Local Reasoning for the POSIX File System

Philippa Gardner, Gian Ntzik, and Adam Wright

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

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

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

1 Introduction

Local reasoning, in the style of separation logic, was introduced to reason about programs that manipulate the RAM memory model. Local reasoning has strong modular properties, which means that it scales. Many forms of *abstract* local reasoning have been introduced to specify structured data libraries: e.g. abstract predicates for linked lists [20], concurrent abstract predicates for abstract concurrent sets [7, 24, 23], and context logic for complex structured data such as the DOM [13]. Despite these advances, there are many other properties of real-world libraries that naturally resonate with this local-reasoning approach but have yet to be studied.

We study abstract local reasoning for the POSIX file system [2]. POSIX has an English specification which naturally describes commands which globally follow directory paths to locally update files or directories[1]. There has been much work on traditional reasoning techniques for specifying POSIX, such as the well-known Z specification [18]. However, the global path constraints associated with this work are substantial. Our aim is to use local reasoning to minimise the global path constraints. POSIX is an interesting test case for abstract local reasoning. It has enough emphasis on local update to suggest that the advantages of local reasoning might apply. However, the complexity of the data combined with concurrency, global paths and local update means that the application of local reasoning to this example is not straightforward.

Current work on abstract local reasoning cannot specify POSIX. For example, context logic works well for reasoning about sequential update of complex data, such as the W3C DOM library for XML update [13, 5, 22]. However, it has no mechanism for reasoning about global paths, it does not extend simply to concurrency, and it does not integrate well with ideas from separation logic. Concurrent abstract predicates [7, 24, 23] work well for reasoning abstractly about

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

Z. Shao (Ed.): ESOP 2014, LNCS 8410, pp. 169–188, 2014.

simple concurrent data structures. However, they do not extend to complex data structures since the implementation details leak into the abstraction [12].

We introduce structural separation logic (SSL) for reasoning abstractly about complex structured data. SSL provides more fine-grained reasoning than context logic, leading to straightforward reasoning about disjoint concurrency and a natural integration with separation logic. Here, we demonstrate SSL by reasoning about POSIX. In [25, 12], we provide the general theory which relies substantially on ideas from the views framework [6][2]. SSL combines fine-grained local reasoning about e.g. directory fragments with global path constraints about the overall structured data. The global path constraints limit the use of the frame rule in our sequential setting, and specify stability requirements of the environment in the concurrent setting. We illustrate our ideas using absolute linear paths, called paths in this paper. In future, it will be very interesting to study general paths (with the backwards .. and symbolic links) as part of our abstract local reasoning agenda, since they walk right across the directory structure.

In this paper, we use SSL to reason about the sequential POSIX file system, demonstrating that our axioms correspond to the English description given in the POSIX standard. We identify a core subset of POSIX, which is both faithful to the standard and a natural subset with which to introduce our reasoning. We model various structures of the file-system state as standard heaps: file heaps mapping file identifiers (inodes) to bytes; and file-descriptor heaps mapping file descriptors to input/output related data. Separation logic can reason about these heap structures. We require SSL to reason about the directory structure, which we regard as a tree-shaped hierarchy[3]. SSL naturally integrates with separation logic, enabling us to reason about directories and the standard heap within the same logic. We demonstrate this integrated reasoning by verifying natural safety properties of a client software installer. Although we concentrate on sequential POSIX in this paper, our results immediately extend to POSIX with disjoint concurrency. In future, we will explore POSIX with shared-memory concurrency.

Related Work. There has been substantial work on formal specifications of file systems [15, 18, 9, 4], leading to a verification challenge by Joshi and Holzmann [17, 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 [19]. Here, we concentrate on demonstrating the advantages of local directory tree reasoning compared with first-order global tree reasoning and reasoning about heap structures with paths as addresses.

A natural question is whether we might use first-order reasoning as in [8], rather than local reasoning in the style of separation logic. For our program-logic application, first-order reasoning leads to scalability problems. Consider one case of a first-order specification of the `rename(p/a, p'/b)` command:

[2] Previous work on segment logic [14] was too complicated, because we needed views.

[3] 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.

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

In the precondition, the assertion $resolve(p, d[t + a[t']])$ states that path p resolves to the directory d containing the subdirectory a and list t of unknown entries. The assertion $resolve(p', d'[t'' \wedge \neg exists(b)])$ states that path p' resolves to directory d' with no b entry. Finally, the assertion $\neg\exists p''.\ p' = p/a/p''$ is a path constraint, stating that path p' cannot be a descendant of p/a which would be an error case in POSIX. In the postcondition, the assertions state that the directory a has gone from d, and a new directory b has been created under directory d' with the contents of the old a.

Now consider program `rename(p/a, p'/b) ; rename(p''/c, p'''/d)`. In this case, we need path constraints in the precondition stating the following properties: path p' is not a descendant of path p/a; p''' is not a descendant of p''/c; and, in addition, p''/c is not a descendant of p/a since directory a has been removed. These syntactic path checks mushroom as more rename commands are added. Hence, this style of reasoning does not scale. Those familiar with separation logic might recognise that this example is analogous to Reynolds' original list example for justifying separation logic [21].

A completely different approach, used in much of the work on the formal specification of file systems in Z [18] and other methods [15], 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, in the specification of `rename(p/a, p/a')` not only we would replace p/a with p/a' in the heap, but also every descendant $p/a/p'$ with $p/a'/p'$ in order to preserve path consistency.

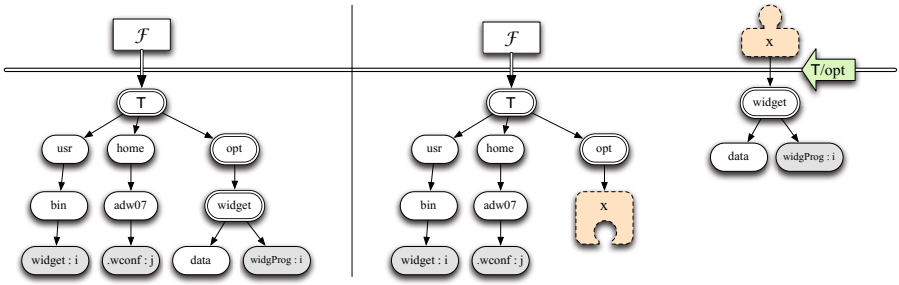

Fig. 1. The left-hand diagram represents a complete directory; the right, the same directory instrumented with abstract addresses

2 Example Specifications

We focus on the sequential POSIX file system in this paper, in particular studying a core fragment of 16 commands. Although this fragment is small, it includes most of the primitive commands that manipulate the structure and perform

input-output (IO) and a large proportion of the file system commands can be implemented using them.

First consider the English description of the `rmdir` command[4]:

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

Intuitively, this command traverses the *global* structure, using the global path `path` to identify the location of the update. It then performs *local* update, removing the empty directory whilst leaving the rest of the file system unchanged.

We capture this combination of global traversal and local update using structural separation logic. Consider figure 1. The left-hand side illustrates part of a standard structured heap, consisting of a heap cell with address $\mathcal{F}$ 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 address* x whose value is a pair consisting of a *path promise* '$\top/opt$' and, in this case, a complete subdirectory. The path promise provides 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. We move from the left-hand to the right-hand view of the state using *abstract allocation* which creates a new abstract heap cell containing the subdirectory; the converse is *abstract deallocation*.

With structured separation logic (SSL), we can reason about such abstract heap cells. For example, the assertion $subdir(\alpha@\mathrm{P}, \mathrm{A}[\varnothing])$ describes the ownership of the abstract heap cell with address given by logical variable α and value given by the path promise P and empty subdirectory $\mathrm{A}[\varnothing]$. Using this assertion, we are able to provide an axiomatic specification of the `rmdir` command:

$$\{expr(\texttt{path}, \mathrm{P/A}) \wedge var(\texttt{r}, -) \star \mathcal{E} \star subdir(\alpha@\mathrm{P}, \mathrm{A}[\varnothing])\}$$
$$\texttt{r:=rmdir(path)}$$
$$\{var(\texttt{r}, 0) \star \mathcal{E} \star subdir(\alpha@\mathrm{P}, \varnothing)\}$$

In the precondition, the assertion for the abstract heap cell at α describes the subdirectory resource necessary for the command to succeed, plus the stable information that the subdirectory is to be found under global path P. In addition, the precondition contains assertions about path expressions and variables. The *expression assertion* $expr(\texttt{path}, \mathrm{P/A})$ is a pure assertion which 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. The *variable assertion* $var(\texttt{r}, -)$ states that the program variable `r` has some

[4] 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 result is to assign -1 to `r` and set the global variable `errno` to ENOENT. We give the full error specifications in our technical report [11]. In this paper we discuss such cases only when required by an example.

arbitrary value, and $\mathcal{E}$ describes the extra variable resource necessary for `path` to be evaluated. This follows the standard variables-as-resource approach [3]

The postcondition states that variable `r` now has value 0, whilst the abstract heap cell α is empty with the path promise P. 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 matches with a body address at the appropriate path promise. The additional variable resource predicate $\mathcal{E}$ is unchanged between the pre and postconditions.

The specification of `rmdir` 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 subdirectory being updated. It also describes the global information that only (incomplete) directories satisfying path promise P associated with α can be framed on. To illustrate this, consider the complete directory $dir(\mathcal{F}, \top[\mathrm{C}+\mathrm{D}[\mathrm{A}[\varnothing]]])$, the path $\mathrm{P} = \top/\mathrm{D}/\mathrm{A}$ and the proof derivation:

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

The initial precondition contains the assertion $dir(\mathcal{F}, \top[\mathrm{C}+\mathrm{D}[\mathrm{A}[\varnothing]]])$ describing a complete directory tree at the file-system root $\top$, with arbitrary contents captured by the logical variable C and a directory named D that contains the empty directory A. This precondition 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, existentially quantifying the abstract address α to ensure that the address is fresh. 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 the resource that `rmdir` does not need. We are now in a position to match `rmdir`'s precondition, where $\top/\mathrm{D}$ is P. After 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`.

Using SSL, we can formalise the English specification in a similar fashion, with the following small axiom:

$$\{\, expr(\texttt{path}, \mathrm{P}/\mathrm{A}) \wedge var(\texttt{r}, -) \star \mathcal{E} \star subdir(\alpha@_{\mathrm{P}}, \mathrm{A} : \mathrm{I}) \,\}$$
`r:=unlink(path)`
$$\{\, var(\texttt{r}, 0) \star \mathcal{E} \star subdir(\alpha@_{\mathrm{P}}, \varnothing) \,\}$$

In the precondition, $subdir(\alpha@\mathrm{P}, \mathrm{A} : \mathrm{I})$ states that a file named A is found at abstract cell address α at the end of path P. The file data is not included in the precondition, but can be found at file inode I. When the last link to a file is removed, the file will no longer be accessible by any path, and we assume garbage collection will remove any associated file data.

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:

$$\{\, expr(\mathtt{path}, \mathrm{P/A}) \wedge var(\mathtt{t}, -) \star \mathcal{E} \star subdir(\alpha@\mathrm{P}, \mathrm{A}[\beta]) \,\}$$
$$\mathtt{t:=stat(path)}$$
$$\{\, var(\mathtt{t}, \mathrm{D}) \star \mathcal{E} \star subdir(\alpha@\mathrm{P}, \mathrm{A}[\beta]) \,\}$$

Notice that the specification uses *body address* β in $\mathrm{A}[\beta]$ to specify that the content of A is not changed by the command. It does not need more detailed 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 so far are enough to implement the 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. Notice that the derived specification exactly matches the English description obtained from POSIX. Following the same process, we can use the core fragment of this paper to "discover" formal specifications of many more complex commands of POSIX.

$\{\, expr(\mathtt{path}, \mathrm{P/A}) \wedge var(\mathtt{r}, -) \star \mathcal{E} \star subdir(\alpha@\mathrm{P}, (\mathrm{A} : \mathrm{I} \vee \mathrm{A}[\varnothing])) \,\}$

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

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

```
  if t = F
```

$\{\, expr(\mathtt{path}, \mathrm{P/A}) \wedge var(\mathtt{r}, -) \star \mathcal{E} \star subdir(\alpha@\mathrm{P}, \mathrm{A} : \mathrm{I}) \,\}$

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

$\{\, var(\mathtt{r}, 0) \star \mathcal{E} \star subdir(\alpha@\mathrm{P}, \varnothing) \,\}$

```
  else if t = D
```

$\{\, expr(\mathtt{path}, \mathrm{P/A}) \wedge var(\mathtt{r}, -) \star \mathcal{E} \star subdir(\alpha@\mathrm{P}, \mathrm{A}[\varnothing]) \,\}$

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

$\{\, var(\mathtt{r}, 0) \star \mathcal{E} \star subdir(\alpha@\mathrm{P}, \varnothing) \,\}$

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

$\{\, var(\mathtt{r}, 0) \star var(\mathtt{t}, -) \star \mathcal{E} \star subdir(\alpha@\mathrm{P}, \varnothing) \,\}$

```
}
```

$\{\, var(\mathtt{r}, 0) \star subdir(\alpha@\mathrm{P}, \varnothing) \,\}$

Fig. 2. An implementation of `remove` and the derived specification

3 File System Specification

We provide an axiomatic specification of our sequential POSIX commands using SSL.

3.1 Abstract Program State

An abstract program state comprises: an abstract *file-system heap*, which represents the directory tree and associated files, as might intuitively reside on a hard disk; a *process heap*, which represents the computer memory during execution; and a *variable store*, which represents the values of program variables.

File-System Heaps. 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 $x, y, z, \cdots$, 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:*

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

where $\varnothing$ *is the empty list of entries,* $a : \iota$ *is a file link associating file name* $a \in \text{FNAMES}$ *with inode* $\iota \in \text{INODES}$*,* $a[ud]$ *is a directory named* a *containing unrooted abstract directory* ud*,* $+$ *is directory composition and* $x \in \text{ABSADDRS}$ *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 $\top \notin \text{FNAMES}$ *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, $d \in \text{DIRS}$*, is defined by* $\text{DIRS} \triangleq \text{UDIRS} \cup \text{RDIRS}$*. Each directory entry has a type* $\text{DETYPES} \triangleq \{\mathrm{F}, \mathrm{D}\}$*, where* F *denotes a hard link to a file and* D *a directory.*

Each body address can be replaced by entries via *context application.*

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

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

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

Many POSIX commands refer to entries in the file system tree by *absolute linear paths* through the directory tree. General paths (with .. and symbolic links) are complex but we should be able to handle general paths using a combination of promises and *obligations* discussed in the conclusions: the abstract address 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 $a \in \text{FNAMES}$ *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* $\text{ABSPATHS} = \{p/x \mid p \in \text{ABPATHS}, x \in \text{ABSADDRS}\}$*. The set of* paths*,* $p \in \text{PATHS}$*, is* $\text{PATHS} \triangleq \text{RELPATHS} \cup \text{ABPATHS} \cup \text{ABSPATHS}$.

The path resolution *function* $\text{resolve} : \text{PATHS} \times \text{DIRS} \rightharpoonup \text{DIRS}$ *is defined by:*

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

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*,* $pfs \in \text{PREFS}$*, is a function in the set*

$$(\{\mathcal{F}\} \rightharpoonup \{\epsilon\} \times \text{RDIRS}) \sqcup (\text{ABSADDRS} \xrightarrow{fin} \text{ABPATHS} \times \text{DIRS}) \sqcup (\text{INODES} \xrightarrow{fin} \text{BYTES})$$

Let $inodes(d)$ *denote the set of all inodes occurring in directory* d*. A pre-file-system-heap,* pfs*, is* complete *if:* $dom(pfs) \cap \text{ABSADDRS} = \varnothing$*;* $pfs(\mathcal{F}) = (\epsilon, rd)$*;* $addrs(rd) = \varnothing$*; and* $inodes(rd) \subseteq \mathbf{dom}(pfs)$[5].

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.

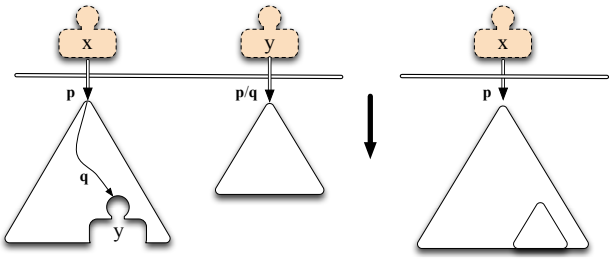

Fig. 3. Collapse relation

[5] 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.

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* $addr \in \textsc{AbsAddrs} \cup \{\mathcal{F}\}$ *and unique* $y \in \textsc{AbsAddrs}$ *such that:*

1. $pfs_1(addr) = (p, d)$ *and* $pfs_1(y) = (p_y, d_y)$*;*
2. $y \in addrs(d)$*;*
3. *there is some* $q \in \textsc{Paths}$ *such that* $p_y = p/q$*;*
4. $\text{resolve}(q, d) = y$*;*
5. $pfs_2 = pfs_1[addr \mapsto (p, d \circ_y d_y)]/y$[6]*.*

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 [25] 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 now define file-system heaps.

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

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

Process Heaps. 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. The `readdir` command returns the names of entries contained within a directory. POSIX allows a high degree of nondeterminism when using `readdir` on a directory whilst modifying its contents: one may see some changes; all changes; or none. Specifying the full behaviour is possible, but complex. To aid comprehension, we chose a *snapshot semantics*.

[6] That is, pfs_2 is equal to the function obtained from pfs_1 by removing y from the domain, mapping $addr$ to $(p, d \circ_y d_y)$, and leaving the other mappings the same.

Definition 7 (Process Heaps). *A process heap, denoted* $\mathrm{ph} \in \textsc{Ph}$*, is a partial function in the set* $(\textsc{DSAddrs} \overset{fin}{\rightharpoonup} \mathcal{P}(\textsc{FNames})) \sqcup (\textsc{OFAddrs} \overset{fin}{\rightharpoonup} (\textsc{Inodes} \times \mathbb{N}))$

Variable Stores. 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:

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

Definition 8 (Abstract Program States). *Given the sets of file-system heaps* FS*, process heaps* Ph *and variable stores* Σ*, the set of* abstract program states*, as* $\in \textsc{AStates}$*, is defined as:* $\textsc{AStates} \triangleq \mathrm{FS} \times \textsc{Ph} \times \Sigma$.

3.2 Programming Language

We define a standard imperative sequential WHILE language with calls to POSIX commands. *Program expressions* are used as the rvalue of assignments and as parameters to control-flow commands. They consist of the standard literals, variable lookup, arithmetic and boolean operations, and path concatenation Expr/Expr. Expression evaluation $[\![\cdot]\!]_\sigma : \textsc{Expr} \to \Sigma \rightharpoonup \textsc{Values}$ is mostly standard[7]. 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:*

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

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. We have formally specified all the commands of this fragment using SSL in [11]. Here, we present specifications for those commands that are used in our examples.

[7] Concatenation: $[\![\mathrm{Expr/Expr'}]\!]_\sigma \triangleq [\![\mathrm{Expr}]\!]_\sigma / [\![\mathrm{Expr'}]\!]_\sigma$ iff $[\![\mathrm{Expr'}]\!]_\sigma \notin \textsc{AbPaths}$.

3.3 Assertions

We describe assertions for reasoning about POSIX programs.[8] Analogous to programs using variables and expressions, assertions use logical variables and expressions. Logical variables are mapped to values by a logical environment, $e \in \text{LEnv}$, 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 $\alpha, \beta, \ldots$.

Cell Assertions		**Directory Assertions**	ϕ
Directory Tree	$\mathit{dir}(\mathcal{F}, \phi)$	Empty Entry	$\varnothing$
Subdirectory	$\mathit{subdir}(\alpha @_E, \phi)$	File Type Entry	$E : I$
File	$\mathit{file}(\mathrm{I}, E)$	Directory Type Entry	$E[\phi]$
File Descriptor	$\mathit{fd}(\mathrm{X}, \mathrm{I}, E)$	File System Root	$\top[\phi]$
Directory Stream	$\mathit{ds}(\mathrm{X}, E)$	Logical Expression	E
Heap	$\mathit{ptr}(E, E)$	Entry List	$\phi + \phi$
Variable	$\mathit{var}(\mathtt{var}, E)$	Context Application	$\phi \circ_\alpha \phi$
Expression	$\mathit{expr}(\text{Expr}, E)$	Path Resolution	$@E$

Fig. 4. Assertion language

Assertions, $P, Q \in \text{Asrts}$, are constructed from: the standard first-order logic connectives and quantifiers; the *separating conjunction* of separation logic, $P \star Q$, and its unit, emp; and the cell assertions of figure 4 which describe file-system heaps, process heaps and variable stores. Key is the subdirectory assertion, $\mathit{subdir}(\alpha @_E, \phi)$, 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 axioms, discussed in Section 3.4.

The file assertion, $\mathit{file}(\mathrm{I}, E)$, 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 $\mathit{var}(\mathtt{var}, E)$, describes program variable `var` with its value given by the logical expression E. Our core program commands accept parameters given by program expressions. The pure assertion $\mathit{expr}(\text{Expr}, E)$ states that the program expression Expr evaluates to the value of the logical expression E. The evaluation requires that we own all the variables used in the expression. Since in an arbitrary expression the variables are unknown, we will

[8] We have been asked whether ramified separation logic for reasoning about dags might be worth exploring [16]. 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.

typically use this assertion in conjunction with an *exact* assertion $\mathcal{E}$, leading to assertions of the form $expr(\mathrm{Expr}, E) \wedge \mathcal{E}$, where $\mathcal{E}$ captures all the variable resource required to evaluate Expr.

Directory assertions, $\phi, \psi \in \textsc{DirAsrts}$, are constructed from the standard first-order connectives and quantifiers, and the directory 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). *Context application*, $\phi \circ_\alpha \psi$, taken from context logic, describes a directory tree that can be separated into an partial directory satisfying ϕ, with abstract body address α bound in the assertion, 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:*

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

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 $top(\phi)$ states that the directory entries satisfy ϕ, and that no sibling entries have been split away. The assertion $can\_create(\mathrm{A})$ 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, $names(\mathrm{S})$ states that every name in the set S is present as an entry.

3.4 Program Logic

We describe our program logic for reasoning about our core fragment of sequential POSIX, comprising standard rules from separation logic, axioms for specifying the POSIX commands (Figure 5), and *abstract allocation and deallocation* axioms 11. The abstract allocation and deallocation axioms 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 the small axioms in Figure 5 to be used whenever required.

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

$$\big\{ subdir\big(\alpha_{@\mathrm{P}},\, (\phi_1 \wedge @_\mathrm{Q}/\beta) \circ_\beta \phi_2\big)\big\}$$
$$id$$
$$\big\{\exists\gamma.\, \big(subdir\,(\alpha_{@\mathrm{P}}, (\phi_1 \wedge @_\mathrm{Q}/\beta) \circ_\beta \gamma) \star subdir\,(\gamma_{@\mathrm{P}/\mathrm{Q}}, \phi_2)\big)\big\}$$

$$\left\{\exists\gamma.\left(\mathit{subdir}\left(\alpha@\mathrm{P}, (\phi_1 \wedge @\mathrm{Q}/\beta) \circ_\beta \gamma\right) \star \mathit{subdir}\left(\gamma@\mathrm{P/Q}, \phi_2\right)\right)\right\}$$
$$\mathit{id}$$
$$\left\{\mathit{subdir}\left(\alpha@\mathrm{P},\ (\phi_1 \wedge @\mathrm{Q}/\beta) \circ_\beta \phi_2\right)\right\}$$

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 an application of two separate parts: the context directory described by ϕ_1 which contains a relative path Q ending in body address β, applied via β to the subdirectory described by ϕ_2. The postcondition states that directory really can be separated into its two subparts: the subdirectory satisfying ϕ_2 is "allocated" into its own abstract heap cell γ whose corresponding body address is at absolute path P/Q; and the context directory at α satisfying ϕ_1 with γ replacing β. *Abstract deallocation* 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 referring 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.

Figure 5 provides the axioms for specifying the commands used in our software installer example, plus the axioms for `rename` as it is the most challenging command. The complete set of axioms is given in [11]. 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 that the subdirectories described by the preconditions contain no abstract body addresses. Commands such as `rename` and `stat` have multiple axioms, each covering a different behaviour specified in POSIX depending on the precondition state[9].

Consider the `mkdir(path)` command. According to its POSIX description, it creates a new empty directory identified by `path`. An existing entry with the same name must not already exist. In our precondition, `path` evaluates to a path of the form P/B/A. The subdirectory assertion $\mathit{subdir}\left(\alpha@\mathrm{P}, \mathrm{B}[\mathrm{C} \wedge \mathit{can\_create}(\mathrm{A})]\right)$ states that the subdirectory 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. In the postcondition, the assertion $\mathit{subdir}\left(\alpha@\mathrm{P}, \mathrm{B}[\mathrm{C} + \mathrm{A}[\varnothing]]\right)$ states that the empty directory A has indeed been 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 $\top$.

Now 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`. In the precondition, it has two subdirectory assertions, one

[9] The preconditions in such cases are mutually exclusive.

$$\left\{\begin{array}{c} expr(\texttt{path}, \mathrm{P/B/A}) \wedge var(\texttt{r}, -) \star \mathcal{E} \\ \star\ subdir(\alpha_{@\mathrm{P}}, \mathrm{B}[\mathrm{C} \wedge can\_create(\mathrm{A})]) \end{array}\right\}$$
$$\texttt{r := mkdir(path)}$$
$$\left\{ var(\texttt{r}, 0) \star \mathcal{E} \star subdir(\alpha_{@\mathrm{P}}, \mathrm{B}[\mathrm{C} + \mathrm{A}[\varnothing]]) \right\}$$

$$\left\{\begin{array}{c} expr(\texttt{path}, \mathrm{P/A}) \wedge var(\texttt{r}, -) \\ \star\ \mathcal{E} \star subdir(\alpha_{@\mathrm{P}}, \mathrm{A}[\varnothing]) \end{array}\right\}$$
$$\texttt{r := rmdir(path)}$$
$$\left\{ var(\texttt{r}, 0) \star \mathcal{E} \star subdir(\alpha_{@\mathrm{P}}, \varnothing) \right\}$$

$$\left\{\begin{array}{c} expr(\texttt{existing}, \mathrm{P/A}) \wedge expr(\texttt{new}, \mathrm{P'/D/B}) \\ \wedge\ var(\texttt{r}, -) \star \mathcal{E} \star subdir(\alpha_{@\mathrm{P}}, \mathrm{A} : \mathrm{I}) \\ \star\ subdir(\beta_{@\mathrm{P'}}, \mathrm{D}[\mathrm{C} \wedge can\_create(B)]) \end{array}\right\}$$
$$\texttt{r := link(existing, new)}$$
$$\left\{\begin{array}{c} var(\texttt{r}, 0) \star \mathcal{E} \star subdir(\alpha_{@\mathrm{P}}, \mathrm{A} : \mathrm{I}) \\ \star\ subdir(\beta_{@\mathrm{P'}}, \mathrm{D}[\mathrm{C} + \mathrm{B} : \mathrm{I}]) \end{array}\right\}$$

$$\left\{\begin{array}{c} expr(\texttt{existing}, \mathrm{P/D/A}) \\ \wedge\ expr(\texttt{new}, \mathrm{P/D/B}) \wedge var(\texttt{r}, -) \star \mathcal{E} \\ \star\ subdir(\alpha_{@\mathrm{P}}, \mathrm{D}[(\mathrm{C} + \mathrm{A} : \mathrm{I}) \wedge can\_create(\mathrm{B})]) \end{array}\right\}$$
$$\texttt{r := link(existing, new)}$$
$$\left\{ var(\texttt{r}, 0) \star \mathcal{E} \star subdir(\alpha_{@\mathrm{P}}, \mathrm{D}[\mathrm{C} + \mathrm{A} : \mathrm{I} + \mathrm{B} : \mathrm{I}]) \right\}$$

$$\left\{\begin{array}{c} expr(\texttt{path}, \mathrm{P/A}) \wedge var(\texttt{r}, -) \\ \star\ \mathcal{E} \star subdir(\alpha_{@\mathrm{P}}, \mathrm{A} : \mathrm{I}) \end{array}\right\}$$
$$\texttt{r := unlink(path)}$$
$$\left\{ var(\texttt{r}, 0) \star \mathcal{E} \star subdir(\alpha_{@\mathrm{P}}, \varnothing) \right\}$$

$$\left\{\begin{array}{c} expr(\texttt{old}, \mathrm{P/A}) \wedge expr(\texttt{new}, \mathrm{P'/D/B}) \\ \wedge var(\texttt{r}, -) \star \mathcal{E} \star subdir(\alpha_{@\mathrm{P}}, \mathrm{A}[\mathrm{C} \wedge complete]) \\ \star\ subdir\left(\beta_{@\mathrm{P'}}, \mathrm{D}\left[\mathrm{C'} \wedge can\_create(\mathrm{B})\right]\right) \end{array}\right\}$$
$$\texttt{r := rename(old, new)}$$
$$\left\{\begin{array}{c} var(\texttt{r}, 0) \star \mathcal{E} \star subdir(\alpha_{@\mathrm{P}}, \varnothing) \\ \star\ subdir\left(\beta_{@\mathrm{P'}}, \mathrm{D}[\mathrm{C'} + \mathrm{B}[\mathrm{C}]]\right) \end{array}\right\}$$

$$\left\{\begin{array}{c} expr(\texttt{old}, \mathrm{P/D/A}) \wedge expr(\texttt{new}, \mathrm{P/D/B}) \\ \wedge\ var(\texttt{r}, -) \star \mathcal{E} \\ \star\ subdir\left(\alpha_{@\mathrm{P}}, \mathrm{D}\left[\begin{array}{c}(\mathrm{C} + \mathrm{A}\left[\mathrm{C'} \wedge complete\right]) \\ \wedge\ can\_create(B)\end{array}\right]\right) \end{array}\right\}$$
$$\texttt{r := rename(old, new)}$$
$$\left\{ var(\texttt{r}, 0) \star \mathcal{E} \star subdir\left(\alpha_{@\mathrm{P}}, \mathrm{D}[\mathrm{C} + \mathrm{B}[\mathrm{C'}]]\right) \right\}$$

$$\left\{\begin{array}{c} expr(\texttt{old}, \mathrm{P/A}) \wedge expr(\texttt{new}, \mathrm{P'/B}) \\ \wedge\ var(\texttt{r}, -) \star \mathcal{E} \\ \star\ subdir(\alpha_{@\mathrm{P}}, \mathrm{A}[\mathrm{C} \wedge complete]) \\ \star\ subdir(\beta_{@\mathrm{P'}}, \mathrm{B}[\varnothing]) \end{array}\right\}$$
$$\texttt{r := rename(old, new)}$$
$$\left\{\begin{array}{c} var(\texttt{r}, 0) \star \mathcal{E} \star subdir(\alpha_{@\mathrm{P}}, \varnothing) \\ \star\ subdir(\beta_{@\mathrm{P'}}, \mathrm{B}[\mathrm{C}]) \end{array}\right\}$$

$$\left\{\begin{array}{c} expr(\texttt{old}, \mathrm{P/A}) \wedge expr(\texttt{new}, \mathrm{P'/B}) \\ \wedge\ var(\texttt{r}, -) \star \mathcal{E} \star subdir(\alpha_{@\mathrm{P}}, \mathrm{A} : \mathrm{I}) \\ \star\ subdir\left(\beta_{@\mathrm{P'}}, \mathrm{B} : \mathrm{I'}\right) \end{array}\right\}$$
$$\texttt{r := rename(old, new)}$$
$$\left\{\begin{array}{c} var(\texttt{r}, 0) \star \mathcal{E} \star subdir(\alpha_{@\mathrm{P}}, \varnothing) \\ \star\ subdir(\beta_{@\mathrm{P'}}, \mathrm{B} : \mathrm{I}) \end{array}\right\}$$

$$\left\{\begin{array}{c} expr(\texttt{old}, \mathrm{P/D/A}) \wedge expr(\texttt{new}, \mathrm{P/D/B}) \\ \wedge\ var(\texttt{r}, -) \star \mathcal{E} \\ \star\ subdir\left(\alpha_{@\mathrm{P}}, \mathrm{D}\left[\begin{array}{c}(\mathrm{C} + \mathrm{A} : \mathrm{I}) \\ \wedge\ can\_create(\mathrm{B})\end{array}\right]\right) \end{array}\right\}$$
$$\texttt{r := rename(old, new)}$$
$$\left\{\begin{array}{c} var(\texttt{r}, 0) \star \mathcal{E} \\ \star\ subdir(\alpha_{@\mathrm{P}}, \mathrm{D}[\mathrm{C} + \mathrm{B} : \mathrm{I}]) \end{array}\right\}$$

$$\left\{\begin{array}{c} expr(\texttt{old}, \mathrm{P/A}) \wedge expr(\texttt{new}, \mathrm{P'/D/B}) \\ \wedge\ var(\texttt{r}, -) \star \mathcal{E} \star subdir(\alpha_{@\mathrm{P}}, \mathrm{A} : \mathrm{I}) \\ \star\ subdir(\beta_{@\mathrm{P'}}, \mathrm{D}[\mathrm{C} \wedge can\_create(\mathrm{B})]) \end{array}\right\}$$
$$\texttt{r := rename(old, new)}$$
$$\left\{\begin{array}{c} var(\texttt{r}, 0) \star \mathcal{E} \star subdir(\alpha_{@\mathrm{P}}, \varnothing) \\ \star\ subdir(\beta_{@\mathrm{P'}}, \mathrm{D}[\mathrm{C} + \mathrm{B} : \mathrm{I}]) \end{array}\right\}$$

$$\left\{\begin{array}{c} expr(\texttt{old}, \mathrm{P/A}) \wedge expr(\texttt{new}, \mathrm{P/A}) \\ \wedge\ var(\texttt{r}, -) \star \mathcal{E} \\ \star\ subdir(\alpha_{@\mathrm{P}}, \mathrm{C} \wedge entry(\mathrm{A})) \end{array}\right\}$$
$$\texttt{r := rename(old, new)}$$
$$\left\{ var(\texttt{r}, 0) \star \mathcal{E} \star subdir(\alpha_{@\mathrm{P}}, \mathrm{C}) \right\}$$

$$\left\{\begin{array}{c} expr(\texttt{path}, \mathrm{P/A}) \wedge var(\texttt{t}, -) \star \mathcal{E} \\ \star\ subdir(\alpha_{@\mathrm{P}}, \mathrm{A}[\beta]) \end{array}\right\}$$
$$\texttt{t := stat(path)}$$
$$\left\{ var(\texttt{t}, \mathrm{D}) \star \mathcal{E} \star subdir(\alpha_{@\mathrm{P}}, \mathrm{A}[\beta]) \right\}$$

$$\left\{\begin{array}{c} expr(\texttt{path}, \mathrm{P/A}) \wedge var(\texttt{t}, -) \\ \star\ \mathcal{E} \star subdir(\alpha_{@\mathrm{P}}, \mathrm{A} : \mathrm{I}) \end{array}\right\}$$
$$\texttt{t := stat(path)}$$
$$\left\{ var(\texttt{t}, \mathrm{F}) \star \mathcal{E} \star subdir(\alpha_{@\mathrm{P}}, \mathrm{A} : \mathrm{I}) \right\}$$

$$\left\{\begin{array}{c} expr(\texttt{path}, \mathrm{P/D}) \wedge var(\texttt{dir}, -) \star \mathcal{E} \\ \star\ subdir(\alpha_{@\mathrm{P}}, \mathrm{D}[top(\mathrm{C})]) \end{array}\right\}$$
$$\texttt{dir := opendir(path)}$$
$$\left\{\begin{array}{c} \exists \mathrm{H}.\ var(\texttt{dir}, \mathrm{H}) \star \mathcal{E} \\ \star\ subdir(\alpha_{@\mathrm{P}}, \mathrm{D}[\mathrm{C} \wedge names(\mathrm{A})]) \\ \star\ ds(\mathrm{H}, \mathrm{A}) \end{array}\right\}$$

$$\left\{\begin{array}{c} var(\texttt{dir}, \mathrm{H}) \star var(\texttt{fn}, -) \\ \star\ ds(\mathrm{H}, \mathrm{A}) \wedge \mathrm{A} \neq \{\} \end{array}\right\}$$
$$\texttt{fn := readdir(dir)}$$
$$\left\{\begin{array}{c} var(\texttt{fn}, \mathrm{B}) \star var(\texttt{dir}, \mathrm{H}) \\ \star\ ds(\mathrm{H}, (\mathrm{A} \setminus \{\ \mathrm{B}\})) \wedge \mathrm{B} \in \mathrm{A} \end{array}\right\}$$

$$\left\{\begin{array}{c} var(\texttt{dir}, \mathrm{H}) \star var(\texttt{fn}, -) \\ \star\ ds(\mathrm{H}, \{\}) \end{array}\right\}$$
$$\texttt{fn := readdir(dir)}$$
$$\left\{\begin{array}{c} var(\texttt{dir}, \mathrm{H}) \star var(\texttt{fn}, \epsilon) \\ \star\ ds(\mathrm{H}, \{\}) \end{array}\right\}$$

$$\left\{ var(\texttt{dir}, \mathrm{H}) \star ds(\mathrm{H}, \mathrm{A}) \right\}$$
$$\texttt{closedir(dir)}$$
$$\left\{ var(\texttt{dir}, \mathrm{H}) \right\}$$

Fig. 5. Axioms for some POSIX commands

for each path. The assertion $\mathit{subdir}(\alpha@\mathrm{P}, \mathrm{A} : \mathrm{I})$ states that the `existing` path P/A identifies a file link named A to the file with inode address I. The assertion $\mathit{subdir}(\beta@\mathrm{P}', \mathrm{D}[\mathrm{C} \wedge \mathit{can\_create}(B)])$ states, as in `mkdir`, that an entry with the name we want to create does not already exist. In the postcondition, the assertion $\mathit{subdir}(\beta@\mathrm{P}', \mathrm{D}[\mathrm{C} + \mathrm{B} : \mathrm{I}])$ states that this new entry has been created with another file link to the same file. Note that, in this first axiom, the update takes place between two different directories; in the second, they take place within the same directory.

The `rename(old,new)` command moves and/or renames the entry identified by the path `old` to that identified by `new`. Consider the first axiom where `old` is a directory and `new` does not exist. In the precondition, the assertion $\mathit{subdir}(\alpha@\mathrm{P}, \mathrm{A}[\mathrm{C} \wedge \mathit{complete}])$ states that the subdirectory A must be complete and the assertion $\mathit{subdir}\big(\beta@\mathrm{P}', \mathrm{D}\big[\mathrm{C}' \wedge \mathit{can\_create}(\mathrm{B})\big]\big)$ states that it is possible to create the new directory B under D. The only path constraints are that the global paths P and P′ must exist in the underlying global directory, thus restricting the application of the frame rule. In contrast to the first-order `rename` axiom discussed in related work, we do not require any additional path constraints to ensure that P′ is not a descendant of P/A. It comes automatically from the separating conjunction.

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, the assertion $\mathit{top}(\mathrm{C})$ (definition 10) states that the entries of the identified subdirectory D are complete at the top level. In the postcondition, the assertion $\mathit{subdir}(\alpha@\mathrm{P}, \mathrm{D}[\mathrm{C} \wedge \mathit{names}(\mathrm{A})])$ declares the set A of all the entries of the directory D, and uses it in the assertion $\mathit{ds}(\mathrm{H}, \mathrm{A})$ to describe the allocated directory stream at H. 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 the order of directory entries is implementation defined in POSIX. The `closedir(dir)` command simply deallocates the directory stream given by `dir`.

Sequential 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 [11], we give a standard soundness result for sequential programs (no external processes modifying the file system), providing an operational semantics and proving soundness in the style of the views framework [6].

4 Software Installer

We now demonstrate our reasoning by considering a *software installer*. New 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 place the bundle's contents correctly in the users' file system which may involve other tasks such as removing any previous installations and dealing with incompatible

user files. Installers are a common class of client programs that perform complex manipulation of file system structure.

Here, we develop an installer for the fictional software "Widget v2". It supersedes "Widget v1", but is incompatible with any v1 user configuration files. Widget v2 consists of a program executable, '*widgProg*' and a data file, '*widgData*'. Following common conventions [1], we place the files in '$\top/opt/widget/$' and create a link from '$\top/usr/bin/widget$' to '$\top/opt/widget/widgProg$'. An example situation the installer may encounter is that in figure 1, where 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 a path does not exist. We consider error specifications for the entire subset in [11]. Here, we discuss only the ENOENT error for `stat`, triggered when a path does not resolve to a file or directory. To describe a file system in which a path cannot resolve, we define the following:

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

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`:

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

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, but `rmdir` only removes empty directories. We can implement a program `rmdirRec` that recursively removes all the directories entries before removing the directory itself. The specification is:

$$\{\ \mathit{expr}(\texttt{path}, \mathrm{P}/\mathrm{A}) \wedge \mathit{var}(\texttt{r}, -) \star \mathcal{E} \star \mathit{subdir}(\alpha_{@\mathrm{P}}, \mathrm{A}[\mathit{complete}])\ \}$$
$$\texttt{r := rmdirRec(path)}$$
$$\{\ \mathit{var}(\texttt{r}, 0) \star \mathcal{E} \star \mathit{subdir}(\alpha_{@\mathrm{P}}, \varnothing)\ \}$$

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

$$\left\{\begin{array}{c} \mathit{expr}(\texttt{source}, \mathrm{P/A}) \wedge \mathit{expr}(\texttt{target}, \mathrm{P'/D}) \wedge \mathit{var}(\mathtt{r}, -) \star \mathcal{E} \\ \star\ \mathit{subdir}(\alpha_{@\mathrm{P}}, \mathrm{A : I}) \star \mathit{subdir}(\beta_{@\mathrm{P'}}, \mathrm{D}[\mathrm{C} \wedge \mathit{can\_create}(\mathrm{A})]) \star \mathit{file}(\mathrm{I}, \mathrm{SD}) \end{array}\right\}$$

```
r := fileCopy(source, target)
```

$$\left\{\begin{array}{c} \exists \mathrm{I'}.\ \mathit{var}(\mathtt{r}, 0) \star \mathcal{E} \star \mathit{subdir}(\alpha_{@\mathrm{P}}, \mathrm{A : I}) \star \mathit{subdir}(\beta_{@\mathrm{P'}}, \mathrm{D}[\mathrm{C} + \mathrm{A : I'}]) \\ \star\ \mathit{file}(\mathrm{I}, \mathrm{SD}) \star \mathit{file}(\mathrm{I'}, \mathrm{SD}) \end{array}\right\}$$

We have implemented both `rmdirRec` and `fileCopy` and derived their specifications in [11].

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:

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

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 $+_{\mathrm{X} \in \mathrm{E}}\, \phi$ 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.

$$\mathit{src}_{\mathrm{Pre}} \triangleq \mathit{subdir}(\delta_{@\mathrm{IL}}, \mathrm{widgProg : J} + \mathrm{widgData : K}) \star \mathit{file}(\mathrm{J}, \mathrm{PROG}) \star \mathit{file}(\mathrm{K}, \mathrm{DAT})$$
$$\mathit{home}_{\mathrm{Pre}} \triangleq \mathit{subdir}\big(\alpha_{@\top}, \mathrm{home}\,[+_{(\mathrm{N},\mathrm{C}) \in \mathrm{H}}\, \mathrm{N}\,[\mathit{infile}(\mathrm{C}, .\mathrm{wconf}) \vee \mathit{out}(\mathrm{C}, .\mathrm{wconf})]]\big)$$
$$\mathit{bin}_{\mathrm{Pre}} \triangleq \mathit{subdir}(\gamma_{@\top/usr}, \mathrm{bin}\,[\mathit{in}(\mathrm{B}, \mathrm{widget}) \vee \mathit{out}(\mathrm{B}, \mathrm{widget})])$$
$$\mathit{opt}_{\mathrm{Pre}} \triangleq \mathit{subdir}(\beta_{@\top}, \mathrm{opt}\,[\mathit{top}(\mathrm{T}) + (\varnothing \vee \mathrm{widget}[\mathrm{T}_w \wedge \mathit{complete}]])$$

Each of these describes the states that parts of the file system may be in for the installer to run safely. The directory entries and file data that make up the Widget v2 installation sources are described by $\mathit{src}_{\mathrm{Pre}}$. We require them to be in a location given by the variable `il`. $\mathit{home}_{\mathrm{Pre}}$ captures all the home directories of the system, along with the fact that some of them will contain a 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.

$$P_i \triangleq \mathit{var}(\mathtt{il}, \mathrm{IL}) \star \mathit{var}(\mathtt{r}, -) \star \mathit{var}(\mathtt{errno}, -) \wedge \mathrm{W} \wedge \mathit{src}_{\mathrm{Pre}} \star \mathit{home}_{\mathrm{Pre}} \star \mathit{bin}_{\mathrm{Pre}} \star \mathit{opt}_{\mathrm{Pre}}$$

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:

$\{ P_i \}$

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

$\{ var(\texttt{t1}, -) \star var(\texttt{t2}, -) \star var(\texttt{errno}, -) \star bin_{\mathrm{Pre}} \star opt_{\mathrm{Pre}} \}$

```
  // 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');
```

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

```
  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.
```

$\{ var(\texttt{r}, -) \star var(\texttt{t1}, \mathrm{F}) \vee var(\texttt{t1}, -1) \star var(\texttt{t2}, \mathrm{D}) \vee var(\texttt{t2}, -1) \star bin_{\mathrm{Pre}} \star opt_{\mathrm{Pre}} \}$

```
    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.
```

$\{ var(\texttt{t2}, \mathrm{D}) \star var(\texttt{r}, -) \star subdir(\beta@_{\mathsf{T}}, \mathrm{opt}[\mathrm{T} + \mathrm{widget}[\mathrm{T}_w \wedge complete]]) \}$

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

$\{ var(\texttt{t2}, \mathrm{D}) \star var(\texttt{r}, 0) \star subdir(\beta@_{\mathsf{T}}, \mathrm{opt}[\mathrm{T}]) \}$

$\left\{ \begin{array}{l} var(\texttt{r}, 0) \star var(\texttt{t1}, \mathrm{F}) \vee var(\texttt{t1}, -1) \star var(\texttt{t2}, \mathrm{D}) \vee var(\texttt{t2}, -1) \\ \star\ subdir(\gamma@_{\mathsf{T}/usr}, \mathrm{bin}[\mathrm{B}]) \star subdir(\beta@_{\mathsf{T}}, \mathrm{opt}[\mathrm{T}]) \end{array} \right\}$

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

$\{ var(\texttt{hDir}, -) \star var(\texttt{user}, -) \star subdir(\alpha@_{\mathsf{T}}, \mathrm{home}[+_{(\mathrm{N},\mathrm{C}) \in \mathrm{H}} \mathrm{N}[infile(\mathrm{C}, .\mathrm{wconf}) \vee out(\mathrm{C}, .\mathrm{wconf})]]) \}$

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

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

```
  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.
```

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

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

$\left\{ \begin{array}{l} var(\texttt{r}, -) \star var(\texttt{il}, \mathrm{IL}) \star subdir(\delta@_{\mathrm{IL}}, v2Dir_{\mathrm{Pre}}) \star subdir(\alpha@_{\mathsf{T}}, home_{\mathrm{Post}}) \\ \star\ subdir(\gamma@_{\mathsf{T}/usr}, \mathrm{bin}[\mathrm{B}]) \star subdir(\beta@_{\mathsf{T}}, \mathrm{opt}[\mathrm{T}]) \star file(\mathrm{J}, \textsc{Prog}) \star file(\mathrm{K}, \textsc{Dat}) \end{array} \right\}$

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

$\left\{ \begin{array}{c} \exists \mathrm{J'}, \mathrm{K'}.\ var(\texttt{r}, 0) \star var(\texttt{il}, \mathrm{IL}) \star src_{\mathrm{Pre}} \star home_{\mathrm{Post}} \\ \star\ subdir(\beta@_{\mathsf{T}}, \mathrm{opt}[\mathrm{T} + \mathrm{widget}[\mathrm{widgProg} : \mathrm{J'} + \mathrm{widgData} : \mathrm{K'}]]) \\ \star\ subdir(\gamma@_{\mathsf{T}/usr}, \mathrm{bin}[\mathrm{B} + \mathrm{widget} : \mathrm{J'}]) \star file(\mathrm{J'}, \textsc{Prog}) \star file(\mathrm{K'}, \textsc{Dat}) \end{array} \right\}$

```
}
```

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

$\{ Q_i \}$

Fig. 6. Widget v2 software installer

$$
\begin{aligned}
v2Files_{\mathrm{Post}}(\mathrm{J},\mathrm{K}) &\triangleq \mathit{file}(\mathrm{J},\textsc{Prog}) \star \mathit{file}(\mathrm{K},\textsc{Dat})\\
\mathit{home}_{\mathrm{Post}} &\triangleq \mathit{subdir}\big(\alpha_{@\top},\mathrm{home}\big[+_{(\mathrm{N},\mathrm{C})\in\mathrm{H}}\,\mathrm{N}\,[\mathrm{C}\wedge \mathit{can\_create}(.\mathrm{wconf})]\big]\big)\\
\mathit{bin}_{\mathrm{Post}}(\mathrm{J}) &\triangleq \mathit{subdir}\,(\gamma_{@\top/usr},\mathrm{bin}[\mathrm{B}+\mathrm{widget}:\mathrm{J}])\\
\mathit{opt}_{\mathrm{Post}}(\mathrm{J},\mathrm{K}) &\triangleq \mathit{subdir}\,(\beta_{@\top},\mathrm{opt}[\mathrm{T}+\mathrm{widget}[\mathrm{widgProg}:\mathrm{J}+\mathrm{widgDat}:\mathrm{K}]])
\end{aligned}
$$

The postcondition is built from these sub-assertions.

$$
\begin{aligned}
Q_i \triangleq\ & \exists \mathrm{R},\mathrm{J}',\mathrm{K}'.\ \mathit{var}(\mathtt{il},\textsc{Il}) \star \mathit{var}(\mathtt{r},\mathrm{R}) \star \mathit{var}(\mathtt{errno},-) \wedge (\mathrm{R}=-1 \Rightarrow \mathrm{W})\\
& \wedge\ \mathrm{R}=0 \Rightarrow \big(\ \mathit{src}_{\mathrm{Pre}} \star \mathit{home}_{\mathrm{Post}} \star \mathit{opt}_{\mathrm{Post}}(\mathrm{J}',\mathrm{K}') \star \mathit{bin}_{\mathrm{Post}}(\mathrm{J}') \star v2Files_{\mathrm{Post}}(\mathrm{J}',\mathrm{K}')\ \big)
\end{aligned}
$$

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, $\{P_i\}\texttt{installWidgetV2}\{Q_i\}$, 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.

5 Conclusions and Future Work

The POSIX file system provides an interesting challenge for local reasoning: complex abstract data update with global absolute paths for identifying the place to do local update. We give a natural axiomatic specification of the sequential POSIX file system using SSL; the general theory is in [25, 12]. We verify safety properties for a client software installer, demonstrating integrated reasoning for the file system and the heap. Our POSIX reasoning provides an illustrative example of reasoning about global access and local update; other natural applications include identifying the ith element of a list [25] and querying the 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.

Acknowledgements. We acknowledge funding from an EPSRC DTA (Ntzik, Wright) and EPSRC programme grant EP/H008373/1 (Gardner, Ntzik and Wright). We also thank Pedro da Rocha Pinto, Ramana Kumar, Azalea Raad, Tom Ridge and Mark Wheelhouse for many interesting discussions.

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] Arkoudas, K., Zee, K., Kuncak, V., Rinard, M.: Verifying a File System Implementation. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 373–390. Springer, Heidelberg (2004)
[5] Calcagno, C., Gardner, P., Zarfaty, U.: Context logic and tree update. SIGPLAN Not. (2005)
[6] Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M., Yang, H.: Views: compositional reasoning for concurrent programs. In: POPL (2013)
[7] Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent abstract predicates. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 504–528. Springer, Heidelberg (2010)
[8] Fisher, K., Foster, N., Walker, D., Zhu, K.Q.: Forest: a language and toolkit for programming with filestores. In: ICFP (2011)
[9] Freitas, L., Fu, Z., Woodcock, J.: POSIX file store in Z/Eves: an experiment in the verified software repository. In: IEEE International Conference on Engineering of Complex Computer Systems (2007)
[10] Freitas, L., Woodcock, J., Butterfield, A.: POSIX and the verification grand challenge: A roadmap. In: ICECCS (2008)
[11] Gardner, P., Ntzik, G., Wright, A.: Local Reasoning for the POSIX File System. Technical report, Imperial College London (2014), `http://www.doc.ic.ac.uk/~gn408/POSIXFS/`
[12] Gardner, P., Raad, A., Wheelhouse, M., Wright, A.: Abstract Local Reasoning for Concurrent Libraries. In preparation (2014)
[13] Gardner, P., Smith, G., Wheelhouse, M., Zarfaty, U.: Local Hoare reasoning about DOM. In: PODS (2008)
[14] Gardner, P., Wheelhouse, M.: Small specifications for tree update, `http://www.doc.ic.ac.uk/~pg/papers/move.pdf`
[15] Hesselink, W.H., Lali, M.: Formalizing a hierarchical file system. In: REFINE (2009)
[16] Hobor, A., Villard, J.: The ramifications of sharing in data structures. In: POPL (2013)
[17] Joshi, R., Holzmann, G.J.: A mini challenge: build a verifiable filesystem. Form. Asp. Comput. (2007)
[18] Morgan, C., Sufrin, B.: Specification of the UNIX Filing System. IEEE Transactions on Software Engineering (1984)
[19] Ntzik, G.: Local Reasoning about File Systems. PhD thesis (expected, 2014)
[20] Parkinson, M., Bierman, G.: Separation logic and abstraction. In: POPL (2005)
[21] Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS (2002)
[22] Smith, G.: Local Reasoning about Web Programs. PhD thesis (2011)
[23] Svendsen, K., Birkedal, L.: Impredicative concurrent abstract predicates. In: Shao, Z. (ed.) ESOP 2014. LNCS, vol. 8410, pp. 149–168. Springer, Heidelberg (2014)
[24] Turon, A., Dreyer, D., Birkedal, L.: Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In: ICFP (2013)
[25] Wright, A.: Structural Separation Logic. PhD thesis, Imperial College London (2013)