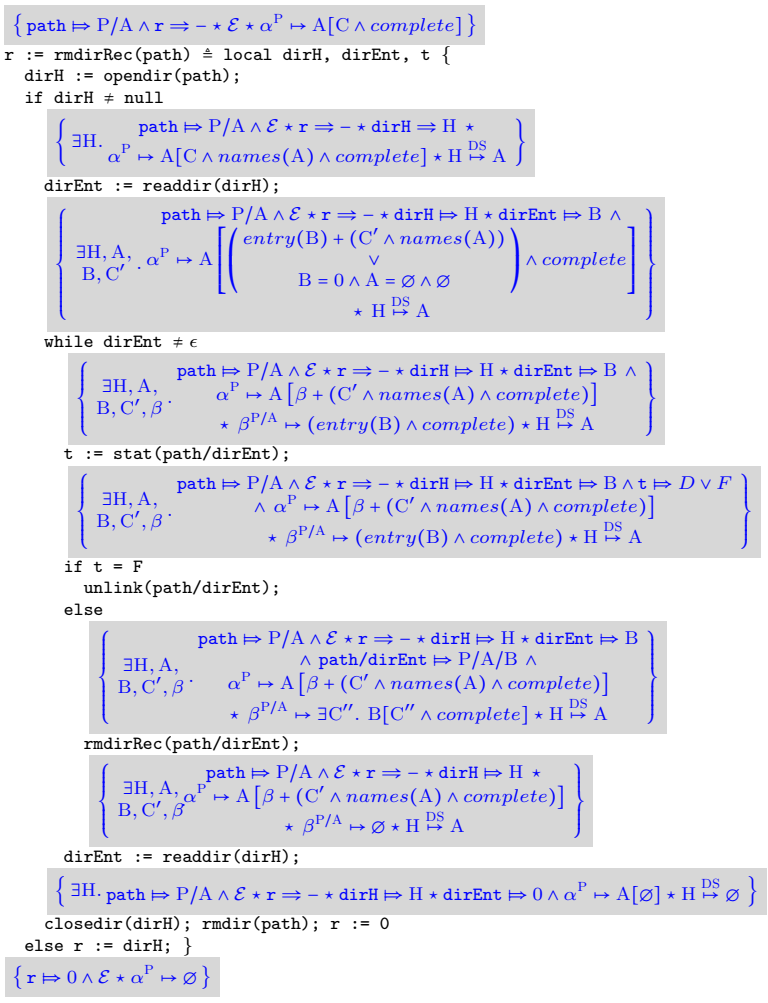

Fig. 9. Proof that `rmdirRec` meets its specification (we sketch only one branch of the `if` for space). We use the rule of disjunction to merge the two possible results for `stat`.

Fig. 10. Code for `fileCopy`, with a proof sketch of its specification.

$$ent(id, a) \triangleq \exists \iota.\ id = a : \iota \vee \exists id'.\ id = a[id']$$
$$noent(id, a) \triangleq \nexists\, id', id''.id \equiv ent(id') + id''$$

$$[\![\texttt{mkdir(path)}]\!](fs, \mathrm{ph}, \sigma) = \begin{cases} \{(fs[\mathcal{F} \mapsto id \circ_{\mathbf{x}} b[id' + a[\varnothing]]], \mathrm{ph}, \sigma)\} & \text{iff } [\![\texttt{path}]\!]_\sigma = p/b/a \wedge fs(\mathcal{F}) \equiv id \circ_{\mathbf{x}} b[id'] \wedge \mathrm{resolve}(p/\mathbf{x}, id) = \mathbf{x} \wedge noent(id', a) \\ \{(fs[\mathcal{F} \mapsto \top[id + a[\varnothing]]], \mathrm{ph}, \sigma)\} & \text{iff } [\![\texttt{path}]\!]_\sigma = \top/a \wedge fs(\mathcal{F}) \equiv \top[d] \wedge noent(d, a) \\ \{\lightning\} & \text{otherwise} \end{cases}$$

$$[\![\texttt{rmdir(path)}]\!](fs, \mathrm{ph}, \sigma) = \begin{cases} \{(fs[\mathcal{F} \mapsto id], \mathrm{ph}, \sigma)\} & \text{iff } [\![\texttt{path}]\!]_\sigma = p/a \wedge fs(\mathcal{F}) \equiv id \circ_{\mathbf{x}} a[\varnothing] \wedge \mathrm{resolve}(p/\mathbf{x}, id) = \mathbf{x} \\ \{\lightning\} & \text{otherwise} \end{cases}$$

$$[\![\texttt{link(existing, new)}]\!](fs, \mathrm{ph}, \sigma) = \begin{cases} \{(fs[\mathcal{F} \mapsto (id \circ_{\mathbf{x}} d[id' + b : \iota]) \circ_{\mathbf{y}} a : \iota], \mathrm{ph}, \sigma)\} & \text{iff } \begin{array}{l} [\![\texttt{existing}]\!]_\sigma = p/a \wedge [\![\texttt{new}]\!]_\sigma = p'/d/b \\ \wedge\ fs(\mathcal{F}) \equiv (id \circ_{\mathbf{x}} d[id']) \circ_{\mathbf{y}} a : \iota \wedge \mathrm{resolve}(p'/\mathbf{x}, id) = \mathbf{x} \\ \wedge\ \mathrm{resolve}(p/\mathbf{y}, id \circ_{\mathbf{x}} d[id']) = \mathbf{y} \wedge noent(id', b) \end{array} \\ \{(fs[\mathcal{F} \mapsto (\top[id' + b : \iota]) \circ_{\mathbf{x}} a : \iota], \mathrm{ph}, \sigma)\} & \text{iff } \begin{array}{l} [\![\texttt{existing}]\!]_\sigma = p/a \wedge [\![\texttt{new}]\!]_\sigma = \top/b \\ \wedge\ fs(\mathcal{F}) \equiv \top[id'] \circ_{\mathbf{x}} a : \iota \wedge \mathrm{resolve}(p'/\mathbf{x}, \top[id]) = \mathbf{x} \\ \wedge\ noent(id', b) \end{array} \\ \{\lightning\} & \text{otherwise} \end{cases}$$

$$[\![\texttt{unlink(path)}]\!](fs, \mathrm{ph}, \sigma) = \begin{cases} \{(fs[\mathcal{F} \mapsto id \circ_{\mathbf{x}} \varnothing], \mathrm{ph}, \sigma)\} & \text{iff } [\![\texttt{path}]\!]_\sigma = p/a \wedge fs(\mathcal{F}) \equiv id \circ_{\mathbf{x}} a : \iota \wedge \mathrm{resolve}(p/\mathbf{x}, id) = \mathbf{x} \\ \{\lightning\} & \text{otherwise} \end{cases}$$

$$[\![\texttt{rename(old, new)}]\!](fs, \mathrm{ph}, \sigma) = \begin{cases} \{(fs[\mathcal{F} \mapsto (id \circ_{\mathbf{x}} \varnothing) \circ_{\mathbf{y}} b : \iota], \mathrm{ph}, \sigma)\} & \text{iff } \begin{array}{l} [\![\texttt{old}]\!]_\sigma = p/a \wedge [\![\texttt{new}]\!]_\sigma = p'/b \\ \wedge\ fs(\mathcal{F}) \equiv (id \circ_{\mathbf{x}} a : \iota) \circ_{\mathbf{y}} b : j \wedge \mathrm{resolve}(p/\mathbf{x}, id) = \mathbf{x} \\ \wedge\ \mathrm{resolve}(p'/\mathbf{y}, id) = \mathbf{y} \end{array} \\ \{(fs[\mathcal{F} \mapsto (id \circ_{\mathbf{x}} d[id' + b : \iota]) \circ_{\mathbf{y}} \varnothing], \mathrm{ph}, \sigma)\} & \text{iff } \begin{array}{l} [\![\texttt{old}]\!]_\sigma = p/a \wedge [\![\texttt{new}]\!]_\sigma = p'/d/b \\ \wedge\ fs(\mathcal{F}) \equiv (id \circ_{\mathbf{x}} d[id']) \circ_{\mathbf{y}} a : \iota \wedge \mathrm{resolve}(p'/\mathbf{x}, id) = \mathbf{x} \\ \wedge\ \mathrm{resolve}(p/\mathbf{y}, id \circ_{\mathbf{x}} d[id']) = \mathbf{y} \wedge noent(id', b) \end{array} \\ \{(fs[\mathcal{F} \mapsto \top[id + b : \iota] \circ_{\mathbf{x}} \varnothing], \mathrm{ph}, \sigma)\} & \text{iff } \begin{array}{l} [\![\texttt{old}]\!]_\sigma = p/a \wedge [\![\texttt{new}]\!]_\sigma = \top/b \\ \wedge\ fs(\mathcal{F}) \equiv \top[id] \circ_{\mathbf{x}} a : \iota \wedge \mathrm{resolve}(p/\mathbf{x}, \top[id]) = \mathbf{x} \\ \wedge\ noent(id, b) \end{array} \\ \{(fs[\mathcal{F} \mapsto (id \circ_{\mathbf{x}} b[id']) \circ_{\mathbf{y}} \varnothing], \mathrm{ph}, \sigma)\} & \text{iff } \begin{array}{l} [\![\texttt{old}]\!]_\sigma = p/a \wedge [\![\texttt{new}]\!]_\sigma = p'/b \\ \wedge\ fs(\mathcal{F}) \equiv (id \circ_{\mathbf{x}} b[\varnothing]) \circ_{\mathbf{y}} a[id'] \wedge \mathrm{resolve}(p/\mathbf{x}, id) = \mathbf{x} \\ \wedge\ \mathrm{resolve}(p'/\mathbf{y}, id) = \mathbf{y} \end{array} \\ \{(fs[\mathcal{F} \mapsto (id \circ_{\mathbf{x}} d[id' + b[id'']]) \circ_{\mathbf{y}} \varnothing], \mathrm{ph}, \sigma)\} & \text{iff } \begin{array}{l} [\![\texttt{old}]\!]_\sigma = p/a \wedge [\![\texttt{new}]\!]_\sigma = p'/d/b \\ \wedge\ fs(\mathcal{F}) \equiv (id \circ_{\mathbf{x}} d[id']) \circ_{\mathbf{y}} a[id''] \wedge \mathrm{resolve}(p'/\mathbf{x}, id) = \mathbf{x} \\ \wedge\ \mathrm{resolve}(p/\mathbf{y}, id \circ_{\mathbf{x}} d[id']) = \mathbf{y} \wedge noent(id', b) \end{array} \\ \{(fs[\mathcal{F} \mapsto \top[id + b[id']] \circ_{\mathbf{x}} \varnothing], \mathrm{ph}, \sigma)\} & \text{iff } \begin{array}{l} [\![\texttt{old}]\!]_\sigma = p/a \wedge [\![\texttt{new}]\!]_\sigma = \top/b \\ \wedge\ fs(\mathcal{F}) \equiv \top[id] \circ_{\mathbf{x}} a[id'] \wedge \mathrm{resolve}(p/\mathbf{x}, \top[id]) = \mathbf{x} \\ \wedge\ noent(id, b) \end{array} \\ \{(fs, \mathrm{ph}, \sigma)\} & \text{iff } \begin{array}{l} [\![\texttt{old}]\!]_\sigma = p/a \wedge [\![\texttt{new}]\!]_\sigma = p/a \\ \wedge\ fs(\mathcal{F}) \equiv id \circ_{\mathbf{x}} id' \wedge \mathrm{resolve}(p/\mathbf{x}, id) = \mathbf{x} \\ \wedge\ ent(id', a) \end{array} \\ \{\lightning\} & \text{otherwise} \end{cases}$$

$$[\![\texttt{t := stat(path)}]\!](fs, \mathrm{ph}, \sigma) = \begin{cases} \{(fs, \mathrm{ph}, \sigma[\mathtt{t} \mapsto \mathrm{F})\} & \text{iff } [\![\texttt{path}]\!]_\sigma = p/a \wedge fs(\mathcal{F}) \equiv id \circ_{\mathbf{x}} a : \iota \wedge \mathrm{resolve}(p/\mathbf{x}, id) = \mathbf{x} \\ \{(fs, \mathrm{ph}, \sigma[\mathtt{t} \mapsto \mathrm{D})\} & \text{iff } [\![\texttt{path}]\!]_\sigma = p/a \wedge fs(\mathcal{F}) \equiv id \circ_{\mathbf{x}} a[id'] \wedge \mathrm{resolve}(p/\mathbf{x}, id) = \mathbf{x} \\ \{(fs, \mathrm{ph}, \sigma[\mathtt{t} \mapsto \textsc{enoent})\} & \text{iff } [\![\texttt{path}]\!]_\sigma = p/a \wedge fs(\mathcal{F}) \equiv \top[d] \wedge \mathrm{resolve}(p/a, \top[d]) = udf \\ \{\lightning\} & \text{otherwise}n \end{cases}$$

Fig. 11. Interpretation of the *structural* and *state* commands of our subset.

$$[\![\texttt{fd := open(path, flags)}]\!](fs, \mathrm{ph}, \sigma) = \begin{cases} \{(fs, \mathrm{ph}[f \mapsto (\iota, 0)][\iota \mapsto \epsilon], \sigma[\texttt{fd} \mapsto f])\} & \text{iff } [\![\texttt{path}]\!]_\sigma = p/b/a \wedge fs(\mathcal{F}) \equiv id \circ_{\mathbf{x}} b[id'] \wedge \mathrm{resolve}(p/\mathbf{x}, id) = \mathbf{x} \\ & \quad \wedge\ noent(id', a) \wedge f \notin dom(\mathrm{ph}) \wedge \iota \notin dom(fs) \\ \{(fs, \mathrm{ph}[f \mapsto (\iota, 0)][\iota \mapsto \epsilon], \sigma[\texttt{fd} \mapsto f])\} & \text{iff } [\![\texttt{path}]\!]_\sigma = \top/a \wedge fs(\mathcal{F}) \equiv \top[d] \\ & \quad \wedge\ noent(d, a) \wedge f \notin dom(\mathrm{ph}) \wedge \iota \notin dom(fs) \\ \{(fs, \mathrm{ph}[f \mapsto (\iota, 0)], \sigma[\texttt{fd} \mapsto f])\} & \text{iff } [\![\texttt{path}]\!]_\sigma = p/a \wedge [\![\texttt{flags}]\!]_\sigma = 0 \wedge fs(\mathcal{F}) \equiv id \circ_{\mathbf{x}} a : \iota \\ & \quad \wedge\ \mathrm{resolve}(p/\mathbf{x}, id) = \mathbf{x} \wedge f \notin dom(\mathrm{ph}) \wedge \iota \in dom(fs) \\ \{(fs, \mathrm{ph}[f \mapsto (\iota, 0)][\iota \mapsto \epsilon], \sigma[\texttt{fd} \mapsto f])\} & \text{iff } [\![\texttt{path}]\!]_\sigma = p/a \wedge [\![\texttt{flags}]\!]_\sigma = \texttt{O\_TRUNC} \wedge fs(\mathcal{F}) \equiv id \circ_{\mathbf{x}} a : \iota \\ & \quad \wedge\ \mathrm{resolve}(p/\mathbf{x}, id) = \mathbf{x} \wedge f \notin dom(\mathrm{ph}) \wedge \iota \in dom(fs) \\ \{\lightning\} & \text{otherwise} \end{cases}$$

$$[\![\texttt{off := lseek(fd, to, wh)}]\!](fs, \mathrm{ph}, \sigma) = \begin{cases} \{(fs, \mathrm{ph}[f \mapsto (\iota, offset)], \sigma[\texttt{off} \mapsto offset])\} & \text{iff } [\![\texttt{fd}]\!]_\sigma = f \wedge [\![\texttt{to}]\!]_\sigma = offset \wedge [\![\texttt{wh}]\!]_\sigma = \texttt{SEEK\_SET} \\ & \quad \wedge\ \mathrm{ph}(f) = (\iota, o) \wedge offset \geq 0 \\ \{(fs, \mathrm{ph}[f \mapsto (\iota, o + offset)], \sigma[\texttt{off} \mapsto o + offset])\} & \text{iff } [\![\texttt{fd}]\!]_\sigma = f \wedge [\![\texttt{to}]\!]_\sigma = offset \wedge [\![\texttt{wh}]\!]_\sigma = \texttt{SEEK\_CUR} \\ & \quad \wedge\ \mathrm{ph}(f) = (\iota, o) \wedge o + offset \geq 0 \\ \{(fs, \mathrm{ph}[f \mapsto (\iota, |bs| + offset)], \sigma[\texttt{off} \mapsto |bs| + offset])\} & \text{iff } [\![\texttt{fd}]\!]_\sigma = f \wedge [\![\texttt{to}]\!]_\sigma = offset \wedge [\![\texttt{wh}]\!]_\sigma = \texttt{SEEK\_END} \\ & \quad \wedge\ \mathrm{ph}(f) = (\iota, o) \wedge fs(\iota) = bs \wedge |bs| + offset \geq 0 \\ \{\lightning\} & \text{otherwise} \end{cases}$$

$$[\![\texttt{sz := write(fd, buf)}]\!](fs, \mathrm{ph}, \sigma) = \begin{cases} \{(fs[\iota \mapsto bs' \cdot bs \cdot bs''], \mathrm{ph}[f \mapsto (\iota, |bs| + o)], \sigma[\texttt{sz} \mapsto |bs|])\} & \text{iff } [\![\texttt{fd}]\!]_\sigma = f \wedge [\![\texttt{buf}]\!]_\sigma = bs \wedge \mathrm{ph}(f) = (\iota, o) \\ & \quad \wedge\ fs(\iota) \equiv bs' \cdot bs''' \cdot bs'' \wedge |bs'| = o \wedge |bs'''| = |bs| \\ \{(fs[\iota \mapsto bs' \cdot bs], \mathrm{ph}[f \mapsto (\iota, |bs| + o)], \sigma[\texttt{sz} \mapsto |bs|])\} & \text{iff } [\![\texttt{fd}]\!]_\sigma = f \wedge [\![\texttt{buf}]\!]_\sigma = bs \wedge \mathrm{ph}(f) = (\iota, o) \\ & \quad \wedge\ fs(\iota) \equiv bs' \cdot bs'' \wedge |bs'| = o \wedge |bs''| < |bs| \\ \{(fs[\iota \mapsto bs'(\underset{zs}{\odot} 0) \cdot bs], \mathrm{ph}[f \mapsto (\iota, |bs| + o)], \sigma[\texttt{sz} \mapsto |bs|])\} & \text{iff } [\![\texttt{fd}]\!]_\sigma = f \wedge [\![\texttt{buf}]\!]_\sigma = bs \wedge \mathrm{ph}(f) = (\iota, o) \\ & \quad \wedge\ fs(\iota) \equiv bs' \wedge o \geq |bs'| \wedge zs = o - |bs'| \\ \{\lightning\} & \text{otherwise} \end{cases}$$

$$[\![\texttt{buf := read(fd, sz)}]\!](fs, \mathrm{ph}, \sigma) = \begin{cases} \{(fs, \mathrm{ph}[f \mapsto (\iota, o + size)], \sigma[\texttt{buf} \mapsto bs'])\} & \text{iff } [\![\texttt{fd}]\!]_\sigma = f \wedge [\![\texttt{sz}]\!]_\sigma = size \in \mathbb{N} \wedge \mathrm{ph}(f) = (\iota, o) \\ & \quad \wedge\ fs(\iota) \equiv bs \cdot bs' \cdot bs'' \wedge |bs| = o \wedge size = |bs'| \\ \{(fs, \mathrm{ph}[f \mapsto (\iota, o + |bs'|)], \sigma[\texttt{buf} \mapsto bs'])\} & \text{iff } [\![\texttt{fd}]\!]_\sigma = f \wedge [\![\texttt{sz}]\!]_\sigma = size \in \mathbb{N} \wedge \mathrm{ph}(f) = (\iota, o) \\ & \quad \wedge\ fs(\iota) \equiv bs \cdot bs' \wedge |bs| = o \wedge size > |bs'| \\ \{(fs, \mathrm{ph}, \sigma[\texttt{buf} \mapsto \epsilon])\} & \text{iff } [\![\texttt{fd}]\!]_\sigma = f \wedge [\![\texttt{sz}]\!]_\sigma = size \in \mathbb{N} \wedge \mathrm{ph}(f) = (\iota, o) \\ & \quad \wedge\ fs(\iota) \equiv bs \wedge o > |bs| \\ \{\lightning\} & \text{otherwise} \end{cases}$$

$$[\![\texttt{close(fd)}]\!](fs, \mathrm{ph}, \sigma) = \begin{cases} \{(fs, \mathrm{ph} \upharpoonright (dom(\mathrm{ph}) - \{f\}), \sigma)\} & \text{iff } [\![\texttt{fd}]\!]_\sigma = f \wedge f \in dom(\mathrm{ph}) \\ \{\lightning\} & \text{otherwise} \end{cases}$$

$$[\![\texttt{dir := opendir(path)}]\!](fs, \mathrm{ph}, \sigma) = \begin{cases} \{(fs, \mathrm{ph}[n \mapsto A], \sigma[\texttt{dir} \mapsto n])\} & \text{iff } [\![\texttt{path}]\!]_\sigma = p/b \wedge fs(\mathcal{F}) \equiv id \circ_{\mathbf{x}} b[id'] \wedge \mathrm{resolve}(p/\mathbf{x}, id) = \mathbf{x} \\ & \quad \wedge\ n \notin dom(\mathrm{ph}) \wedge \forall a.\ a \in A \iff \exists id'', id'''.\ id' \equiv id'' + ent(id''', a) \\ \{\lightning\} & \text{otherwise} \end{cases}$$

$$[\![\texttt{fn := readdir(dir)}]\!](fs, \mathrm{ph}, \sigma) = \begin{cases} \{(fs, \mathrm{ph}[n \mapsto A \setminus \{a\}], \sigma[\texttt{fn} \mapsto a])\} & \text{iff } [\![\texttt{dir}]\!]_\sigma = n \wedge \mathrm{ph}(n) = A \wedge a \in A \\ \{(fs, \mathrm{ph}, \sigma[\texttt{fn} \mapsto 0])\} & \text{iff } [\![\texttt{dir}]\!]_\sigma = n \wedge \mathrm{ph}(n) = \varnothing \\ \{\lightning\} & \text{otherwise} \end{cases}$$

Fig. 12. Interpretation of *IO* commands of our subset.