Reasoning about the POSIX File System

Local Update and Global Pathnames

Gian Ntzik Philippa Gardner
Imperial College London, UK
{gian.ntzik08,p.gardner}@imperial.ac.uk

Abstract

We introduce a program logic for specifying a core sequential subset of the POSIX file system and for reasoning abstractly about client programs working with the file system. The challenge is to reason about the combination of *local* directory update and *global* pathname traversal (including '..' and symbolic links) which may overlap the directories being updated. Existing reasoning techniques are either based on first-order logic and do not scale, or on separation logic and can only handle linear pathnames (no '..' or symbolic links). We introduce *fusion logic* for reasoning about local update and global pathname traversal, introducing a novel *effect frame* rule to propagate the effect of a local update on overlapping pathnames. We apply our reasoning to the standard recursive remove utility (`rm -r`), discovering bugs in well-known implementations.

Categories and Subject Descriptors F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Logics of programs

Keywords POSIX, file systems, local reasoning, global pathnames, separation logic

1. Introduction

POSIX client programs manipulate the file system by using the POSIX programming interfaces [1] to update regular files or directories[1]. POSIX file system operations typically identify the entries to be updated using pathnames (paths). The specification of the POSIX file system and the modular verification of POSIX client programs provide a significant challenge to existing reasoning techniques. The update is *local*, in that it simply affects the identified entry, but the pathname resolution is *global*, in that it is able to traverse up and down the whole directory hierarchy. Existing reasoning techniques are either based on first-order logic which do not scale (see related work), or on separation logic which can only handle simple linear paths [3, 18]. We provide modular sequential reasoning about the POSIX file system based on separation logic [24], introducing new reasoning techniques to account for local update and global pathname resolution.

In the simple setting without '..' and symbolic links, we can use the *linear* pathname `/usr/lib/tex` to remove the directory `tex`. The local update, which removes the empty directory `tex` from `lib`, has no effect on the linear pathname `/usr/lib` which identifies its location; the path does not change and remains valid after update. This separation of the linear pathname and local update makes the reasoning comparatively straightforward. With arbitrary pathnames, there is no such separation and the reasoning is more difficult. Reasoning about arbitrary pathnames is essential for accurately specifying POSIX, since no primitive file system operation is restricted to linear pathnames. Reasoning about arbitrary pathnames is also essential for verifying client programs: standard utilities such as recursive remove accept arbitrary pathnames as input; Makefiles and admin scripts frequently use '..' to traverse a directory relative to some starting directory; and installers use symbolic links to reference different versions of files.

POSIX pathnames use '..' to traverse up the directory structure and symbolic links to jump between directories. They thus cut across the underlying inductive definition of the directory tree and overlap with the directory subtree(s) being updated. For example, we can use the path `/usr/lib/tex/../tex` to remove the directory `tex`. In this case, the update is not separate from the path `/usr/lib/tex/..` that identifies its location. The path is no longer valid after update. To complicate things further, a directory in `lib` which is different (disjoint) from `tex` can be identified by a path using it, for example

[1] POSIX defines many types of files, which include regular files and directories, and uses the term 'file' to refer to a file of any type. To avoid confusion, we use 'file' for a regular file and 'entry' for a file of any type.

`/usr/lib/tex/../latex`. Removing `tex`, will invalidate this path as well; the local update has a global effect.

When providing a sequential specification of POSIX, a fundamental question is whether to use global reasoning (e.g. using first-order logic) and add mechanisms to account for disjointness, or whether to start with local reasoning (e.g. using separation logic) and add mechanisms to account for sharing. There has been much work on traditional global reasoning techniques for specifying POSIX [2, 15, 19], such as the well-known Z specification [22] and Forest [14]. This work mainly showed that implementations were correct with respect to the POSIX specification. We have recently developed a local reasoning technique for specifying POSIX file systems restricted to linear pathnames [18]. In particular, we have mainly focused on how to reason about client programs and demonstrated that global reasoning does not provide scalable reasoning about client programs. We summarise this argument against global reasoning in the related work, and also argue that our previous local reasoning work does not extend to arbitrary pathnames.

We introduce *fusion logic*, providing a specification of a sequential core fragment of POSIX file systems with arbitrary pathnames and verifying file-system client programs. With fusion logic, we start from local reasoning and introduce new techniques to account for the sharing of pathnames which overlap the local update. In this paper, we concentrate on pathnames using '..', as this is enough to introduce the important features of the reasoning. We also summarise how to extend our reasoning to symbolic links, giving full details in the accompanying technical report [23].

With fusion logic, we reason precisely about disjoint local update, whilst at the same time accounting for overlapping pathnames which can become obsolete after the update. We achieve this by combining a specific permission algebra for describing the disjoint and shared structure, with a novel *effect separating conjunction* and *effect frame rule* which percolates the effect of the update through the frame. The permission structure provides the following sharing information and update capability: the standard *full permission* (value 1), used to describe disjoint entries with exclusive permission to update; Boyland-style *fractional permissions* [4, 6] (values in $(0, 1)$), used to describe shared entries which can be read, cannot be updated and, hence, cannot overlap with an updatable entry; and a non-standard *shadow* permission (value 0), used to describe shared entries which can be read as part of a pathname, cannot themselves be updated, but can overlap with an updatable entry.

The shadow permission provides accurate information about directory entries and pathnames *before* update. However, after update, the information may be out of date, as the pathname can become invalid. The local update of a directory entry can have a global effect on shared paths and needs to be propagated through the reasoning. This is achieved by an *effect separating conjunction* $\hat{*}$ and an *effect frame rule*:

$$\textsc{EffectFrame}\,\frac{\vdash \{P\}\mathbb{C}\{Q\}}{\vdash \{P * R\}\mathbb{C}\{Q \mathbin{\hat{*}} R\}}$$

The precondition of the conclusion of the effect frame is standard. The postcondition is unusual. The effect separating conjunction is only present in postconditions. It enables an update to be propagated to the frame using a set of *effect fusion* axioms to make R consistent with Q. After propagation, the resulting postcondition has no more effect separating conjunctions, and hence can be used as the precondition of another command. The effect frame rule enables us to reason locally. We can keep specifications small and consider the global effects only when necessary. Our effect frame rule is a particular example of the generalised frame rule of the views framework [11]. We can therefore appeal to the general soundness result of the views framework to prove the soundness of our reasoning.

We provide a sequential specification for a core subset of the POSIX file system which is faithful to the English specification. We demonstrate our modular reasoning about client programs using the standard recursive remove utility (`rm -r`), giving a natural specification, demonstrating that the busybox, GNU Core Utils and FreeBSD implementations are incorrect, and providing a simple fix for the busybox implementation which does satisfy our specification. Finally, we are able to specify correctness properties for a software installer with a versioning strategy based on symbolic links: the software installer either fully installs the software or rolls back to its previous state; and the installation keeps previous versions intact and usable.

We believe POSIX is an ideal real-world example for highlighting the difficulties associated with reasoning about local update and global traversal to the place where the update occurs. We have demonstrated the need for shadow permissions and the effect frame rule for sequential reasoning about POSIX. There are many other examples where we believe these techniques will be useful, such as graph algorithms and DOM querying. By studying such examples, we aim eventually to develop a general logic for reasoning about the combination of local update and global traversal.

1.1 Related Work

Global approach

Specifications of file systems based on first-order logic have been widely studied [2, 14–16, 19, 22], leading to the verification grand challenge by Joshi and Holzmann [21]. The work mainly focused on the specification of the file system and the correctness of implementations. First-order reasoning scales poorly when reasoning about file-system client programs. The POSIX English specification defines many operations using disjointness conditions between entries: for example, `rename` moves a directory from one place to another, as long as the directory being moved does not contain the place to which it is moving. First-order reasoning

enforces such disjointness conditions with reachability side-conditions. These constraints increase non-linearly with respect to the size of programs, as we have demonstrated in previous work [18] using `rename` as an example. Essentially, this is similar to the non-linear increase of reachability constraints when using first-order logic, to reason about standard heap update, compared to separation logic as demonstrated by Reynolds [24].

Local approach

Specifications of file systems based on separation logic have only recently been studied. Ernst *et al.* [13] give a global first-order specification of the POSIX file system, and show that a heap-based implementation of VFS (Virtual Filesystem Switch), an abstraction layer within the Linux implementation of file systems, is correct. They use separation logic to reason about the implementation. Their reasoning is limited to linear paths which nicely match the inductive directory tree structure. It is neither clear how to extend to arbitrary pathnames, nor how their techniques can be lifted to a local specification of POSIX. On the other hand, we wish to link our local POSIX specification to an implementation in the future. This is an obvious option to explore.

In our previous work [18], we introduced structural separation logic (SSL) to reason abstractly about complex structured data, observing that previous work on context logic [7] for reasoning about trees was not fine-grained enough. We provided an abstract specification of a sequential core fragment of the POSIX file system with linear paths, and used it to reason about client programs such as a simple software installer. Structural separation logic combines fine-grained local reasoning about directory fragments with global path promises which limits the frame to those environments satisfying the path promises. We conjectured that a combination of path promises (the environment must preserve certain path information) and path obligations (the current thread must preserve certain path information) would extend the work to arbitrary paths. However, when we attempted to expand on this we found that path promises and obligations do not work. A path can be broken during a local update, whereas path obligations require that the path is preserved.

In the work presented here, we introduce fusion logic to specify a sequential core fragment of the POSIX file system with arbitrary pathnames and symbolic links. In fact, we believe that our specification is also simpler than SSL in that our axioms are smaller and more readable since they are based on simple tree assertions rather than assertions based on tree contexts: for example, compare the axioms for `rename` given in figure 4 with those of SSL. We can also verify more realistic client programs: for example, unlike SSL, we can verify a software installer with versioning based on symbolic links. In future, we will verify Makefiles and admin scripts which require '..' for their relative pathnames. Indeed, we are finally in a position to talk to system administrators about interesting file-system use cases. For a more technical discussion on why SSL does not work with arbitrary paths, we refer the reader to the technical report [23].

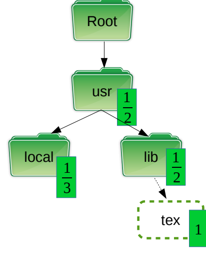
A natural question is whether techniques from fine-grained concurrent separation logics (such as deny-guarantee [12] and concurrent abstract predicates [10]), involving standard permissions and the standard separating conjunction, can be adapted to specify POSIX file systems. For example, da Rocha Pinto *et al.* [9] define concurrent abstract predicates for concurrent indexes that allow overlapping updates and reads. In particular, the predicate $\mathsf{in}_{\mathsf{def}}(k, v)$, stating that a thread can update the key k with value v, can be used at the same time as the predicate $\mathsf{read}(k)$, stating that a thread can only read the key k but other threads may change it. The predicates are required to be stable, meaning that $\mathsf{read}(k)$ does not imply that the key exists. POSIX commands require stronger preconditions. We need to know that an entry really does exist before update, and might not exist after update. A shadow permission in the precondition guarantees that the entry does exist. A shadow permission in the postcondition describes an entry that may or may not exist depending on how the update is propagated by the effect conjunction. We did spend a significant amount of time trying to use shadow permissions with just the standard separating conjunction and the frame rule. However, all our attempts involved complex well-formedness constraints on the preconditions of axioms and on the frame rule, severely complicating the reasoning. We now believe that our effect frame rule is fundamental.

Another natural question is whether Hobor and Villard's reasoning about graph algorithms [20], using "sepish" (overlapping conjunction) connective of [17, 20], can be adapted to POSIX file systems. Sepish allows overlaps at the cost of disjointness. It is possible to regain disjointness with reachability constraints. However, just as the `rename` example is used to show that first-order reasoning does not scale [18], it can also be used to show that sepish reasoning does not scale. In contrast, fusion expresses overlaps whilst still retaining disjointness information. Our reasoning does scale, because no reachability constraints are needed, but also the effect frame makes the reasoning modular; it is a frame rule.

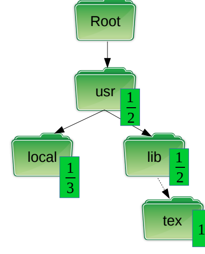
Finally, Biri and Galmiche propose a separation logic for simple tree updates with paths [3]. They use a similar tree model but without any permission structure to control resource sharing. This approach leads to non-local axioms and, as the authors admit, a frame rule that is only sound when allocation of new nodes in prohibited. In contrast, we avoid such issues by using the permission system to control resource sharing as is standard with fractional permissions.

2. Example Specifications

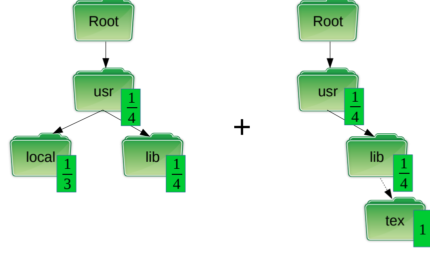
We focus on a sequential core fragment of POSIX file system commands. Even though small, it includes the basic primitive commands for manipulating the file system's structure.



(a) An instrumented directory tree on which we can run `mkdir` with pathname `/usr/local/../lib/tex`.



(b) The resulting instrumented directory tree after running `mkdir` with pathname `/usr/local/../lib/tex`.



(c) Composition of two instrumented directory trees, equivalent to figure 1b by fusion.

Figure 1: Examples of partial directory trees.

Consider the command `r := mkdir(path)` where the variable `path` has value `/usr/local/../lib/tex`. According to its POSIX description, it creates a new empty directory named `tex` within the directory identified by the path `/usr/local/../lib`. The intuitive tree *footprint* of `mkdir` is a partial directory tree consisting of only those entries required by the path traversal, the global *pathname footprint*, and those entries updated by the command, the local *update footprint*. The local update footprint requires full ownership so that it can be updated. The global pathname footprint requires partial ownership so that paths identifying different disjoint entries can share common prefixes.

To do the update in our example, we require partial ownership of the entries in the directory tree that are required by the path `/usr/local/../lib`, and full knowledge and ownership that `tex` is not in the directory at the end of the path. To express this information, we instrument the entries of a directory tree with permissions and extend entries to also describe entry non-existence. Consider figure 1a, which gives a partial directory tree corresponding to the footprint of `mkdir(path)`. The *fractional permissions* on `usr`, `local` and `lib` indicate that they are shared with the rest of the file system (the rest of the file system is not depicted). In addition, the `lib` directory contains the *non-existent entry* `tex` with *full permission* given by 1. This indicates that, although we do not have a complete description of the directory `lib`, we know for certain that `tex` is not in the directory and we have exclusive permission to create a new entry named `tex`. Figure 1b depicts the tree after the update, with a new empty sub-directory `tex` under `lib`, again with full permission.

We introduce fusion logic, consisting of *tree assertions* for reasoning about such tree footprints. A tree assertion satisfied by the partial directory tree in figure 1a is $\mathsf{tree}\big(\top\big[\mathtt{usr}_{1/2}\big[\mathtt{local}_{1/3}[\varnothing] * \mathtt{lib}_{1/2}\big[\widehat{\mathtt{tex}_1}\big]\big]\big]\big)$, where $\top$ is the root directory, $\widehat{\mathtt{tex}_1}$ denotes the non-existence of a `tex` entry with permission 1 and the fractions denote sharing with the rest of the directory tree. The shape of such tree assertions is familiar: for example, from ambient logic [8]. The addition of permission instrumentation and non-entries to such tree assertions is novel to fusion logic, although they have been used in reasoning about concurrency ([6] for permissions; [10] for non-entry). It is also convenient to use path syntax in tree assertions. For example, the assertion $\mathsf{tree}\big(/\mathtt{usr}_{1/2}/\mathtt{local}_{1/3}/../\mathtt{lib}_{1/2}/\widehat{\mathtt{tex}_1}\big)$ is satisfied by the same partial directory tree as the previous assertion. The instrumented path syntax $/\mathtt{usr}_{1/2}/\mathtt{local}_{1/3}/../\mathtt{lib}_{1/2}$ identifies the nodes of the partial directory tree and the order in which they are traversed. The $\widehat{\mathtt{tex}_1}$ assertion is as before.

We can give the following small axiom for `mkdir`:

$$\big\{\mathsf{var}(\mathtt{r}, -) * \mathsf{pvar}(\mathtt{path}, p/a_1) * \mathsf{tree}(p/\widehat{a}_1)\big\}$$
$$\mathtt{r\ :=\ mkdir(path)}$$
$$\big\{\mathsf{var}(\mathtt{r}, 0) * \mathsf{pvar}(\mathtt{path}, p/a_1) * \mathsf{tree}(p/a_1[\varnothing])\big\}$$

In the precondition, the variable assertion $\mathsf{var}(\mathtt{r}, -)$ states that the program variable `r`, in which the return value of the command will be stored, currently has some value that we do not care about. The variable path assertion $\mathsf{pvar}(\mathtt{path}, p/a_1)$ states that the program variable `path` has a value given by the path p/a_1. The tree assertion $\mathsf{tree}(p/\widehat{a}_1)$ describes a partial directory tree where the nodes in the tree are de-

scribed by path p and the non-entry a has full permission and hence can be updated. In the postcondition, the tree assertion $\mathsf{tree}(p/a_1[\varnothing])$ states that, under p, we now have an empty directory named a with full permission. The command returns the value 0 to indicate a successful update and this specification implies that nothing else has been modified.

Now consider the command `r := rmdir(path)` where, this time, the variable `path` is a linear path `/usr/lib/tex`. According to the English specification, it removes the directory `tex` under path `/usr/lib` when it is empty. Consider figure 1b. The command will remove the directory `tex` that we had previously created. However, with this path, the tree in figure 1b is bigger than the command's intuitive footprint. Since the path does not go through the `local` directory, it should not be part of the small footprint. Figure 1c illustrates how the instrumented directory tree in figure 1b can be separated: the tree on the right of the composition is a tree footprint for our example; and the tree on the left is not necessary since either the structure is already shared by the tree on the right, or the structure (in this case the `local` directory) is not touched.

The tree in figure 1b can be described by the assertion given by path syntax:

$$\mathsf{tree}\big(/\mathtt{usr}_{1/2}/\mathtt{local}_{1/3}/../\mathtt{lib}_{1/2}/\mathtt{tex}_1[\varnothing]\big)$$

This can be rewritten as the following assertion, satisfied by the partial directory trees in figure 1c:

$$\mathsf{tree}\big(/\mathtt{usr}_{1/4}/\mathtt{local}_{1/3}/../\mathtt{lib}_{1/4}[\varnothing]\big) \\ * \mathsf{tree}\big(/\mathtt{usr}_{1/4}/\mathtt{lib}_{1/4}/\mathtt{tex}_1[\varnothing]\big)$$

Then, we can frame-off the top assertion and use the bottom one as the precondition of the update. After the update, we can frame it back on, and *fuse* the resulting assertions to yield one satisfied by the directory tree in figure 1a.

When the paths are complex (non-linear), `rmdir` requires more complicated reasoning. Consider the case when `path` is `/usr/lib/tex/../tex`. The tree footprint is the right hand side of figure 1c. This time, interestingly, the update footprint and the pathname footprint overlap, as both require the same `tex` sub-directory. Standard permissions are not enough to cope with this situation. The local update requires full permission 1, but the global path traversal requires fractional permissions $0 < \pi < 1$ which is inconsistent with the full permission of the update. Our solution is to introduce a non-standard *shadow* permission 0, with the unusual meaning that $\mathtt{tex}_0$ denotes that we know the directory `tex` exists before the update (in this case, the command has followed a path using `tex`) but we do not know if it will exist after the update (in this case, `rmdir` has removed it).

Consider the following path-based assertion satisfied by the right-hand-side tree of figure 1c:

$$\mathsf{tree}\big(\mathtt{usr}_{1/4}/\mathtt{lib}_{1/4}/\mathtt{tex}_0/../\mathtt{tex}_1[\varnothing]\big)$$

which is equivalent to the assertion:

$$\mathsf{tree}\big(\top\big[\mathtt{usr}_{1/4}\big[\mathtt{lib}_{1/4}[\mathtt{tex}_0[\varnothing] * \mathtt{tex}_1[\varnothing]]\big]\big]\big)$$

The local update on the directory has an effect on the path in the sense that, after update, the path `/usr/lib/tex` no longer exists. This means that not only must the $\mathtt{tex}_1[\varnothing]$ be removed, but also the effect of the update must be propagated through the rest of the tree assertion. We achieve this by introducing the *effect separating conjunction* $\hat{*}$, which propagates the effects of updates to shared entries. In this example, after update, we obtain the assertion:

$$\mathsf{tree}\big(\top\big[\mathtt{usr}_{1/4}\big[\mathtt{lib}_{1/4}\big[\mathtt{tex}_0[\varnothing] \mathbin{\hat{*}} \widehat{\mathtt{tex}_1}\big]\big]\big]\big)$$

The composition $\mathtt{tex}_0[\varnothing] \mathbin{\hat{*}} \widehat{\mathtt{tex}_1}$ propagates the effect of the update by discarding the now outdated $\mathtt{tex}_0[\varnothing]$, leading to the assertion $\mathsf{tree}\big(\top\big[\mathtt{usr}_{1/4}\big[\mathtt{lib}_{1/4}\big[\widehat{\mathtt{tex}_1}\big]\big]\big]\big)$.

It is helpful in postconditions to introduce the path-based syntax $\mathsf{tree}\big(/\mathtt{usr}_{1/4}/\mathtt{lib}_{1/4}/\mathtt{tex}_0/..{/\!\!/}\widehat{\mathtt{tex}_1}\big)$ as equivalent to $\mathsf{tree}\big(\top\big[\mathtt{usr}_{1/4}\big[\mathtt{lib}_{1/4}\big[\mathtt{tex}_0[\varnothing] \mathbin{\hat{*}} \widehat{\mathtt{tex}_1}\big]\big]\big]\big)$. Using this path-based notation, the small axiom for `rmdir` is:

$$\big\{\mathsf{var}(\mathtt{r}, -) * \mathsf{pvar}(\mathtt{path}, p/a_1) * \mathsf{tree}(p/a_1[\varnothing])\big\}$$
```
r := rmdir(path)
```
$$\big\{\mathsf{var}(\mathtt{r}, 0) * \mathsf{pvar}(\mathtt{path}, p/a_1) * \mathsf{tree}(p{/\!\!/}\widehat{a}_1)\big\}$$

This specification is simple despite the complex behaviour of updates using arbitrary pathnames.

The local update of `rmdir` does not affect only the path identifying the update, as seen in the previous example. Any disjoint directory from the one being removed can be identified by a path that overlaps it. *Local* update can therefore have a *global* effect. We introduce the *effect frame rule* which uses the effect separating conjunction $\hat{*}$ to propagate the effects of the update, to larger footprints:

$$\textsc{EffectFrame}\,\frac{\vdash\{P\}\mathbb{C}\{Q\}}{\vdash\{P * R\}\mathbb{C}\{Q \mathbin{\hat{*}} R\}}$$

This rule is an instance of the generalised frame rule of views [11]. We prove soundness by appealing to views, as summarised in section 4.3 and appendix B.

Now consider running `rmdir(path)` on a directory tree described by:

$$\mathsf{tree}\big(/\mathtt{usr}_{1/4}/\mathtt{lib}_{1/4}/\mathtt{tex}_1[\varnothing]\big) \\ * \mathsf{tree}\big(/\mathtt{usr}_{1/4}/\mathtt{lib}_{1/4}/\mathtt{tex}_0/../../\mathtt{local}_{1/3}[\varnothing]\big)$$

where `path` is `/usr/lib/tex`. The proof derivation is:

In the following, let: $p = /\mathtt{usr}_{1/4}/\mathtt{lib}_{1/4}$

$\left\{\begin{array}{l}\mathsf{var}(\mathtt{r}, -) * \mathsf{pvar}(\mathtt{path}, p/\mathtt{tex}_1) * \mathsf{tree}(p/\mathtt{tex}_1[\varnothing]) \\ \quad * \mathsf{tree}\big(p/\mathtt{tex}_0/../../\mathtt{local}_{1/3}[\varnothing]\big)\end{array}\right\}$

EFFECT FRAME:

$\big\{\mathsf{var}(\mathtt{r}, -) * \mathsf{pvar}(\mathtt{path}, p/\mathtt{tex}_1) * \mathsf{tree}(p/\mathtt{tex}_1[\varnothing])\big\}$
```
// apply the rmdir axiom
r := rmdir(path)
```
$\big\{\mathsf{var}(\mathtt{r}, 0) * \mathsf{pvar}(\mathtt{path}, p/\mathtt{tex}_1) * \mathsf{tree}\big(p{/\!\!/}\widehat{\mathtt{tex}_1}\big)\big\}$
```
// consequence rule:
// propagate effect on the path
```
$\big\{\mathsf{var}(\mathtt{r}, 0) * \mathsf{pvar}(\mathtt{path}, p/\mathtt{tex}_1) * \mathsf{tree}\big(p/\widehat{\mathtt{tex}_1}\big)\big\}$

$\left\{\begin{array}{l}\mathsf{var}(\mathtt{r}, 0) * \mathsf{pvar}(\mathtt{path}, p/\mathtt{tex}_1) * \mathsf{tree}\big(p/\widehat{\mathtt{tex}_1}\big) \\ \quad \hat{*}\ \mathsf{tree}\big(p/\mathtt{tex}_0/../../\mathtt{local}_{1/3}[\varnothing]\big)\end{array}\right\}$
```
// consequence rule:
// propagate effect on the recombined resource
```
$\left\{\begin{array}{l}\mathsf{var}(\mathtt{r}, 0) * \mathsf{pvar}(\mathtt{path}, p/\mathtt{tex}_1) * \mathsf{tree}\big(p/\widehat{\mathtt{tex}_1}\big) \\ \quad * \mathsf{tree}\big(p/../\mathtt{local}_{1/3}[\varnothing]\big)\end{array}\right\}$

In the precondition, the assertion $\mathsf{tree}(p/\texttt{tex}_1[\varnothing])$ describes all the resource required by `rmdir`. We therefore frame-off the assertion $\mathsf{tree}(p/\texttt{tex}_0/../../\texttt{local}_{1/3}[\varnothing])$. Notice, however, that the path of this assertion overlaps the directory being removed. After framing-off this assertion, we can apply `rmdir`'s axiom to yield $\mathsf{tree}(p/\!\!/\widehat{\texttt{tex}_1})$. Note the use of $/\!\!/$ in the assertion, as required by the axiom's postcondition. In this particular example, the path p does not overlap with the removed `tex` directory, and so we use the consequence rule to obtain $\mathsf{tree}(p/\widehat{\texttt{tex}_1})$. Next, we frame-on the assertion we had previously set aside, using the effect separating conjunction $\hat{*}$. Using the consequence rule, we can remove $\texttt{tex}_0$ and one '..' to obtain an assertion where all effects have been propagated (no more $\hat{*}$). The postcondition is now ready to be used as a precondition for another command.

Consider the case where after the previous program we run `mkdir(path)` where `path` is using the previously removed directory, e.g. `/usr/lib/tex/../../local/lib`. From `mkdir`'s precondition, the assertion $\mathsf{tree}(p/\widehat{a}_1)$ states that the final component of the path must not exist, but the prefix p must. Given the previous example's postcondition we see that the prefix `/usr/lib/tex/../../local` is no longer valid and therefore we cannot establish `mkdir`'s precondition. This didactic example highlights the fact that extra care must be taken when using such complex paths in programs. This is especially true with programs that work on path arguments. In section 5, we demonstrate that well-known implementations of recursive remove are incorrect. Fortunately, by reasoning about the effects of updates on paths, we can expose this lack of care.

Finally, consider the `r := rename(old, new)` command which moves entries in the directory tree. It has many cases depending on whether a file or directory is moved and whether the target exists. Consider the case of moving a directory to an already existing target. According to the English specification, both paths must identify directories but the one identified by `new` must be empty[2]. The two paths may have a shared pathname prefix and, crucially, may mutually overlap the directories they identify. The axiomatic specification for this case is:

$$\left\{\begin{array}{c}\mathsf{var}(\mathtt{r},-) * \mathsf{pvar}(\mathtt{old}, p/a_1) * \mathsf{pvar}(\mathtt{new}, p'/b_1)\\ * \,\mathsf{tree}(p/a_1[d]) * \mathsf{tree}(p'/b_1[\varnothing])\end{array}\right\}$$
$$\mathtt{r\ :=\ rename(old,\ new)}$$
$$\left\{\begin{array}{c}\mathsf{var}(\mathtt{r},0) * \mathsf{pvar}(\mathtt{old}, p/a_1) * \mathsf{pvar}(\mathtt{new}, p'/b_1)\\ * \,\mathsf{tree}(p/\!\!/\widehat{a}_1) \mathbin{\hat{*}} \mathsf{tree}(p'/b_1[d])\end{array}\right\}$$

In the precondition, $\mathsf{tree}(p/a_1[d])$ describes the tree where, at the end of p, there is a directory named a with contents given by directory forest d. The assertion has full permission of both a and all its contents d (full permission propagates to descendants). The assertion $\mathsf{tree}(p'/b_1[\varnothing])$ describes the directory tree where, at the end of p', there is an empty directory b with full permission. The full permission on both updated directories ensures that they are disjoint. There might be shadow permissions in p and p', meaning that they might overlap with e.g. the directories a and b. The full permission on a guarantees that b is not its descendant. In the postcondition, $\mathsf{tree}(p/\!\!/\widehat{a}_1)$ ensures that effects are propagated to p. The assertion $\mathsf{tree}(p'/b_1[d])$ does not require any propagation, as the addition of d does not affect p'. However, the removal of $a_1[d]$ under p may, and $\hat{*}$ ensures effect propagation to p'.

3. Core POSIX File Systems

A program state comprises: a *file system*, which represents the directory tree and associated files; a *process heap*, which represents the memory contents of a client; and a *variable store*, which represents the values of program variables.

3.1 File System

The file system structure is a directed acyclic graph consisting of a directory tree and files[3]. Files are uniquely identified by *inodes* from the set INODES, ranged over by $\iota, k, \ldots$. We use the set FNAMES, ranged over by $a, b, \ldots$, for naming both files and directories.

Definition 1 (Directories)**.** *Concrete* directory forests, $cd \in$ DIRS*, are defined by:*

$$cd ::= \varnothing \mid a[cd] \mid a : \iota \mid cd \otimes cd$$

where $\varnothing$ is the empty directory forest, $a[cd]$ is a directory named a containing sub-directories cd, $a : \iota$ is a file link associating file name a with inode ι, and $\otimes$ is the composition of directories. The equivalence relation, $cd \equiv cd'$, states that $\otimes$ is commutative and associative with identity $\varnothing$. Well-formed directory forests have sibling distinct names. We only consider concrete directory forests that are well formed.

The set of directory trees *is* DTREES $\triangleq \{\top[cd] \mid cd \in$ DIRS$\}$ *where* $\top \notin$ FNAMES *denotes the root directory. Entry types are defined as* FTYPES $\triangleq \{F, D\}$*, where F denotes a file link type and D denotes a directory type.*

Definition 2 (File Heaps)**.** *Let* BYTES *be the set of byte sequences, ranged over by α, β. A* file heap*,* $\mathrm{f} \in$ FILES*, defined as* f : INODES $\stackrel{fin}{\rightharpoonup}$ BYTES*, is a partial function mapping inodes to byte sequences.*

A directory tree combined with a file heap forms a file system. However, for a file system to be well formed, we require that the inode of every file link in the directory tree must be in the domain of the file heap.

Definition 3 (File Systems)**.** File systems *are defined as:*

$$\text{FS} \triangleq \{fs \in \text{DTREES} \times \text{FILES} \mid \text{inodes}(fs\downarrow_1) \subseteq \text{dom}(fs\downarrow_2)\}$$

where $fs\downarrow_i$ denotes the ith projection on fs and inodes : DTREES $\rightarrow \mathcal{P}$(INODES) *denotes the set of all inodes referenced within a directory tree.*

---

[2] Note that the primitive operation `rename` works differently from the shell utility `mv`, where `mv old new` will place `old` into `new`.

[3] As most implementations we only allow hard links to files. Directory hard links introduce cycles which are not detectable during directory traversal.

Definition 4 (Concrete Paths)**.** *A concrete relative path, $crp \in$ RELPATHS, is a sequence of filenames defined by:*

$$crp ::= . \mid .. \mid a \mid ./crp \mid ../crp \mid a/crp$$

with an equivalence relation, $crp \equiv crp'$, stating that $/$ is associative with '.' as identity.

Absolute paths $\text{ABSPATHS} \triangleq \{/crp \mid crp \in \text{RELPATHS}\}$, *are paths that begin at the root directory. Equivalence is extended to absolute paths: $/crp \equiv /crp' \iff crp \equiv crp'$. The set of all paths is* $\text{PATHS} \triangleq \text{RELPATHS} \cup \text{ABSPATHS}$.

Commands accept concrete paths as arguments. In order to relate the partial directory tree footprints of commands with the paths they use, we instrument paths with permissions in section 4.1.

3.2 Process Heaps

A process heap represents the contents of the heap during program execution. It consists of structures used for controlling access to directories and files by IO (input/output) commands: directory streams and open file descriptions. It also contains a standard program heap. In this paper, we only consider directory IO. We refer the reader to our technical report [23] for details on file IO and standard program heaps.

A directory stream is the POSIX abstraction for directory iterators. The command `opendir(path)` is used to create a directory stream, for the directory identified by `path`. The command `readdir(dir)` is then used to iterate over the directory stream `dir`, either returning an entry or `null` (0) when the iteration completes. POSIX does not specify the ordering of returned entries and allows directory streams to act as mutable iterators in which case updates to the directory's contents may or may not be observed. For simplicity, in this paper we consider directory streams to act as immutable iterators simply providing a snapshot.

Definition 5 (Process Heaps)**.** *Assume a set of directory stream addresses* DSADDRS. *Process heaps,* $\text{ph} \in \text{PH}$*, are heaps of directory streams:*

$$\text{PH} \triangleq \text{DSADDRS} \overset{fin}{\rightharpoonup} \mathcal{P}(\text{FNAMES})$$

3.3 Variable Stores

Assume a set of program variables VARS, and a set of values $\text{VALS} \triangleq \text{PATHS} \cup \mathbb{Z} \cup \text{FTYPES} \cup \text{DSADDRS} \cup \{\mathbf{true}, \mathbf{false}\}$. A variable store, $\sigma \in \Sigma$, is a partial function, $\sigma : \text{VARS} \rightharpoonup \text{VALS}$, mapping program variables to values.

Definition 6 (Program States)**.** *A program state, $st \in$ STATES, consists of a variable store, a process heap and a file system:* $\text{STATES} \triangleq \Sigma \times \text{PH} \times \text{FS}$.

3.4 Core Fragment

In this paper we consider the following fragment of POSIX file system commands, $\mathbb{C}_{\text{fs}}$:

```
C_fs ::=   r := mkdir(path) | r := rmdir(path)
         | r := unlink(path) | r := rename(old, new)
         | dir := opendir(path) | closedir(dir)
         | fn := readdir(dir) | t := stat(path)
         | p := realpath(path)
```

This fragment includes the primitive POSIX commands for manipulating the file-system structure and reading directories. In our technical report [23] we also include primitive commands for file IO. In POSIX, the commands are defined as C function interfaces. Here, we adapt them to a simple imperative language, abstracting the details of C. Our aim is to focus on file-system updates, which should be the same irrespective of the programming language environment.

We reason about programs written in a sequential WHILE language with calls to the commands our fragment. The language uses *program expressions*, $\text{Expr} \in \text{EXPR}$, which are constructed from values, variables, arithmetic and boolean operations (such as $+, -, \wedge$ and $\vee$) and path composition (using $/$). Expression evaluation $[\![-]\!]_{-} : \text{EXPR} \to \Sigma \rightharpoonup \text{VALS}$ is standard. The programming language is given by the following grammar:

$$\mathbb{C} ::= \begin{array}{l} \textbf{local } \texttt{var} \textbf{ in } \mathbb{C} \textbf{ end} \mid \textbf{if } \text{Expr} \textbf{ then } \mathbb{C} \textbf{ else } \mathbb{C} \textbf{ fi} \\ \mid \textbf{while } \text{Expr} \textbf{ do } \mathbb{C} \textbf{ done} \mid \textbf{skip} \mid \mathbb{C};\mathbb{C} \mid \texttt{x := } \text{Expr} \mid \mathbb{C}_{\text{fs}} \end{array}$$

4. Local Reasoning

File-system programs manipulate the concrete file-system states defined in the previous section. We instrument file systems with further information to express ownership and sharing in the form of permissions (§4.1). This follows the general approach of views [11], where reasoning is performed in terms of instrumented views of the underlying concrete program states. The instrumented views form the basis of assertions (§4.2) used in the program logic and axiomatic specifications of commands (§4.3).

4.1 Instrumentation

We instrument the directory entries of definition 1 with permissions in the range $[0, 1]$ to regulate their ownership:

- *Full permission*, with value 1, indicates full ownership on an entry, where we know that only we can update and overlapping entries can only be read. In the directory case, we know it is complete; nothing from the concrete directory is missing.
- *Fractional permission*, with value $0 < \pi < 1$, indicates partial ownership of an entry and that we can only read it. In the directory case, we own only some of its contents.
- *Shadow permission*, with value 0, indicates partial ownership of a directory, which cannot be updated but we know an overlapping directory with full permission can.

Inspired by fine-grained reasoning on concurrent sets [10], we add further instrumentation to *locally* describe the non-existence of entries. Otherwise, we would have to in-

spect all the entries in a directory to determine if something does not exist, leading to bigger footprints.

Definition 7 (Instrumented Directories)**.** Instrumented directory forests, $d \in \text{IDIRS}$*, are defined by:*

$$d ::= \varnothing \mid a_\pi[d] \mid a_\pi : \iota \mid \widehat{\langle S \rangle}_\pi \mid d \otimes d$$

where $\pi \in [0,1]$*,* $S \subseteq \text{FNAMES}$ *and* $\widehat{\langle S \rangle}_\pi$ *denotes a set of non-existing entries with permission* π*.*

Well-formed instrumented directory forests have the following constraints: siblings with $\pi = 1$ *have distinct names, all the descendants of a directory with* $\pi = 1$ *also have (a sum of)* $\pi = 1$*, and finally all the descendants of a directory with* $\pi = 0$ *also have* $\pi = 0$*. We only consider well-formed instrumented directory forests.* $\text{IDTREES} \triangleq \{\top[d] \mid d \in \text{IDIRS}\}$ *denotes instrumented directory trees.*

Note that full permission on a directory means all descendants also have full permission to enforce completeness. Shadow permission is also enforced on descendants accordingly[4]. We use sets for non-existing entries instead of individual names so that we can easily express the non-existence of potentially (countably) infinite sets of names. For example, $a_{1/2}\left[\widehat{\langle \text{FNAMES} \rangle}_{1/3}\right]$ describes an empty directory because we know all the entry names do not exist. In contrast, $a_{1/2}[\varnothing]$, does not describe an empty directory. The fractional permission $1/2$ indicates that we only know part of its contents. In this case we know $\varnothing$, in other words we know nothing about the contents of a.

In section 2 we informally described how instrumented directory entries are composed or *fused*. *Fusion* allows entries to be composed by merging entries with the same name and summing their permissions. Recall the `rmdir` example of section 2, where updating a full permission directory induces effects on shared shadow permission directories. These effects must be propagated so that globally shared directories remain consistent. *Effect fusion* extends normal fusion so that update effects are propagated by discarding any inconsistent shadow permission directories.

Definition 8 (Fusion equivalences)**.** *The* fusion equivalence *relation,* $\equiv \subseteq \text{IDIRS} \times \text{IDIRS}$*, is defined by the axioms:*

$$a_\pi[d] \otimes a_{\pi'}[d'] \equiv a_{\pi+\pi'}[d \otimes d'] \quad \text{if } \pi, \pi' > 0 \qquad (1)$$

$$a_\pi[d \otimes d'] \otimes a_0[d'] \equiv a_\pi[d \otimes d'] \qquad (2)$$

$$a_\pi : \iota \otimes a_{\pi'} : \iota \equiv a_{\pi+\pi'} : \iota \quad \text{if } \pi, \pi' > 0 \qquad (3)$$

$$\widehat{\langle S \rangle}_\pi \otimes \widehat{\langle S' \rangle}_\pi \equiv \widehat{\langle S \uplus S' \rangle}_\pi \quad \text{if } \pi > 0 \qquad (4)$$

$$\widehat{\langle S \rangle}_\pi \otimes \widehat{\langle S \rangle}_{\pi'} \equiv \widehat{\langle S \rangle}_{\pi+\pi'} \quad \text{if } \pi, \pi' > 0 \qquad (5)$$

and standard axioms stating $\otimes$ *is associative and commutative with unit* $\varnothing$*.*

The effect fusion equivalence *relation,* $\equiv_\wedge \subseteq \text{IDIRS} \times \text{IDIRS}$*, is defined by extending* $\equiv$ *with the following axioms:*

$$\widehat{\langle \{a\} \rangle}_1 \otimes a_0[d] \equiv_\wedge \widehat{\langle \{a\} \rangle}_1 \qquad (6)$$

$$a_1 : \iota \otimes a_0[d] \equiv_\wedge a_1 : \iota \qquad (7)$$

$$a_1[d] \otimes a_0[d'] \equiv_\wedge a_1[d] \quad \text{if } d \otimes d' \not\equiv d \qquad (8)$$

Both sets of axioms are closed under structural rules and equivalence accordingly.

Equation (1) allows a directory to be split into two parts, each getting some of its contents. Equation (2) allows creation of shadow permission directories. Note that the full permission directory retains all of its contents. Equation (3) simply allows sharing of file links. Equations (4) and (5) deal with sets of non-existing entries. The former allows the splitting of a set into disjoint subsets, in which case the subsets preserve the original permission. Note that by this axiom the permission on a set of non-existing entries applies to every individual element of the set. The latter allows sets to be shared. The effect fusion equations deal with stale shadow permission directories resulting from update. In equations (6) and (7) the directory has been removed and replaced by a file. In equation (8), when the side condition holds the contents in $a_0[d']$ do not agree with those in $a_1[d]$. Essentially, $a_0[d']$ is stale and we discard it.

In [23] we also instrument the file heap to allow fine grained local reasoning for IO. In this paper, we assume coarse grained IO and do not instrument file heaps.

Definition 9 (Instrumented Program States)**.** Instrumented filesystems *are defined as:* $\text{IFS} \triangleq \text{IDTREES} \times \text{FILES}$*. An instrumented program state,* $is \in \text{ISTATES}$*, is a triple comprising a variable store, a process heap and an instrumented file system:* $\text{ISTATES} \triangleq \Sigma \times \text{PH} \times \text{IFS}$*.*

We relate instrumented program states to concrete program states via a reification function. Reification is crucial for defining semantic consequence (§ 4.2) and justifying the soundness of our logic (§ 4.3).

Definition 10 (Reification)**.** *Instrumented program states are reified to program states via the reification function,* $\lfloor - \rfloor : \text{ISTATES} \to \mathcal{P}(\text{STATES})$*, defined as:*

$$\lfloor (\sigma, \text{ph}, (\top[d], \text{f})) \rfloor \triangleq \{(\sigma, \text{ph}, (\top[cd], \text{f}')) \mid \text{strip}(d) = cd \wedge \text{f}' = \text{f} \uplus \text{f}''\}$$

where the function $\text{strip} : \text{IDIRS} \rightharpoonup \text{DIRS}$ *is defined by propagating any outstanding effects (using* $\equiv_\wedge$*), stripping permissions and discarding non-existing entries when the result is a well-formed directory forest.*

Reification propagates all outstanding effects via $\equiv_\wedge$, before stripping directory instrumentation, and extends the file heap in all possible ways so that the reified file system is well formed according to definition 3.

Finally, we also instrument pathnames to use them as convenient syntax for describing the tree footprints of paths used

[4] Definition 7 allows all entries to have 0 permission. However, in our program reasoning and specifications we only use it for directories. Any other resource with 0 permission is effectively invalid.

by programs as described in section 2. The instrumentation simply adds permissions to the entry name path component.

Definition 11 (Instrumented Pathnames)**.** *Instrumented relative pathnames, $rp \in$ IRELPATHS, are defined by:*

$$rp ::= . \mid .. \mid a_\pi \mid ./rp \mid ../rp \mid a_\pi/rp$$

with an equivalence relation, $rp \equiv rp'$, stating that / is associative with '.' as identity. Absolute instrumented paths, IABSPATHS $\triangleq \{/rp \mid rp \in$ IRELPATHS$\}$*, are paths that begin at the root directory. Equivalence on instrumented relative paths is extended to instrumented absolute paths: $/rp \equiv /rp' \iff rp \equiv rp'$. The set of all instrumented paths is* IPATHS $\triangleq$ IRELPATHS $\cup$ IABSPATHS.

Instrumented linear paths are defined as ILPATHS $\triangleq$ $\{p \in$ IPATHS $\mid \nexists p', p'' \in$ IPATHS. $p \equiv p'/../p''\}$ and are ranged over by lp. The function lin : IPATHS $\to$ ILPATHS produces the linear path equivalent to an instrumented path:

$$\mathsf{lin}(p) = lp \overset{\text{def}}{\iff} p \equiv_{..} lp \wedge lp \in \text{ILPATHS}$$

where $\equiv_{..}$ extends $\equiv$ on paths with $a_\pi/../rp \equiv_{..} rp$ and $/../rp \equiv_{..} /rp$. Finally, we use the function $\mathrm{strippath}$: IPATHS $\to$ PATHS, to relate instrumented paths to concrete paths by simply removing the permission instrumentation.

We use instrumented directory trees to express the local tree footprints of commands, such as those of figure 1. We think of a local tree footprint consisting of two parts: a *path footprint*, which includes the directories required by a path; and an *update footprint*, which includes the entries we wish to update. We use the notation $/rp/d$ to describe such tree footprints. For example, in $/\mathtt{usr}_{1/2}/\mathtt{local}_{1/3}/../\mathtt{lib}_{1/2}/\mathtt{tex}_1[\varnothing]$ the absolute instrumented path $/\mathtt{usr}_{1/2}/\mathtt{local}_{1/3}/../\mathtt{lib}_{1/2}$ corresponds to the path footprint and the instrumented directory $\mathtt{tex}_1[\varnothing]$ corresponds to the update footprint. The path footprint is the natural one-holed context arising from the path. For example, the instrumented path $/\mathtt{usr}_{1/2}/\mathtt{local}_{1/3}$ gives rise to the path footprint $\top\big[\mathtt{usr}_{1/2}\big[\mathtt{local}_{1/3}[-]\big]\big]$, where $-$ is the context hole, and $/\mathtt{usr}_{1/2}/\mathtt{local}_{1/3}/../\mathtt{lib}_{1/2}$ gives rise to the path footprint $\top\big[\mathtt{usr}_{1/2}\big[\mathtt{local}_{1/3}[\varnothing] \otimes \mathtt{lib}_{1/2}[-]\big]\big]$. The notation $/\mathtt{usr}_{1/2}/\mathtt{local}_{1/3}/../\mathtt{lib}_{1/2}/\mathtt{tex}_1[\varnothing]$ thus describes the tree footprint given by the composition of the path footprint context with the update footprint directory:

$$\begin{aligned} &\top\big[\mathtt{usr}_{1/2}\big[\mathtt{local}_{1/3}[\varnothing] \otimes \mathtt{lib}_{1/2}[-]\big]\big] \circ \mathtt{tex}_1[\varnothing] \\ &= \top\big[\mathtt{usr}_{1/2}\big[\mathtt{local}_{1/3}[\varnothing] \otimes \mathtt{lib}_{1/2}[\mathtt{tex}_1[\varnothing]]\big]\big] \end{aligned}$$

Definition 12. *One-holed instrumented directory contexts, $c \in$ ICDIRS, are defined by*

$$c ::= \varnothing \mid a_\pi[c] \mid a_\pi : \iota \mid \widehat{\langle S \rangle}_\pi \mid c \otimes d \mid d \otimes c \mid -$$

with an equivalence relation, $c \equiv c'$, stating that $\otimes$ is associative and commutative with identity $\varnothing$. The same well-formedness conditions of instrumented directories (definition 7) apply. Context application, $\circ$: ICDIRS $\times$ ICDIRS $\to$ ICDIRS*, is standard.*

The conversion of an instrumented path to the corresponding one-holed context (up to equivalence) is subtle. Starting at the root, it involves walking inductively down the instrumented path rp, constructing the context as it goes. It is defined by a reduction relation on pairs of instrumented contexts and paths.

Definition 13 (Path-to-context reduction)**.**
$\downarrow \subseteq$ (ICDIRS $\times$ IPATHS) $\times$ (ICDIRS $\times$ IPATHS)

$$\begin{aligned}
(\top[-]\,, /rp) &\downarrow (\top[-]\,, rp) \\
(\top[c]\,, ..) &\downarrow (\top[c' \circ (a_\pi[d] \otimes -)]\,, .) \\
&\quad \textit{if } c \equiv c' \circ (a_\pi[d' \otimes -]) \\
(\top[c]\,, ..) &\downarrow (\top[c]\,, .) \quad \textit{if } c \equiv d \otimes - \\
(\top[c]\,, a_\pi) &\downarrow (\top[c \circ a_\pi[-]]\,, .) \\
(\top[c]\,, ./rp) &\downarrow (\top[c]\,, rp) \\
(\top[c]\,, ../rp) &\downarrow (\top[c' \circ (a_\pi[d'] \otimes -)]\,, rp) \\
&\quad \textit{if } c \equiv c' \circ (a_\pi[d \otimes -]) \\
(\top[c]\,, ../rp) &\downarrow (\top[c]\,, rp) \quad \textit{if } c \equiv d \otimes - \\
(\top[c]\,, a_\pi/rp) &\downarrow (\top[c \circ a_\pi[-]]\,, rp)
\end{aligned}$$

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

The reduction $\downarrow$ takes (on the left hand side) a partially constructed context and the path to convert, and produces (on the right hand side) an extended context, given by the first component in the path, and the remaining path. When the initial path is a single component ('..' or a_π), the reduction produces the final directory context and '.' to indicate that the path is fully reduced. We require two cases for '..': one case for going up to the parent directory when inside a normal directory; and the other for staying in the root directory when already at the top.

Note that $\downarrow$ is deterministic up to equivalence and each reduction decreases the remaining path by one component. Starting with the context $\top[-]$ and an absolute instrumented path $/rp$, the reduction relation $\downarrow^*$ will reach a unique final context (up to equivalence) and '.' in a finite number of steps. For example, consider the reduction for the instrumented path $/\mathtt{usr}_{1/2}/\mathtt{local}_{1/3}/../\mathtt{lib}_{1/2}$:

$$\begin{aligned}
&\big(\top[-]\,, /\mathtt{usr}_{1/2}/\mathtt{local}_{1/3}/../\mathtt{lib}_{1/2}\big) \\
&\downarrow \big(\top[-]\,, \mathtt{usr}_{1/2}/\mathtt{local}_{1/3}/../\mathtt{lib}_{1/2}\big) \\
&\downarrow \big(\top\big[\mathtt{usr}_{1/2}[-]\big]\,, \mathtt{local}_{1/3}/../\mathtt{lib}_{1/2}\big) \\
&\downarrow \big(\top\big[\mathtt{usr}_{1/2}\big[\mathtt{local}_{1/3}[-]\big]\big]\,, ../\mathtt{lib}_{1/2}\big) \\
&\downarrow \big(\top\big[\mathtt{usr}_{1/2}\big[\mathtt{local}_{1/3}[\varnothing] \otimes -\big]\big]\,, \mathtt{lib}_{1/2}\big) \\
&\downarrow \big(\top\big[\mathtt{usr}_{1/2}\big[\mathtt{local}_{1/3}[\varnothing] \otimes \mathtt{lib}_{1/2}[-]\big]\big]\,, .\big)
\end{aligned}$$

We define the notation p/d for arbitrary instrumented paths, by first reducing p to the corresponding context and then filling that context with the directory forest d.

Definition 14 (Path/tree notation)**.**

$$\begin{aligned} &rp/d = d' \\ \iff &\exists c.\, (-, rp) \downarrow^* (c, .) \wedge c \circ d = d' \end{aligned}$$

$$\begin{aligned} &/rp/d = \top[d'] \\ \iff &\exists c.\, (\top[-]\,, rp) \downarrow^* (\top[c]\,, .) \wedge c \circ d = d' \end{aligned}$$

Assertions P, Q	
Directory Tree	$\mathsf{tree}(\mathrm{E}/\Delta)$
Effect Directory Tree	$\mathsf{tree}(\mathrm{E}/\!\!/\Delta)$
Root Directory	$\mathsf{tree}(\top[\Delta])$
File Heap	$\mathsf{file}(\iota, \mathrm{E})$
Directory Stream	$\mathsf{ds}(\mathtt{dir}, \mathrm{E})$
Program Variable Value	$\mathsf{var}(\mathtt{x}, \mathrm{E})$
Separating Conjunction	$P * Q$
Effect Separating Conjunction	$P \mathbin{\hat{*}} Q$
Logical Expression	E
Empty	emp

Directory Assertions Δ, Δ'	
File Type Entry	$\mathrm{E}_{\mathrm{E}'} : \iota$
Directory Type Entry	$\mathrm{E}_{\mathrm{E}'}[\Delta]$
Non Existing Entries	$\widehat{\langle \mathrm{E} \rangle}_{\mathrm{E}'}$
Fusion	$\Delta * \Delta'$
Effect fusion	$\Delta \mathbin{\hat{*}} \Delta'$
Logical Expression	E
Empty Entry	$\varnothing$

where $\mathrm{E}, \mathrm{E}' \in$ LEXPRS are logical expressions and ι is a logical variable evaluating to an inode.

Figure 2: Fusion logic assertions.

The notation $/rp/d$ is defined for any number of '..'s. However, notice that, due to '..', the notation rp/d may not always be defined. For example, the notation $../\varnothing$ is not defined, but $\mathtt{usr}_{1/2}/\mathtt{local}_{1/3}/../../\varnothing$ is. The '..' requires a parent directory to go up, which may not always be present in a relative path. This is not surprising, in that relative paths only have meaning with respect to a suitable context.

4.2 Assertions

Assertions use logical variables and logical expressions, in a similar way to programs using program variables and expressions. We extend values, VALS from section 3.3, to include instrumented paths, instrumented directories, permissions, byte sequences and sets of such values, and use a logical environment, $\mathrm{e} \in$ LENVS, to map logical variables to the extended values. Logical expressions, $\mathrm{E} \in$ LEXPRS, are constructed from values, logical variables, arithmetic, boolean and set operations (such as $+, -, \wedge, \vee, \cup$ and $\cap$) and path composition (using $/$). Logical expression evaluation function, $(\!| - |\!)_{_} :$ LEXPRS $\rightarrow$ LENVS $\rightharpoonup$ VALS, is standard. We treat all variables in assertions that are not program variables as logical variables. We slightly abuse notation and use $p, q, \ldots$ for logical instrumented path variables, $a, b, \ldots$ for logical entry name variables, $\iota, j, \ldots$ for logical inode variables and $d, d', \ldots$ for logical instrumented directory tree variables.

Assertions $P, Q \in$ ASRTS are constructed from standard logical connectives and quantifiers and the left-hand-side assertions of figure 2. The assertion $\mathsf{tree}(\mathrm{E}/\Delta)$ describes a directory tree with path footprint given by the instrumented path that is the value of the logical expression E and update footprint described by the directory assertion Δ. The assertion $\mathsf{tree}(\mathrm{E}/\!\!/\Delta)$ describes a directory tree with path footprint given by E and update footprint described by Δ, where update effects are propagated from Δ to the path footprint. Recall the `rmdir` specification from section 2, where we use an assertion of this form so that when the path overlaps with the removed directory, the directory's removal is propagated to the path. To directly describe directory trees, without using paths we use the assertion $\mathsf{tree}(\top[\Delta])$ which describes the root directory with contents given by the directory assertion Δ. The assertion $\mathsf{file}(\iota, \mathrm{E})$ describes a file with inode ι and byte contents given by the logical expression E. The assertion $\mathsf{ds}(\mathtt{dir}, \mathrm{E})$ describes a directory stream with address `dir` containing the set of unread entries given by the logical expression E. The assertion $\mathsf{var}(\mathtt{x}, \mathrm{E})$ states that the value of the program variable `x` is given by the logical expression E. We follow the approach of variables as resource [5] in which the assertion states ownership of the variable. $P * Q$ is the standard *separating conjunction* describing states split into two parts, one satisfying P and the other satisfying Q. The *effect separating conjunction* $P \mathbin{\hat{*}} Q$ extends $*$, by propagating effects of P and Q to each other, with emp being the identity of both $*$ and $\hat{*}$. Finally, directory tree assertions, $\Delta \in$ IDIRENTRYASRTS, are constructed from standard logical connectives and quantifiers and the right-hand-side assertions of figure 2. These essentially lift instrumented directory trees to the assertion language. We overload $*$ and $\hat{*}$ to describe fusion (definition 8). Instrumented program states satisfying the assertions are given by satisfaction relations, $\models$: (LENVS $\times$ ISTATES) $\times$ ASRTS, and, $\models'$: (LENVS $\times$ IDIRS) $\times$ IDIRENTRYASRTS, defined in figure 3, for top level and directory assertions, respectively.

In $P * Q$, the separating conjunction states that P and Q have no effect on each other. It is always possible to replace $*$ with $\hat{*}$: $P * Q \implies P \mathbin{\hat{*}} Q$. The converse does not apply. Replacing $\hat{*}$ with $*$ is only possible by propagating the effects between the two assertions by using the effect fusion axioms of definition 8. These are directly lifted to the logic. For example, from equation (6) in definition 8 we have the logical axiom: $\widehat{\langle\{a\}\rangle}_1 \mathbin{\hat{*}} a_0[d] \iff \widehat{\langle\{a\}\rangle}_1$.

We also define the following derived assertions to use in our specifications:

Derived assertions

$$\widehat{a}_\pi \triangleq \widehat{\langle\{a\}\rangle}_\pi \qquad \mathsf{ent}_\pi(a) \triangleq (\exists\iota.\ a_\pi : \iota) \vee a_\pi[\mathbf{true}]$$

$$\bigoplus_{x \in \emptyset} \Delta \triangleq \varnothing \qquad \bigoplus_{x \in \{n\}} \Delta \triangleq \Delta[n/x]$$

$$\bigoplus_{x \in \{n\} \uplus S} \Delta \triangleq \Delta[n/x] \bigoplus_{x \in S} \Delta$$

$$
\begin{array}{rcl}
\mathrm{e}, (\emptyset, \emptyset, (\top[d], \emptyset)) \models \mathsf{tree}(\mathrm{E}/\Delta) & \iff & \exists d', d''.\, \mathrm{e}, d' \models' \Delta \wedge \top[d''] = /(\!|\mathrm{E}|\!)_{\mathrm{e}}/d' \wedge d'' \equiv d \\
\mathrm{e}, (\emptyset, \emptyset, (\top[d], \emptyset)) \models \mathsf{tree}(\mathrm{E}/\!/\Delta) & \iff & \exists d', d''.\, \mathrm{e}, d' \models' \Delta \wedge \top[d''] = /(\!|\mathrm{E}|\!)_{\mathrm{e}}/d' \wedge d'' \equiv_{\wedge} d \\
\mathrm{e}, (\emptyset, \emptyset, (\top[d], \emptyset)) \models \mathsf{tree}(\top[\Delta]) & \iff & \mathrm{e}, d \models' \Delta \\
\mathrm{e}, (\emptyset, \emptyset, (\varnothing, \mathrm{f})) \models \mathsf{file}(\iota, \mathrm{E}) & \iff & \mathrm{f} = \mathrm{e}(\iota) \mapsto (\!|\mathrm{E}|\!)_{\mathrm{e}} \\
\mathrm{e}, (\sigma, \mathrm{ph}, (\varnothing, \emptyset)) \models \mathsf{ds}(\mathtt{dir}, \mathrm{E}) & \iff & \exists ds.\ \mathrm{ph} = ds \mapsto (\!|\mathrm{E}|\!)_{\mathrm{e}} \wedge \sigma = \mathtt{dir} \mapsto ds \\
\mathrm{e}, (\sigma, \emptyset, (\varnothing, \emptyset)) \models \mathsf{var}(\mathtt{x}, \mathrm{E}) & \iff & \sigma = \mathtt{var} \mapsto (\!|\mathrm{E}|\!)_{\mathrm{e}} \\
\\
\mathrm{e}, (\sigma, \mathrm{ph}, (\top[d], \mathrm{f})) \models P * Q & \iff & \exists \sigma', \sigma'', \mathrm{ph}', \mathrm{ph}'', d', d'', \mathrm{f}', \mathrm{f}''.(\sigma, \mathrm{ph}, (\top[d], \mathrm{f})) = (\sigma' \uplus \sigma'', \mathrm{ph}' \uplus \mathrm{ph}'', (\top[d], \mathrm{f}' \uplus \mathrm{f}')) \\
& & \wedge\, d \equiv d' \otimes d'' \wedge \mathrm{e}, (\sigma', \mathrm{ph}', (\top[d'], \mathrm{f}')) \models P \wedge \mathrm{e}, (\sigma'', \mathrm{ph}'', (\top[d''], \mathrm{f}'')) \models Q \\
\mathrm{e}, (\sigma, \mathrm{ph}, (\top[d], \mathrm{f})) \models P \mathbin{\hat{*}} Q & \iff & \exists \sigma', \sigma'', \mathrm{ph}', \mathrm{ph}'', d', d'', \mathrm{f}', \mathrm{f}''.(\sigma, \mathrm{ph}, (\top[d], \mathrm{f})) = (\sigma' \uplus \sigma'', \mathrm{ph}' \uplus \mathrm{ph}'', (\top[d], \mathrm{f}' \uplus \mathrm{f}')) \\
& & \wedge\, d \equiv_{\wedge} d' \otimes d'' \wedge \mathrm{e}, (\sigma', \mathrm{ph}', (\top[d'], \mathrm{f}')) \models P \wedge \mathrm{e}, (\sigma'', \mathrm{ph}'', (\top[d''], \mathrm{f}'')) \models Q \\
\mathrm{e}, (\emptyset, \emptyset, (\top[d], \emptyset)) \models \mathrm{E} & \iff & \top[d] = (\!|\mathrm{E}|\!)_{\mathrm{e}} \\
\mathrm{e}, (\emptyset, \emptyset, (\varnothing, \emptyset)) \models \mathsf{emp} & \iff & \mathbf{true} \\
\\
\mathrm{e}, d \models' \mathrm{E}_{\mathrm{E}'} : \iota & \iff & d \equiv (\!|\mathrm{E}|\!)_{\mathrm{e}_{(\!|\mathrm{E}'|\!)_{\mathrm{e}}}} : \mathrm{e}(\iota) \\
\mathrm{e}, d \models' \widehat{\langle \mathrm{E} \rangle}_{\mathrm{E}'} & \iff & d \equiv \widehat{\langle (\!|\mathrm{E}|\!)_{\mathrm{e}} \rangle}_{(\!|\mathrm{E}'|\!)_{\mathrm{e}}} \\
\mathrm{e}, d \models' \mathrm{E}_{\mathrm{E}'}[\Delta] & \iff & \exists d'.\, d \equiv (\!|\mathrm{E}|\!)_{\mathrm{e}_{(\!|\mathrm{E}'|\!)_{\mathrm{e}}}}[d'] \wedge \mathrm{e}, d' \models' \Delta \\
\mathrm{e}, d \models' \Delta * \Delta' & \iff & \exists d', d''.\, d \equiv d' \otimes d'' \wedge \mathrm{e}, d' \models' \Delta \wedge \mathrm{e}, d'' \models' \Delta' \\
\mathrm{e}, d \models' \Delta \mathbin{\hat{*}} \Delta' & \iff & \exists\, \mathrm{e}, d', d''.\, d \equiv_{\wedge} d' \otimes d'' \wedge \mathrm{e}, d' \models' \Delta \wedge \mathrm{e}, d'' \models' \Delta' \\
\mathrm{e}, d \models' \mathrm{E} & \iff & d \equiv (\!|\mathrm{E}|\!)_{\mathrm{e}} \\
\mathrm{e}, \varnothing \models' \varnothing & \iff & \mathbf{true}
\end{array}
$$

Figure 3: Satisfaction relation.

$$
\begin{aligned}
\mathsf{pvar}(\mathtt{path}, p) &\triangleq \exists cp.\ \mathsf{var}(\mathtt{path}, cp) \wedge \mathsf{strippath}(p) = cp \\
\mathsf{all}(d) &\triangleq\ d \wedge \exists S, S'.\ (\mathrm{FNAMES} = S \uplus S') \\
&\quad \wedge \left(\circledast_{n \in S} \exists \pi.\ \mathsf{ent}_{\pi}(n) \right) * \exists \pi.\ \widehat{\langle S' \rangle}_{\pi} \\
\mathsf{names}(S) &\triangleq \forall a.\ (a \in S \iff \mathbf{true} * \exists \pi.\ \mathsf{ent}_{\pi}(a)) \\
\mathrm{E} \doteq \mathrm{E}' &\triangleq \mathrm{E} = \mathrm{E}' \wedge \mathsf{emp}
\end{aligned}
$$

where $\mathsf{strippath}(p)$ is the strippath function from definition 11 lifted to the assertion language. The assertion $\widehat{a}_{\pi}$ is derived notation for a singleton set of a non-existing entry with name a and permission π. $\mathsf{ent}_{\pi}(a)$ describes a file or directory with name a and permission π. $\circledast_{x \in S} \Delta$ is the iterative version of $*$ on elements of the set S. The assertion $\mathsf{pvar}(\mathtt{path}, p)$ states that the path value of the program variable $\mathtt{path}$ matches the instrumented pathname p once permissions are stripped. The predicate $\mathsf{all}(d)$ describes a complete list of entries d, requiring ownership of all possible entries either existing (set S) or non-existing (set S'). $\mathsf{names}(S)$ describes entries for every file name in the set S and finally, $\mathrm{E} \doteq \mathrm{E}'$ provides a $*$-separated version of equals.

Consider the following two instrumented directories: $a_1[\varnothing]$ and $a_1\left[\widehat{b}_1\right]$. By reification (definition 10), both reify to exactly the same concrete directories. In the first, the directory contains no entries. In the second, we know that b does not exist but also that no other entries exist. They are semantically equivalent. This fact is part of the general notion of *semantic entailment* of views [11].

Definition 15 (Semantic Entailment)**.** *The semantic entailment relation, $\preccurlyeq\ \subset$ ASRTS $\times$ ASRTS, is defined as:*

$$
\begin{gathered}
P \preccurlyeq Q \iff \\
\forall\, R \in \mathrm{ASRTS}, \mathrm{e} \in \mathrm{LENVS}.\ \lfloor [\![P * R]\!]_{\mathrm{e}} \rfloor \subseteq \lfloor [\![Q * R]\!]_{\mathrm{e}} \rfloor
\end{gathered}
$$

where $[\![P]\!]_{\mathrm{e}} \triangleq \{ is \in \mathrm{ISTATES} \mid \mathrm{e}, is \models P \}$ *denotes the set of all instrumented states satisfying* P *and* $\lfloor - \rfloor$ *is the reification function from definition 10.*

With semantic entailment we can generalise the previous example: $a_1[\Delta \wedge \mathsf{notnamed}(S)] \preccurlyeq a_1\left[\Delta * \widehat{\langle S \rangle}_1\right]$ where $\mathsf{notnamed}(S)$ asserts that the names in S are not mentioned as top level entries in the directory and is defined as:

$$
\begin{gathered}
\mathsf{notnamed}(S) \triangleq \\
\left(\begin{gathered} \forall a.\, a \in S \iff \\ \neg \left(\begin{gathered} \mathbf{true} * \exists \pi.\ \mathsf{ent}_{\pi}(a) \\ \vee \left(\exists S'.\ \widehat{\langle S' \rangle}_{\pi} \wedge a \in S' \right) \end{gathered} \right) \end{gathered} \right)
\end{gathered}
$$

This allows us to express an empty directory simply with assertions of the form $a_1[\varnothing]$ and introduce sets of non-existing entries only when needed.

When $\pi < 1$, the assertion $a_{\pi}[\varnothing]$ does not mean that a is empty, but simply that we do not own anything from its contents (they are potentially framed off). Note that this is different from the assertion $a_{\pi}[\mathbf{true}]$ where we own anything, including nothing, from its contents. The only way to say that a partially-owned directory is empty is by owning the fact that every file and directory name does not exist within it, for example as with the assertion $a_{1/2}\left[\widehat{\langle \mathrm{FNAMES} \rangle}_{1/2}\right]$.

4.3 Program Logic

We present a Hoare-style program logic comprising: our small axioms for our POSIX fragment; the standard axioms for **skip** and assignment from separation logic; our new *effect frame* rule; the semantic consequence rule of the views framework [11]; and the standard Hoare inference rules for control-flow statements (e.g. **if**, **while**) and logical connectives (e.g., existential elimination, disjunction). We adopt a fault-avoiding partial-correctness interpretation of Hoare triples $\{P\}\ \mathbb{C}\ \{Q\}$: when the program $\mathbb{C}$ is run on a state reified from P, it will not fault and, if it terminates, will result in a state reified from Q.

The small axioms for our POSIX fragment are given in figure 4. They match very closely the English descriptions given in the POSIX standard. For example, consider the axiom for `mkdir(path)` which creates a new empty directory identified by `path`, as long as an existing entry with the same name does not already exist. In the precondition, the assertion $\mathsf{pvar}(\mathtt{path}, p/a_1)$ states that `path` has a path-prefix p and ends with the name a with full permission 1. Assertion $\mathsf{tree}(p/\widehat{a}_1)$ states that a does not exist at the end of p and a new entry can be created as the permission is 1. Assertion $\mathsf{var}(\mathtt{r}, -)$ states that variable `r` has an undeclared value using the variables-as-resource interpretation [5]. In the postcondition, assertion $\mathsf{tree}(p/a_1[\varnothing])$ states that a new empty directory, named a with permission 1, has been created at the end of path p. Assertion $\mathsf{var}(\mathtt{r}, 0)$ states that the return value captured by `r` is set to 0 indicating a successful update.

Now consider `rmdir(path)` which removes the empty directory identified by `path`. In the precondition, assertion $\mathsf{tree}(p/a_1[\varnothing])$ states that at the end of the path-prefix p there is an empty directory a which can be removed as indicated by full permission 1. As seen in the `rmdir` example in section 2, the path p can overlap with directory a. The removal of a therefore has an effect on p since parts of p may no longer exist. In the postcondition, assertion $\mathsf{tree}(p/\!\!/\widehat{a}_1)$ states that the entry a_1 does not exist and there is a potential effect of the removal of directory a on the path p which needs to be propagated, as described in section 2.

The propagation of local directory update effects can be more complicated than just effecting the path to the update. Consider the `rename` command. The first two axioms in figure 4 give the cases for renaming files and are straightforward. The next two axioms give the cases for renaming directories. The last axiom is a simple degenerate case. The third and fourth axioms are the most interesting. They state that the removal of the directory a may have an effect on both p, as given by $\mathsf{tree}(p/\!\!/\widehat{a}_1)$, and on p', as given by the effect separating conjunction $\hat{*}$.

As seen in section 2, local directory updates have global effects that need to be propagated to the environment. This is achieved by the *effect frame* rule:

$$\textsc{EffectFrame}\ \frac{\vdash \{P\}\mathbb{C}\{Q\}}{\vdash \{P * R\}\mathbb{C}\{Q \mathbin{\hat{*}} R\}}$$

As with the standard frame rule from separation logic, we can extend the state using an arbitrary assertion R. Unlike the frame rule, we must propagate the effect of the local update to R, as given by the *effect propagating conjunction* $\hat{*}$ in the postcondition of the conclusion. Note that the precondition of the conclusion, as well the preconditions of every axiom, use the standard separating conjunction which require pre-states where all global effects have already been propagated. This highlights a crucial aspect of our reasoning. After performing an update, it is essential to propagate all effects on the owned resources using the effect fusion axioms before continuing with subsequent updates.

Soundness The views framework [11] has associated with it a general soundness result. This means that it is enough to give certain parameters and establish certain properties for the soundness result to hold. We show that our effect frame rule is an instance of the generalised frame rule of views. The detailed proof can be found in the accompanying technical report [23]. We determine the necessary parameters: the primitive commands, $\mathbb{C}_{\mathrm{fs}}$ (§3.4); the concrete states, STATES (definition 6); the separation algebra $(\text{ISTATES}, \bullet, (\emptyset, \emptyset, (\varnothing, \emptyset)))$; the axioms of figure 4; and the reification function $\lfloor - \rfloor$ of definition 10. We give a more detailed account of the parameters and state the required properties in appendix B.

Theorem 1 (Soundness)**.** *Assume* $\vdash \{P\}\mathbb{C}\{Q\}$*. Then, for all* $\mathrm{e} \in \text{LENVS}$*,* $st \in \lfloor [\![P]\!]_{\mathrm{e}} \rfloor$ *and* $st' \in \text{STATES}$*, if* $\mathbb{C}, st \rightsquigarrow^* \textbf{skip}, st'$ *in the operational semantics then* $st' \in \lfloor [\![Q]\!]_{\mathrm{e}} \rfloor$*.*

5. Recursive remove

The primitive POSIX command `rmdir` removes a directory only if it is empty. More often than not, we need to remove non-empty directories. The POSIX standard does not specify a programming interface to do that (as in a C function), and so this either has to be implemented, or the shell (command line) utility `rm` with the option `-r` has to be used, which is specified in POSIX as one of the required utility programs systems must implement. An implementation of this utility is given in figure 5a. In fact, it is the implementation found in busybox stylised to our simple programming language.

Intuitively, the command removes the entry identified by its pathname argument, even if it is a non-empty directory. Formally, the specification is:

$$\begin{array}{c}\{\mathsf{pvar}(\mathtt{path}, p/a_1) * \mathsf{tree}(p/\mathsf{ent}_1(a))\}\\ \mathtt{rmr(path)}\\ \{\mathsf{pvar}(\mathtt{path}, p/a_1) * \mathsf{tree}(p/\!\!/\widehat{a}_1)\}\end{array}$$

In the precondition, `path` identifies either a file or directory with a full permission. In the postcondition, this entry is removed and we know that this has an effect on the path prefix p due to possible overlap of the path with the removed entry. For simplicity, we omit return values where possible.

$$\left\{\begin{array}{c}\mathsf{var}(\mathtt{r}, -) * \mathsf{pvar}(\mathtt{path}, p/a_\pi) * \mathsf{lin}(p/a_\pi) \doteq lp \\ * \,\mathsf{tree}(p/(d \wedge \mathsf{ent}_\pi(a)))\end{array}\right\}$$
```
r := realpath(path)
```
$$\left\{\begin{array}{c}\mathsf{pvar}(\mathtt{r}, lp) * \mathsf{pvar}(\mathtt{path}, p/a_\pi) * \mathsf{lin}(p/a_\pi) \doteq lp \\ * \,\mathsf{tree}(p/d)\end{array}\right\}$$

$$\left\{\mathsf{var}(\mathtt{r}, -) * \mathsf{pvar}(\mathtt{path}, p/a_1) * \mathsf{tree}(p/\widehat{a}_1)\right\}$$
```
r := mkdir(path)
```
$$\left\{\mathsf{var}(\mathtt{r}, 0) * \mathsf{pvar}(\mathtt{path}, p/a_1) * \mathsf{tree}(p/a_1[\varnothing])\right\}$$

$$\left\{\mathsf{var}(\mathtt{r}, -) * \mathsf{pvar}(\mathtt{path}, p/a_1) * \mathsf{tree}(p/a_1[\varnothing])\right\}$$
```
r := rmdir(path)
```
$$\left\{\mathsf{var}(\mathtt{r}, 0) * \mathsf{pvar}(\mathtt{path}, p/a_1) * \mathsf{tree}(p /\!\!/ \widehat{a}_1)\right\}$$

$$\left\{\mathsf{var}(\mathtt{r}, -) * \mathsf{pvar}(\mathtt{path}, p/a_1) * \mathsf{tree}(p/a_1 : \iota)\right\}$$
```
r := unlink(path)
```
$$\left\{\mathsf{var}(\mathtt{r}, 0) * \mathsf{pvar}(\mathtt{path}, p/a_1) * \mathsf{tree}(p/\widehat{a}_1)\right\}$$

$$\left\{\begin{array}{c}\mathsf{var}(\mathtt{r}, -) * \mathsf{pvar}(\mathtt{old}, p/a_1) * \mathsf{pvar}(\mathtt{new}, p'/b_1) \\ * \,\mathsf{tree}(p/a_1 : \iota) * \mathsf{tree}(p'/b_1 : \iota')\end{array}\right\}$$
```
r := rename(old, new)
```
$$\left\{\begin{array}{c}\mathsf{var}(\mathtt{r}, 0) * \mathsf{pvar}(\mathtt{old}, p/a_1) * \mathsf{pvar}(\mathtt{new}, p'/b_1) \\ * \,\mathsf{tree}(p/\widehat{a}_1) * \mathsf{tree}(p'/b_1 : \iota)\end{array}\right\}$$

$$\left\{\begin{array}{c}\mathsf{var}(\mathtt{r}, -) * \mathsf{pvar}(\mathtt{old}, p/a_1) * \mathsf{pvar}(\mathtt{new}, p'/b_1) \\ * \,\mathsf{tree}(p/a_1 : \iota) * \mathsf{tree}\left(p'/\widehat{b}_1\right)\end{array}\right\}$$
```
r := rename(old, new)
```
$$\left\{\begin{array}{c}\mathsf{var}(\mathtt{r}, 0) * \mathsf{pvar}(\mathtt{old}, p/a_1) * \mathsf{pvar}(\mathtt{new}, p'/b_1) \\ * \,\mathsf{tree}(p/\widehat{a}_1) * \mathsf{tree}(p'/b_1 : \iota)\end{array}\right\}$$

$$\left\{\begin{array}{c}\mathsf{var}(\mathtt{r}, -) * \mathsf{pvar}(\mathtt{old}, p/a_1) * \mathsf{pvar}(\mathtt{new}, p'/b_1) \\ * \,\mathsf{tree}(p/a_1[d]) * \mathsf{tree}(p'/b_1[\varnothing])\end{array}\right\}$$
```
r := rename(old, new)
```
$$\left\{\begin{array}{c}\mathsf{var}(\mathtt{r}, 0) * \mathsf{pvar}(\mathtt{old}, p/a_1) * \mathsf{pvar}(\mathtt{new}, p'/b_1) \\ * \,\mathsf{tree}(p /\!\!/ \widehat{a}_1) \mathbin{\hat{*}} \mathsf{tree}(p'/b_1[d])\end{array}\right\}$$

$$\left\{\begin{array}{c}\mathsf{var}(\mathtt{r}, -) * \mathsf{pvar}(\mathtt{old}, p/a_1) * \mathsf{pvar}(\mathtt{new}, p'/b_1) \\ * \,\mathsf{tree}(p/a_1[d]) * \mathsf{tree}\left(p'/\widehat{b}_1\right)\end{array}\right\}$$
```
r := rename(old, new)
```
$$\left\{\begin{array}{c}\mathsf{var}(\mathtt{r}, 0) * \mathsf{pvar}(\mathtt{old}, p/a_1) * \mathsf{pvar}(\mathtt{new}, p'/b_1) \\ * \,\mathsf{tree}(p /\!\!/ \widehat{a}_1) \mathbin{\hat{*}} \mathsf{tree}(p'/b_1[d])\end{array}\right\}$$

$$\left\{\begin{array}{c}\mathsf{var}(\mathtt{r}, -) * \mathsf{pvar}(\mathtt{old}, p/a_\pi) * \mathsf{pvar}(\mathtt{new}, p'/a_\pi) \\ * \,\mathsf{lin}(p/a_1) \doteq \mathsf{lin}(p'/a_1) * \mathsf{tree}(p/(d \wedge \mathsf{ent}_\pi(a)))\end{array}\right\}$$
```
r := rename(old, new)
```
$$\left\{\begin{array}{c}\mathsf{var}(\mathtt{r}, 0) * \mathsf{pvar}(\mathtt{old}, p/a_\pi) * \mathsf{pvar}(\mathtt{new}, p'/a_\pi) \\ * \,\mathsf{lin}(p/a_1) \doteq \mathsf{lin}(p'/a_1) * \mathsf{tree}(p/d)\end{array}\right\}$$

$$\left\{\mathsf{var}(\mathtt{t}, -) * \mathsf{pvar}(\mathtt{path}, p) * \mathsf{tree}(p/\varnothing)\right\}$$
```
t := stat(path)
```
$$\left\{\mathsf{var}(\mathtt{t}, D) * \mathsf{pvar}(\mathtt{path}, p) * \mathsf{tree}(p/\varnothing)\right\}$$

$$\left\{\mathsf{var}(\mathtt{t}, -) * \mathsf{pvar}(\mathtt{path}, p/a_\pi) * \mathsf{tree}(p/a_\pi : \iota)\right\}$$
```
t := stat(path)
```
$$\left\{\mathsf{var}(\mathtt{t}, F) * \mathsf{pvar}(\mathtt{path}, p/a_\pi) * \mathsf{tree}(p/a_\pi : \iota)\right\}$$

$$\left\{\mathsf{var}(\mathtt{dir}, -) * \mathsf{pvar}(\mathtt{var}, p) * \mathsf{tree}(p/\mathsf{all}(d))\right\}$$
```
dir := opendir(path)
```
$$\left\{\begin{array}{c}\mathsf{ds}(\mathtt{dir}, S) * \mathsf{pvar}(\mathtt{var}, p) \\ * \,\mathsf{tree}(p/\,(d \wedge \mathsf{names}(S)))\end{array}\right\}$$

$$\left\{\mathsf{var}(\mathtt{fn}, -) * \mathsf{ds}(\mathtt{dir}, S) \wedge S \neq \emptyset\right\}$$
```
fn := readdir(dir)
```
$$\left\{\mathsf{var}(\mathtt{fn}, a) * \mathsf{ds}(\mathtt{dir}, (S \setminus \{a\})) \wedge a \in S\right\}$$

$$\left\{\mathsf{var}(\mathtt{fn}, -) * \mathsf{ds}(\mathtt{dir}, \emptyset)\right\}$$
```
fn := readdir(dir)
```
$$\left\{\mathsf{var}(\mathtt{fn}, 0) * \mathsf{ds}(\mathtt{dir}, \emptyset)\right\}$$

$$\left\{\mathsf{ds}(\mathtt{dir}, S)\right\}$$
```
closedir(dir)
```
$$\left\{\mathsf{var}(\mathtt{dir}, \text{-})\right\}$$

Figure 4: Small axioms for the POSIX file-system fragment.

The implementation shown in figure 5a does not meet this intuitive specification. By trying to prove that the specification holds using our program logic we can easily detect the issue. The implementation first performs a `stat` to test whether `path` identifies a file or directory. The file case is simple. In the directory case, lines 7 to 14 read the contents of the directory one by one and recursively call `rmr` on each. Assume that the implementation satisfies the specification so that we can use it at the recursive call on line 10. However, note that from the specification's postcondition, removing the entry has an effect on the path that identifies it, which means that the path may no longer be valid after the update. In fact, we cannot establish a loop invariant where we know that the path prefix `path` exists.

At some point, the implementation of figure 5a will reach line 3 and invoke `stat` using a non-existing path. The axioms of figure 4 require paths to exist. In POSIX, commands fail with an error code when invalid paths are used. We specify such error cases in the technical report [23]. Here, we only discuss the `E_NOENT` error for `stat` that is triggered when a path is empty or does not identify an existing file or directory. This is captured by the following predicate:

$$\begin{array}{c}\mathtt{E_NOENT}(\mathtt{path}, v) \triangleq \\ \mathsf{var}(\mathtt{path}, v) \wedge \\ ((v \doteq 0) \vee (\exists p, a, p', \pi.\ \mathsf{pvar}(\mathtt{path}, p/a_\pi/p') * \mathsf{tree}(p/\widehat{a}_\pi)))\end{array}$$

with which we can give the following specification for `stat`:

$$\left\{\mathsf{var}(\mathtt{t}, -) * \mathsf{var}(\mathtt{errno}, -) * \mathtt{E_NOENT}(\mathtt{path}, v) \wedge t\right\}$$
```
t := stat(path)
```
$$\left\{\mathsf{var}(\mathtt{t}, -1) * \mathsf{var}(\mathtt{errno}, \mathtt{E_NOENT}) * \mathsf{var}(\mathtt{path}, v) * t\right\}$$

In the precondition we use the predicate to assert that the state is erroneous. Note that we use the variable t to capture the directory tree in which the error occurs. In the postcondition, the state is preserved, the global variable `errno` is

```
1:  rmr(path) ≜
2:      local t, dir, fn in
3:          t := stat(path)
4:          if t = F then
5:              unlink(path)
6:          else if t = D then
7:              dir := opendir(path)
8:              fn := readdir(dir)
9:              while fn ≠ 0 do
10:                 rmr(path/fn)
11:                 fn := readdir(dir)
12:             done
13:             closedir(dir)
14:             rmdir(path)
15:         fi
16:     end
```

(a) A naive recursive remove.

```
1:  rmrSafe(path) ≜
2:      local lp in
3:          lp := realpath(path)
4:          rmr(lp)
5:      end
```

(b) A safe recursive remove.

Figure 5: A naive recursive remove (top), that fails to remove a directory completely, and the suggested fix (bottom).

assigned the error code explaining the error and the return variable `r` is assigned -1 to indicate failure. Using the `stat` error specification we can prove that the actual specification for the naive implementation of figure 5a is:

$$\left\{\mathsf{var}(\mathtt{errno}, -) * \mathsf{pvar}(\mathtt{path}, p/a_1) * \mathsf{tree}(p/\mathsf{ent}_1(a))\right\}$$
$$\mathtt{rmr(path)}$$
$$\left\{\begin{array}{c}\mathsf{var}(\mathtt{errno}, e) * \mathsf{pvar}(\mathtt{path}, p/a_1) \\ * \,\mathsf{tree}(p /\!\!/ (\widehat{a}_1 \vee (e = \mathtt{E_NOENT} \wedge a_1[\mathbf{true}])))\end{array}\right\}$$

In the postcondition, the entry identified by the path argument may have been removed, or not. When the entry is not removed we know that it is a directory, but we do not know the actual remaining contents[5].

Correcting the implementation turns out to be simple. When `path` is linear, we get the following specification:

$$\left\{\mathsf{pvar}(\mathtt{path}, p/a_1) * \mathsf{islin}(p/a_1) * \mathsf{tree}(p/\mathsf{ent}_1(a))\right\}$$
$$\mathtt{rmr(path)}$$
$$\left\{\mathsf{pvar}(\mathtt{path}, p/a_1) * \mathsf{islin}(p) * \mathsf{tree}(p/\widehat{a}_1)\right\}$$

where $\mathsf{islin}(p) \triangleq \mathsf{lin}(p) \doteq p$ states that p is a linear path. In the postcondition, the entry identified by `path` is removed and we know that p remains valid, since linear paths never overlap with updated resource. The proof that the implementation satisfies the specification is given in figure 6.

A simple way to correct the implementation is to use the POSIX command `realpath`, which computes the linear path of any given valid path. In figure 5b we demonstrate such an implementation that meets the intended specification. Note that we reuse the naive `rmr` of figure 5a.

[5] A detailed proof is given in our technical report [23].

```
1:  {pvar(path, p/a_1) * islin(p/a_1) * tree(p/ent_1(a))}
2:  rmr(path) ≜
3:      local t, dir, fn in
4:          {var(t, −) * var(dir, −) * var(fn, −) * pvar(path, p/a_1)
                 * islin(p/a_1) * tree(p/ent_1(a))}
5:          t := stat(path)
6:          {∃d, t. var(t, t) * var(dir, −) * var(fn, −) * pvar(path, p/a_1)
             * islin(p/a_1) * tree(p/ ((a_1 : ι ∧ t = F) ∨ (a_1[d] ∧ t = D)))}
7:          if t = F then
8:              {var(t, F) * pvar(path, p/a_1) * islin(p/a_1) * tree(p/a_1 : ι)}
9:              unlink(path)
10:             {var(t, F) * pvar(path, p/a_1) * islin(p) * tree(p/â_1)}
11:         else if t = D then
12:             {var(t, D) * var(dir, −) * var(fn, −) * pvar(path, p/a_1)
                     * islin(p/a_1) * tree(p/a_1[d])}
13:             dir := opendir(path); fn := readdir(dir)
14:             {         ∃S, b, d'. ds(dir, S) * var(fn, b)
                          * pvar(path, p/a_1) * islin(p/a_1)
                 * tree(p/a_1[(       (b = 0 ∧ S = ∅ ∧ ∅)
                               ∨ (ent_1(b) * d' ∧ names(S \ {b})))])}
15:             while fn ≠ 0 do
16:                 rmr(path/fn); fn := readdir(dir)
17:             done
18:             {ds(dir, ∅) * var(fn, 0) * pvar(path, p/a_1) * islin(p/a_1)
                     * tree(p/a_1[∅])}
19:             closedir(dir); rmdir(path)
20:         fi
21:     end
22: {pvar(path, p/a_1) * islin(p) * tree(p/â_1)}
```

Figure 6: Proof derivation on a naive recursive remove implementation when we specifically use a linear pathname.

The proof is given in figure 7. Initially we use `realpath` to get a linear pathname for the `path` argument. However, before applying the linear `rmr` specification we showed earlier, we need to frame off from the state on line 7 the parts of the non-linear pathname p that are not shared with the linear pathname p_l. We do this on line 8 where the assertion $\mathsf{tree}(p_l/\mathsf{ent}_1(a))$ is the appropriate local resource for the linear specification of `rmr`. However, note that the recursive remove will have an effect on the framed-off assertion $\mathsf{tree}(p'/\varnothing)$. Next, on line 9, we apply the `rmr` specification to reach the state on line 11. Then we frame back on the rest of the state, including $\mathsf{tree}(p'/\varnothing)$. Note the use of $\hat{*}$ on the combined state. We propagate the effects on line 13, where the assertion $\mathsf{tree}(p /\!\!/ \widehat{a}_1)$ states that the path-prefix p is affected. The final postcondition follows trivially.

We have further investigated the widely used implementations of the `rm` utility found in GNU Coreutils and FreeBSD, to find similar issues in both. Executing the following set of commands on the system's shell:

```
$> mkdir -p /tmp/a/b/c
$> mkdir -p /tmp/a/e
$> rm -r /tmp/a/b/../..
```

which should remove the directory `/tmp/a` and its contents. Both implementations actually result in the directory not being removed but becoming empty instead. We have reported these issues to the appropriate mailing lists of the implementations we investigated.

This example demonstrates two things: i) the behaviour of non-linear paths with respect to update is not something

1: $\{\mathsf{pvar}(\texttt{path}, p/a_1) * \mathsf{tree}(p/\mathsf{ent}_1(a))\}$
2: $\texttt{rmrSafe(path)} \triangleq$
3: **local** `lp` **in**
4: $\{\mathsf{var}(\texttt{lp}, -) * \mathsf{pvar}(\texttt{path}, p/a_1) * \mathsf{tree}(p/\mathsf{ent}_1(a))\}$
5: $\left\{\begin{array}{l} \exists p_l.\, \mathsf{var}(\texttt{lp}, -) * \mathsf{pvar}(\texttt{path}, p/a_1) \\ \quad * \mathsf{tree}(p/\mathsf{ent}_1(a)) * \mathsf{lin}(p) \doteq p_l \end{array}\right\}$
6: `lp := realpath(path)`
7: $\left\{\begin{array}{l} \exists p_l.\, \mathsf{pvar}(\texttt{lp}, p_l) * \mathsf{pvar}(\texttt{path}, p/a_1) \\ \quad * \mathsf{tree}(p/\mathsf{ent}_1(a)) * \mathsf{lin}(p) \doteq p_l \end{array}\right\}$
8: $\left\{\begin{array}{c} \exists p_l, p'.\, \mathsf{pvar}(\texttt{lp}, p_l) * \mathsf{pvar}(\texttt{path}, p/a_1) * \mathsf{lin}(p) \doteq p_l \\ * \mathsf{tree}(p/\varnothing) \iff \mathsf{tree}(p_l/\varnothing) * \mathsf{tree}(p'/\varnothing) \\ * \mathsf{tree}(p_l/\mathsf{ent}_1(a)) * \mathsf{tree}(p'/\varnothing) \end{array}\right\}$
9: $\{\mathsf{pvar}(\texttt{lp}, p_l) * \mathsf{islin}(p_l) * \mathsf{tree}(p_l/\mathsf{ent}_1(a))\}$
10: `rmr(lp)`
11: $\{\mathsf{pvar}(\texttt{lp}, p_l) * \mathsf{islin}(p_l) * \mathsf{tree}(p_l/\widehat{a}_1)\}$
12: $\left\{\begin{array}{c} \exists p_l, p'.\, \mathsf{pvar}(\texttt{lp}, p_l) * \mathsf{pvar}(\texttt{path}, p/a_1) * \mathsf{lin}(p) \doteq p_l \\ * \mathsf{tree}(p/\varnothing) \iff \mathsf{tree}(p_l/\varnothing) * \mathsf{tree}(p'/\varnothing) \\ * \mathsf{tree}(p_l/\widehat{a}_1) \hat{*} \mathsf{tree}(p'/\varnothing) \end{array}\right\}$
13: $\left\{\begin{array}{c} \exists p_l.\, \mathsf{pvar}(\texttt{lp}, p_l) * \mathsf{pvar}(\texttt{path}, p/a_1) * \mathsf{lin}(p) \doteq p_l \\ * \mathsf{tree}(p /\!\!/ \widehat{a}_1) \end{array}\right\}$
14: **end**
15: $\{\mathsf{pvar}(\texttt{path}, p/a_1) * \mathsf{tree}(p /\!\!/ \widehat{a}_1)\}$

Figure 7: Proof derivation of the fixed recursive delete.

we can brush under the carpet, since clients expect the intuitive behaviour to hold as specified informally in the documentation and all it takes is one sufficiently “wrong” path to get unspecified results; and ii) reasoning about the effects of updates is crucial both to verify that unwanted effects do not leak to clients and to discover the cases where they do.

6. Symbolic links

Even though this paper focuses on paths with '..', the same techniques apply to symbolic links, requiring only minor extensions. We briefly discuss them here and refer readers to our technical report [23] for further details.

A symbolic link is a special file whose contents is a path (the symbolic link's value). They are special because they affect path resolution. For example, when resolving `/usr/lib/tex`, where `lib` is a symbolic link with value `/usr/local/lib`, once we reach `lib`, we restart the resolution with its value appending the remainder of the original path, i.e. `/usr/local/lib/tex`. The same problems as with paths that use '..' apply: paths that use symbolic links may overlap update.

First, we extend instrumented directories (definition 7) with symbolic links: $d ::= \ldots \mid a_\pi @ cap$, where $a_\pi @ cap$ is a symbolic link file named a with path value cap. Instrumented directory contexts (definition 12) and concrete directories (definition 1) are extended similarly. Next, we extend the fusion equivalences of definition 8 with:

$$\begin{array}{lll} a_\pi @ cap \otimes a_{\pi'} @ cap \equiv a_{\pi+\pi'} @ cap & & \text{if } \pi, \pi' > 0 \\ a_1 @ cap \otimes a_0[d] \quad \equiv_\wedge a_1 @ cap & & \end{array}$$

Lifting these extensions to the assertion language is straightforward. In order to relate tree footprints with paths as in the previous sections, we extend instrumented pathnames with a symbolic link component: $rp ::= \ldots \mid a_\pi @ ap$. This provides information about the footprint of paths with symbolic links. For example, given the instrumented pathname $\texttt{/usr}_{1/4}\texttt{/}(\texttt{lib}_1@\texttt{/usr}_{1/4}\texttt{/local}_{1/2}\texttt{/lib}_{1/2})\texttt{/tex}_1$ we know that `tex` is actually located in `/usr/local/lib`. Finally, we extend the pathname reduction used for the p/d notation with rules such as the following:

$$\begin{array}{c} (\top[c], (a_\pi @/rp')/rp) \downarrow (\top[(c \circ \varnothing) \otimes -], /rp'/rp) \\ \text{if } c \equiv c' \circ (a_\pi @/\, \mathsf{strippath}(rp') \otimes -) \end{array}$$

Note that symbolic link loops may occur in the directory structure, as in $\top[\texttt{usr}[\texttt{la}@\texttt{/usr/lb} \otimes \texttt{lb}@\texttt{/usr/la}]]$. However, by using the p/d notation, it is not possible to express paths with such loops. The path based tree footprint assertions used in our axioms are based on this notation. Therefore, paths used in our preconditions are always safe.

Crucially, we do not need to change our existing axioms, merely extend them with more commands such as `symlink` for creating symbolic links.

$$\begin{array}{c} \{\mathsf{var}(\texttt{r}, -) * \mathsf{var}(\texttt{target}, cp) * \mathsf{pvar}(\texttt{link}, p/a_1) * \mathsf{tree}(p/\widehat{a}_1)\} \\ \texttt{r := symlink(target, link)} \\ \left\{\begin{array}{c} \mathsf{var}(\texttt{r}, -) * \mathsf{var}(\texttt{target}, cp) * \mathsf{pvar}(\texttt{link}, p/a_1) \\ * \mathsf{tree}(p/a_1 @ cp) \end{array}\right\} \end{array}$$

One typical use of symbolic links is a form of versioning for installed software, where multiple versions of the same program may co-exist under different directories, e.g. `/usr/lib/gcc-4.8`, `/usr/lib/gcc-4.9`, with a symbolic link, e.g. `/usr/lib/gcc` pointing to the most recent version. Maintaining such structures is typically the responsibility of software installers. Using the extensions outlined in this section, we can reason about correctness properties of such software installers. For example, in appendix A we reason about a stylised installer using symbolic links to maintain multiple versions of the same software, proving that either it installs successfully and any previous installation remains intact, or leaves the file system unmodified if something goes wrong.

7. Conclusions and Future Work

We give a natural axiomatic specification for a core sequential fragment of the POSIX file system with arbitrary pathnames, concentrating on '..' but also summarising our extension to symbolic links. We study client programs such as the standard recursive remove utility and a software installer with version control. Our novel effect separating conjunction and effect frame rule, combined with shadow permissions, enable the effect of a local directory update to be propagated to the overlapping pathnames. We believe these ideas are fundamental and will be widely applicable to a variety of examples. Natural examples are graph algorithms and DOM querying, where we can compare our techniques with those based on sepish [17, 20]. We need to understand the trade-off between the two approaches. In future we aim to extend our reasoning to concurrent POSIX clients and relate our specifications with file-system implementations.

Acknowledgments

We acknowledge funding from an EPSRC DTA (Ntzik), EPSRC programme grants EP/H008373/1 (Gardner, Ntzik) and EP/K008528/1 (Gardner). We also thank Pedro da Rocha Pinto, Azalea Raad, Jose Fragoso Santos and the anonymous reviewers for their comments and suggestions. No new data was collected in the course of this research.

References

[1] POSIX.1-2008, IEEE 1003.1-2008, The Open Group Base Specifications Issue 7. URL `http://pubs.opengroup.org/onlinepubs/9699919799/`.

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

[3] N. Biri and D. Galmiche. Models and separation logics for resource trees. *Journal of Logic and Computation*, 17(4):687–726, 2007.

[4] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission Accounting in Separation Logic. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, 2005.

[5] R. Bornat, C. Calcagno, and H. Yang. Variables as Resource in Separation Logic. *Electronic Notes in Theoretical Computer Science*, 155:247–276, 2006.

[6] J. Boyland. Checking Interference with Fractional Permissions. In *Static Analysis*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003.

[7] C. Calcagno, P. Gardner, and U. Zarfaty. Context Logic and Tree Update. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, 2005.

[8] L. Cardelli and A. D. Gordon. Ambient logic. *Mathematical Structures in Computer Science*, 2003.

[9] P. da Rocha Pinto, T. Dinsdale-Young, M. Dodds, P. Gardner, and M. Wheelhouse. A Simple Abstraction for Complex Concurrent Indexes. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA, 2011.

[10] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent Abstract Predicates. In *ECOOP*. Springer Berlin Heidelberg, 2010.

[11] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: Compositional Reasoning for Concurrent Programs. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, 2013.

[12] M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-Guarantee Reasoning. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP, 2009.

[13] G. Ernst, G. Schellhorn, D. Haneberg, J. Pfähler, and W. Reif. Verification of a Virtual Filesystem Switch. In *Verified Software: Theories, Tools, Experiments*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014.

[14] K. Fisher, N. Foster, D. Walker, and K. Q. Zhu. Forest: A Language and Toolkit for Programming with Filestores. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP, 2011.

[15] 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.

[16] L. Freitas, J. Woodcock, and A. Butterfield. POSIX and the Verification Grand Challenge: A Roadmap. *2014 19th International Conference on Engineering of Complex Computer Systems*, 0:153–162, 2008.

[17] P. Gardner, S. Maffeis, and G. D. Smith. Towards a Program Logic for JavaScript. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, 2012.

[18] P. Gardner, G. Ntzik, and A. Wright. Local Reasoning for the POSIX File System. In *Programming Languages and Systems*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014.

[19] W. H. Hesselink and M. Lali. Formalizing a Hierarchical File System. *REFINE*, 2009.

[20] A. Hobor and J. Villard. The Ramifications of Sharing in Data Structures. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, 2013.

[21] R. Joshi and G. J. Holzmann. A Mini Challenge: Build a Verifiable Filesystem. *Formal Aspects of Computing*, 19(2): 269–272, 2007.

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

[23] G. Ntzik and P. Gardner. Reasoning about the POSIX File System: Local Update and Global Pathnames. Technical report, Imperial College London, 2015. URL `http://hdl.handle.net/10044/1/25816`.

[24] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, 2002.

A. Sofware Installer

We demonstrate reasoning about programs using symbolic links by considering a stylised software installer. New software is typically packaged and users either download it to their local file system, or obtain it from media containing a file system, e.g. a usb key. An installer program is responsible for correctly and safely placing the files of the package in the local file system. This may involve correctly dealing with existing installations. If a previous installation is detected, for example an older version of the software, some installers may refuse to proceed while others may completely remove it before installing their version. However, more advanced installers are capable of safely maintaining older versions

and install new ones. This typically involves the use of symbolic links as version agnostic "pointers" to version specific files and directories.

Here we give an example of this installation strategy by considering an installer for the fictional software "widget" with version number 2.0. This consists of an executable file, called "widget", and a data file called "data". Following common conventions the installer will place the data file inside the directory `/opt/widget`, and the executable file inside the directory `/usr/bin`. However, in order to maintain previous installations `/opt/widget` is a symbolic link to a directory specific to a previously installed version, e.g. `/opt/widget-1.8`. Similarly, `/usr/bin/widget` is a symbolic link to a version specific executable file, e.g. `/usr/bin/widget-1.8`. As such, the state the file system before the installer executes may include the following:

```
/usr/bin/widget-1.4
        /widget-1.8
        /widget@/usr/bin/widget-1.8
    /opt/widget-1.4
        /widget-1.8
        /widget@/opt/widget-1.8
```

Here, there are two versions of "widget" already installed: versions 1.4 and 1.8. The symbolic links `/usr/bin/widget` and `/opt/widget` point to the executable file and data directory of the most recent version respectively. Such a structure allows multiple versions of the same software to co-exist in isolation. Accessing the software through the symbolic links results in the most recent version being used, with the older versions still remaining accessible. This versioning strategy is widely used in many POSIX operating systems.

The implementation of an installer for "widget" 2.0 is given in figure 10. The installer assumes that the files to install are initially located in the directory given by the variable `il`, i.e. that is where the client downloaded the installer package to. The precondition:

$$P_i \triangleq \mathrm{W} \land \mathsf{var}(\mathtt{r}, -) * \mathsf{var}(\mathtt{errno}, -) * \mathsf{pvar}(\mathtt{il}, il) * src_{Pre} * bin_{Pre} * opt_{Pre}$$

captures the state of the file system in the variable W describes the initial states of variables, the installation source directory src_{Pre}, the `/usr/bin` directory bin_{Pre} and the program data directory opt_{Pre} with their definitions given in figure 8. Note that the precondition allows the state to be in a condition where installation is not safe. The file `/usr/bin/widget` may not exist, or it may not even be a symbolic link. The file `/usr/bin/widget-2.0` may already exist and even be a directory. Similarly for the contents of the `/opt` directory. Such cases may result from previously corrupt installations or some other client actions. The installer needs to query the file system and if it determines that installation is unsafe then it should stop and leave the file system unmodified. This is captured by the postcondition:

$$Q_i \triangleq \exists r, \iota', j'.\ \mathsf{var}(\mathtt{r}, r) * \mathsf{var}(\mathtt{errno}, -) \land (r = -1 \implies \mathrm{W}) \land \left(r = 0 \implies \begin{array}{c} src_{Post} * bin_{Post}(\iota') * opt_{Post}(j') \\ * files_{Post}(\iota', j') \end{array} \right)$$

If the installer returns -1 to the return variable `r`, then the file system state is that of the variable W from the precondition. Thus in this case, the file system is not modified. On the other hand, if the installer succeeds by returning 0, then the file system is described by the combination of the resources given in figure 9. Note that the pre- and postcondition do not mention any possible previous installation. Effectively, these are framed-off and thus remain unchanged.

In summary, the installer's specification:

$$\{P_i\}\ \mathtt{installWidget}\ \{Q_i\}$$

is stating two main properties: i) if the installer encounters a state where it cannot safely install, it will produce an error and leave the file system unmodified, and ii) any previous installation is guaranteed to remain intact.

Note that the implementation is using a variant of the `stat` command, called `lstat`. The difference is that if the path given to `stat` identifies a symbolic link, it follows it, whereas `lstat` does not. The specification for `lstat` is exactly the same as that of `stat` in figure 4, extended with the following axiom for handling a symbolic link:

$$\begin{array}{c} \{\mathsf{var}(\mathtt{t}, -) * \mathsf{pvar}(\mathtt{path}, p/a_\pi) * \mathsf{tree}(p/a_\pi @ cp)\} \\ \mathtt{t\ :=\ lstat(path)} \\ \{\mathsf{var}(\mathtt{t}, S) * \mathsf{pvar}(\mathtt{path}, p/a_\pi) * \mathsf{tree}(p/a_\pi @ cp)\} \end{array}$$

where in the postcondition it returns the value S to indicate that the path identifies a symbolic link file. The existing axioms for `stat`, need only be extended with the following axioms that follow the symbolic link at the end of the path:

$$\begin{array}{c} \{\mathsf{var}(\mathtt{t}, -) * \mathsf{pvar}(\mathtt{path}, p) * \mathsf{tree}(p/_S a_\pi : \iota)\} \\ \mathtt{t\ :=\ stat(path)} \\ \{\mathsf{var}(\mathtt{t}, F) * \mathsf{pvar}(\mathtt{path}, p) * \mathsf{tree}(p/_S a_\pi : \iota)\} \end{array}$$

$$\begin{array}{c} \{\mathsf{var}(\mathtt{t}, -) * \mathsf{pvar}(\mathtt{path}, p) * \mathsf{tree}(p/_S a_\pi[\varnothing])\} \\ \mathtt{t\ :=\ stat(path)} \\ \{\mathsf{var}(\mathtt{t}, D) * \mathsf{pvar}(\mathtt{path}, p) * \mathsf{tree}(p/_S a_\pi[\varnothing])\} \end{array}$$

where the $\mathsf{tree}(p/a_\pi/_S\Delta)$ is defined defined as:

$$\mathsf{tree}(p/a_\pi/_S\Delta) \triangleq \mathsf{tree}(p/a_\pi/\Delta) \lor \exists p'.\ \mathsf{tree}(p/a_\pi @ \mathsf{strippath}(p')) * \mathsf{tree}(p'/_S\Delta)$$

Furthermore, we use the command `fileCopy` to copy files with the following specification:

$$\begin{array}{c} \left\{ \begin{array}{c} \mathsf{var}(\mathtt{r}, -) * \mathsf{pvar}(\mathtt{source}, p/a_\pi) * \mathsf{pvar}(\mathtt{target}, p'/b_1) \\ * \mathsf{tree}(p/a_\pi : \iota) * \mathsf{tree}\left(p'/\widehat{b_1}\right) * \mathsf{file}(\iota, \beta) \end{array} \right\} \\ \mathtt{r\ :=\ fileCopy(source,\ target} \\ \left\{ \begin{array}{c} \exists \iota'.\ \mathsf{var}(\mathtt{r}, 0) * \mathsf{pvar}(\mathtt{source}, p/a_\pi) * \mathsf{pvar}(\mathtt{target}, p'/b_1) \\ * \mathsf{tree}(p/a_\pi : \iota) * \mathsf{tree}(p'/b_1 : \iota') * \mathsf{file}(\iota, \beta) * \mathsf{file}(\iota', \beta) \end{array} \right\} \end{array}$$

$$src_{Pre} \triangleq \mathsf{tree}(il/(\texttt{`widget'}_1 : \iota * \texttt{`data'}_1 : j)) * \mathsf{file}(\iota, \beta) * \mathsf{file}(j, \beta')$$
$$bin_{Pre} \triangleq \mathsf{tree}\left(/\texttt{`usr'}_{\pi_1}/\texttt{`bin'}_{\pi_2}/ \left(\begin{array}{c} \mathsf{ent}_1(\texttt{`widget'}) \vee \widehat{\texttt{`widget'}_1} \\ * \, \mathsf{ent}_1(\texttt{`widget-2.0'}) \vee \widehat{\texttt{`widget-2.0'}_1} \end{array} \right) \right)$$
$$opt_{Pre} \triangleq \mathsf{tree}\left(/\texttt{`opt'}_{\pi_3}/ \left( \begin{array}{c} \mathsf{ent}_1(\texttt{`widget'}) \vee \widehat{\texttt{`widget'}_1} \\ * \, \mathsf{ent}_1(\texttt{`widget-2.0'}) \vee \widehat{\texttt{`widget-2.0'}_1} \end{array} \right) \right)$$

Figure 8: Components of the installer's precondition.

$$src_{Post} \triangleq src_P$$
$$bin_{Post}(\iota) \triangleq \mathsf{tree}\left(/\texttt{`usr'}_{\pi_1}/\texttt{`bin'}_{\pi_2}/ \left(\begin{array}{c} \texttt{`widget'}_1 @ (/\texttt{`usr'}/\texttt{`bin'}/\texttt{`widget-2.0'}) \\ * \, \texttt{`widget-2.0'}_1 : \iota \end{array} \right) \right)$$
$$opt_{Post}(j) \triangleq \mathsf{tree}\left(/\texttt{`opt'}_{\pi_3}/ \left( \begin{array}{c} \texttt{`widget'}_1 @ (/\texttt{`opt'}/\texttt{`widget-2.0'}) \\ * \, \texttt{`widget-2.0'}_1 [\texttt{`data'}_1 : j] \end{array} \right) \right)$$
$$files_{Post}(\iota, j) \triangleq \mathsf{file}(\iota, \beta) * \mathsf{file}(j, \beta')$$

Figure 9: Components of installer's postcondition.

Our example installer, along with a proof sketch that it meets its specification: $\{P_i\}$ `installWidget` $\{Q_i\}$, is given in figure 10.

The installer expects that both `/usr/bin/widget` and `/opt/widget` are symbolic links, to some previous installation and we do not care which. The implementation initially checks whether this is true in lines 5 to 8, using the `lstat` specification defined earlier. If the paths do not identify symbolic links then the installer cannot proceed safely, e.g. assuming that a previous installation is now corrupt, and stops assigning -1 to the return variable. Next, the installer expects that `/opt/widget-2.0` and `/usr/bin/widget-2.0` do not exist since the former is the data directory of the new version it wants to install and the latter is an executable file. To check if this is true, the installer again relies on `lstat` to get the type of file identified by the paths in lines 12 to 15. If any of these paths identifies entries that exists, the installer stops without modifying anything. Otherwise, the installer can now safely install the new version in lines 19 to 28. First, it creates the new data directory `/opt/widget-2.0` and then proceeds to copy the executable file to the standard directory `/usr/bin` and the data file to the newly created data directory. Note that at this point in the proof we use the `fileCopy` specification given earlier. Finally, the installer needs to update the symbolic links `/usr/bin/widget` and `/opt/widget` to point the newly installed version. This is done by first deleting them with `unlink`, and then recreating them with the new path values using `symlink` as specified in section 6.

B. Soundness

Views [11] is a framework for proving the soundness of existing and new programming logics and type systems. By instantiating a set of parameters and establishing a set of properties the views framework provides a soundness result.

Here, we give a sketch proof of the soundness of our logic based on the views framework. For a more detailed proof we refer the reader to the technical report [23].

First, we summarise our parameterisation of the framework. The parameters are specialised to separation algebras.

Definition 16 (Views Parameters)**.** *The views framework requires the following parameters to be supplied:*

1. ***Atomic Commands:*** *The file system primitive commands, $\mathbb{C}_{fs}$, defined in section 3.4.*
2. ***Machine States:*** *The concrete program states,* STATES*, given in definition 6.*
3. ***Interpretation of Atomic Commands:*** *A function $[\![-]\!](-) : \mathbb{C}_{fs} \to \text{STATES} \to \mathcal{P}(\text{STATES})$ that associates each atomic command with a state transformer. We define this in the technical report [23].*
4. ***Separation Algebra:*** *The partial commutative monoid $(\text{ISTATES}, \bullet, (\emptyset, \emptyset, (\varnothing, \emptyset)))$, defined using disjoint function union for variables stores, process heaps and file heaps, and $\otimes$ for directory trees.*
5. ***Axiomatisation:*** *The axioms of figure 4, which we denote with the set $Axioms$.*
6. ***Reification:*** *The reification function in definition 10.*
7. ***Disjunction:*** *A total function, $\bigvee : \mathcal{P}(\mathcal{P}(\text{ISTATES})) \to \mathcal{P}(\text{ISTATES})$, which we define as set union.*

Views provides a general operational semantics in the form of an operational transition relation. It is parameterised by the interpretation function for primitive commands.

Definition 17 (Operational Semantics)**.** *The multi-step operational transition relation, $-, - \rightsquigarrow^* -, - : (\mathbb{C} \times \text{STATES}) \times$*

1: $\{P_i\}$
2: `r := installWidget` $\triangleq$
3: **local** `t1,t2` **in**
4: $\{\mathsf{var}(\texttt{t1}, -) * \mathsf{var}(\texttt{t2}, -) * \mathsf{var}(\texttt{errno}, -) * bin_{Pre} * opt_{Pre}\}$
5: `t1 := lstat('/usr/bin/widget');`
6: `t2 := lstat('/opt/widget');`
7: $\left\{\begin{array}{l} \mathsf{var}(\texttt{t1}, t1) \land t1 = D \lor F \lor S \lor -1 * \mathsf{var}(\texttt{t2}, t2) \land t2 = D \lor F \lor S \lor -1 \\ * \mathsf{var}(\texttt{errno}, e) \land (t1 = -1 \lor t2 = -1 \implies e = \texttt{E_NOENT}) * bin_{Pre} * opt_{Pre} \end{array}\right\}$
8: **if** `t1` $\neq S \lor$ `t2` $\neq S$ **then**
9: `r := -1`
10: **else**
11: $\left\{\begin{array}{c} \mathsf{var}(\texttt{t1}, S) * \mathsf{var}(\texttt{t2}, S) * \mathsf{var}(\texttt{errno}, -) \\ * \mathsf{tree}\Big(/\texttt{`usr'}_{\pi_1}/\texttt{`bin'}_{\pi_2}/((\exists cp.\texttt{`widget'}_1 @cp) * \mathsf{ent}_1(\texttt{`widget-2.0'}) \lor \widehat{\texttt{`widget-2.0'}_1})\Big) \\ * \mathsf{tree}\Big(/\texttt{`opt'}_{\pi_3}/((\exists cp.\texttt{`widget'}_1 @cp) * \mathsf{ent}_1(\texttt{`widget-2.0'}) \lor \widehat{\texttt{`widget-2.0'}_1})\Big) \end{array}\right\}$
12: `t1 := lstat('/opt/widget-2.0');`
13: `t2 := lstat('/usr/bin/widget-2.0');`
14: $\left\{\begin{array}{c} \mathsf{var}(\texttt{t1}, t1) \land t1 = D \lor F \lor S \lor -1 * \mathsf{var}(\texttt{t2}, t2) \land t2 = D \lor F \lor S \lor -1 \\ * \mathsf{var}(\texttt{errno}, e) \land (t1 = -1 \lor t2 = -1 \implies e = \texttt{E_NOENT}) \\ * \mathsf{tree}\Big(/\texttt{`usr'}_{\pi_1}/\texttt{`bin'}_{\pi_2}/((\exists cp.\texttt{`widget'}_1 @cp) * \mathsf{ent}_1(\texttt{`widget-2.0'}) \lor \widehat{\texttt{`widget-2.0'}_1})\Big) \\ * \mathsf{tree}\Big(/\texttt{`opt'}_{\pi_3}/((\exists cp.\texttt{`widget'}_1 @cp) * \mathsf{ent}_1(\texttt{`widget-2.0'}) \lor \widehat{\texttt{`widget-2.0'}_1})\Big) \end{array}\right\}$
15: **if** `t1` $\neq$ `-1` $\lor$ `t2` $\neq$ `-1` **then**
16: `r := -1`
17: **else**
18: $\left\{\begin{array}{c} \mathsf{var}(\texttt{t1}, -1) * \mathsf{var}(\texttt{t2}, -1) * \mathsf{var}(\texttt{errno}, -) \\ * \mathsf{tree}\Big(/\texttt{`usr'}_{\pi_1}/\texttt{`bin'}_{\pi_2}/((\exists cp.\texttt{`widget'}_1 @cp) * \widehat{\texttt{`widget-2.0'}_1})\Big) \\ * \mathsf{tree}\Big(/\texttt{`opt'}_{\pi_3}/((\exists cp.\texttt{`widget'}_1 @cp) * \widehat{\texttt{`widget-2.0'}_1})\Big) \end{array}\right\}$
19: `r := mkdir('/opt/widget-2.0')`
20: $\left\{\begin{array}{c} \mathsf{var}(\texttt{t1}, -1) * \mathsf{var}(\texttt{t2}, -1) * \mathsf{var}(\texttt{errno}, -) \\ * \mathsf{tree}\Big(/\texttt{`usr'}_{\pi_1}/\texttt{`bin'}_{\pi_2}/((\exists cp.\texttt{`widget'}_1 @cp) * \widehat{\texttt{`widget-2.0'}_1})\Big) \\ * \mathsf{tree}\Big(/\texttt{`opt'}_{\pi_3}/((\exists cp.\texttt{`widget'}_1 @cp) * \texttt{`widget-2.0'}_1[\varnothing])\Big) \end{array}\right\}$
21: `r := fileCopy(il/'widget', '/bin/widget-2.0');`
22: `r := fileCopy(il/'data', '/opt/widget-2.0/data');`
23: $\left\{\begin{array}{c} \exists \iota', j'.\ \mathsf{var}(\texttt{t1}, -1) * \mathsf{var}(\texttt{t2}, -1) * \mathsf{var}(\texttt{errno}, -) \\ * \mathsf{tree}\Big(/\texttt{`usr'}_{\pi_1}/\texttt{`bin'}_{\pi_2}/((\exists cp.\texttt{`widget'}_1 @cp) * \texttt{`widget-2.0'}_1 : \iota')\Big) \\ * \mathsf{tree}\Big(/\texttt{`opt'}_{\pi_3}/((\exists cp.\texttt{`widget'}_1 @cp) * \texttt{`widget-2.0'}_1[\texttt{`data'}_1 : j'])\Big) \\ * files_{Post}(\iota', j') \end{array}\right\}$
24: `r := unlink('/usr/bin/widget');`
25: `r := unlink('/opt/widget');`
26: $\left\{\begin{array}{c} \exists \iota', j'.\ \mathsf{var}(\texttt{t1}, -1) * \mathsf{var}(\texttt{t2}, -1) * \mathsf{var}(\texttt{errno}, -) \\ * \mathsf{tree}\Big(/\texttt{`usr'}_{\pi_1}/\texttt{`bin'}_{\pi_2}/(\widehat{\texttt{`widget'}_1} * \texttt{`widget-2.0'}_1 : \iota')\Big) \\ * \mathsf{tree}\Big(/\texttt{`opt'}_{\pi_3}/(\widehat{\texttt{`widget'}_1} * \texttt{`widget-2.0'}_1[\texttt{`data'}_1 : j'])\Big) \\ * files_{Post}(\iota', j') \end{array}\right\}$
27: `r := symlink('/usr/bin/widget-2.0', '/usr/bin/widget');`
28: `r := symlink('/opt/widget-2.0', '/opt/widget')`
29: $\{\exists \iota', j'.\ \mathsf{var}(\texttt{t1}, -1) * \mathsf{var}(\texttt{t2}, -1) * \mathsf{var}(\texttt{errno}, -) * bin_{Post} * opt_{Post} * files_{Post}(\iota', j')\}$
30: **fi**
31: **fi**
32: **end**
33: $\{Q_i\}$

Figure 10: Implementation and proof sketch of an installer for "widget" 2.0.

$(\mathbb{C} \times \text{STATES} \cup \{\lightning\})$, *relates programs and their initial states to either a terminal state or a distinguished* fault *state* $\lightning$, *where* $\lightning \notin$ STATES. *Command/state pairs not in the relation are deemed to be divergent.*

Given the parameters of definition 16, we establish the following properties. Again, these are specialised versions that are induced by separation algebras.

Definition 18 (Views Properties)**.** *The Views framework requires the following properties to be satisfied:*

1. ***Atomic Soundness:*** *It is sufficient to show that for every* $(P, \mathcal{A}, Q) \in$ AXIOMS, *where* $\mathcal{A} \in \mathbb{C}_{fs}$, *and every* $R \in$ ASRTS *and* $\mathrm{e} \in$ LENVS,
$$[\mathcal{A}]\left(\lfloor [\![P]\!]_{\mathrm{e}} \bullet [\![R]\!]_{\mathrm{e}} \rfloor\right) \subseteq \lfloor [\![Q]\!]_{\mathrm{e}} \bullet [\![R]\!]_{\mathrm{e}} \rfloor$$
2. ***Join Distributivity:*** $p \bullet \bigvee\{q_i\}_{i \in I} = \bigvee\{p \bullet q_i\}_{i \in I}$ *where* $p, q \in$ ISTATES.
3. ***Join Morphism:*** $\lfloor \bigvee\{p_i\}_{i \in I} \rfloor = \bigcup\{\lfloor p_i \rfloor\}_{i \in I}$ *where* $p \in$ ISTATES.

Having established these properties, the framework provides a soundness result for the program logic.

Theorem 2 (Soundness)**.** *Assume that* $\vdash \{P\}\mathbb{C}\{Q\}$ *is derivable in the program logic. Then, for all* $\mathrm{e} \in$ LENVS *and*

$st \in \lfloor [\![P]\!]_{\mathrm{e}} \rfloor$ *and* $st' \in \textsc{States}$, *if* $\mathbb{C}, st \rightsquigarrow^* \mathbf{skip}, st'$ *then* $st' \in \lfloor [\![Q]\!]_{\mathrm{e}} \rfloor$.

We establish the properties by the following two lemmas and refer the reader to the technical report [23] for the details of atomic soundness.

Lemma 1 (Atomic Soundness)**.** *The atomic soundness property holds. A detailed proof is given in the technical report [23].*

Lemma 2 (Join Distributivity and Morphism)**.** *Join distributivity and morphism hold in our instantiation of views.*

Proof. We instantiate the disjunction function as set union. Both properties follow trivially from the properties of set union and the reification of definition 10. □

The soundness of the effect frame rule is justified by showing it is an instance of the generalised frame rule of views and that it satisfies the required soundness property.

Definition 19 (Effect Frame Instantiating)**.** *We define the effect frame rule of our logic as an instance of the generalised frame rule of views:*

$$\frac{\vdash \{P\} \mathbb{C} \{Q\}}{\vdash \{f(P)\} \mathbb{C} \{f(Q)\}}$$

where we instantiate the views transforming function f *as:*

$$f_{q,r} \triangleq \lambda x.\ \textit{if}\ x = q\ \textit{then}\ x \mathbin{\hat{*}} r\ \textit{else}\ x * r$$

where the function chooses the normal separating conjunction for the precondition and the effect separating conjunction for the postcondition.

Lemma 3 (Effect Frame Soundness)**.** *From the views framework, the effect frame instance of the generalised frame rule is sound if the* f*-step preservation property holds.*

$$\begin{gathered} \forall P, Q, R, R' \in \textsc{Asrts}, \mathrm{e} \in \textsc{LEnvs}, \mathcal{A} \in \mathbb{C}_{fs}. \\ [\mathcal{A}]\left(\lfloor [\![P]\!]_{\mathrm{e}} \bullet [\![R]\!]_{\mathrm{e}} \rfloor\right) \subseteq \lfloor [\![Q]\!]_{\mathrm{e}} \bullet [\![R]\!]_{\mathrm{e}} \rfloor \\ \Longrightarrow \\ [\mathcal{A}]\left(\lfloor f_{Q,R'}([\![P]\!]_{\mathrm{e}} \bullet [\![R]\!]_{\mathrm{e}} \rfloor)\right) \subseteq \lfloor f_{Q,R'}([\![Q]\!]_{\mathrm{e}} \bullet [\![R]\!]_{\mathrm{e}}) \rfloor \end{gathered}$$

Proof. By atomic soundness and lemma 1 the top of the implication is true. By the transformation function, for the precondition we have:

$$\lfloor f_{Q,R'}([\![P]\!]_{\mathrm{e}} \bullet [\![R]\!]_{\mathrm{e}}) \rfloor = \lfloor [\![P * R * R']\!]_{\mathrm{e}} \rfloor$$

and for the postcondition we have:

$$\lfloor f_{Q,R'}([\![Q]\!]_{\mathrm{e}} \bullet [\![R]\!]_{\mathrm{e}}) \rfloor = \lfloor ([\![Q * R \mathbin{\hat{*}} R']\!]_{\mathrm{e}} \rfloor$$

Therefore it suffices to show that:

$$\begin{gathered} \forall P, Q, R \in \textsc{Asrts}, \mathcal{A} \in \mathbb{C}_{\mathrm{fs}}. \\ [\mathcal{A}]\left(\lfloor [\![P]\!]_{\mathrm{e}} \bullet [\![R]\!]_{\mathrm{e}} \rfloor\right) \subseteq \lfloor [\![Q \mathbin{\hat{*}} R]\!]_{\mathrm{e}} \rfloor \end{gathered}$$

which established similarly to lemma 1. □