

Reasoning about the POSIX File System

Local Update and Global Pathnames: Technical Report

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.

Keywords file systems, POSIX, 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 directories¹. 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

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.

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 `/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 appendix E.

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*:

$$\text{EFFECTFRAME} \frac{\vdash \{P\} \mathbb{C} \{Q\}}{\vdash \{P * R\} \mathbb{C} \{Q \hat{*} R\}}$$

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 [10]. 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 demonstrated in [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.

Figure 1 provides an example of structural separation logic abstract directory heaps. The left hand side illustrates a heap of a structured heap cell at address $\mathcal{F}$ containing a complete directory tree. In the right hand side, the directory tree in the heap cell $\mathcal{F}$ has become incomplete: the widget sub-directory has been allocated in its own heap cell at address x and its original location replaced with context hole x . The heap cell address x is associated with a path promise $/opt$ which provides the stability condition that body address x must be at the end of $/opt$. The path promise combines the local knowledge of the widget sub-directory with the global knowledge of its location in the directory structure.

The heap cell at address x satisfies the structural separation logic assertion $\alpha^{/opt} \mapsto \text{widget}[\text{data}[\emptyset] + \text{widget} : i]$, when the value of α is x , the path $/opt$ in the superscript is the path promise associated with α and $\text{widget}[\text{data}[\emptyset] + \text{widget} : i]$ is a context logic assertion describing directory contents. Such assertions form the basis of the local reasoning for file system proposed in [18], limited to the case when pathnames are linear (no use of `..` or symbolic links).

In the restricted case of linear paths, path promises can be trivially maintained by the environment. For example, in the right hand side of figure 1 the heap cell at address $\mathcal{F}$ has the implicit obligation to preserve the existence and the location

of context hole x , which is identified by a unique linear path (the one matching the promise). However, if we wish to have non-linear path promises we would require explicit obligation information in the environment. For example, consider the case where the path promise for heap cell x is the non linear path $/usr/./opt$. Then, the heap cell $\mathcal{F}$, not only has the (implicit) obligation to maintain the linear path to context hole x , but also the obligation to maintain the path $/$. 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, this proves to be extremely difficult. Consider the case where we wish to associate the x heap cell with the path $/opt/widget/././opt$, which goes inside the widget directory that is in the heap cell. It is not possible to generate a promise for this path under the general theory of structural separation logic [25], as promises are made only about the environment with their stability enforced by obligations. Relaxing this restriction, by allowing “self” promises, is not a solution because the promise would require the obligation that the widget directory is not changed, which would prohibit any kind of update to the heap cell. The problem lies in the fact that the system of promises/obligations of structural separation logic requires that the information captured by promises is disjoint from the update, which is not the case with arbitrary pathnames in POSIX file systems.

It seems that the only viable way for structural separation logic to be used with arbitrary non-linear paths is to amend heap cell assertions with predicates that combine the information about the global environment captured by promises with the local information captured by the cell’s contents to assert that there is enough resource to resolve a path. This would lead to assertions of the form $\alpha^q \mapsto d \wedge \text{resolve}(d, q, p)$, where $\text{resolve}(d, q, p)$ is a predicate saying that given the local data d and promise(s) q we know that the path p exists. However, even this is unsatisfactory because the information required by the *resolve* predicate cannot be confined to just the heap cell we want to associate a path with. Consider the case of a command updating two directories, e.g. the primitive rename command. The path identifying the first directory may require knowledge of the second directory and vice versa, which we cannot obtain through promises (we cannot make promises for things that are updated). Thus the resolve predicate for each path requires knowledge of both directories.

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

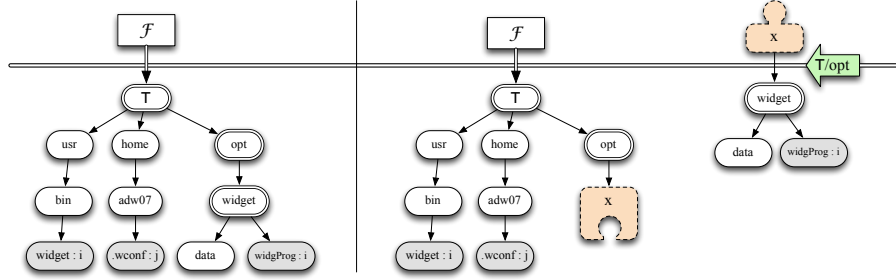


Figure 1: SSL reasoning

rename given in figure 5 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.

A natural question is whether techniques from fine-grained concurrent separation logics (such as deny-guarantee [12] and concurrent abstract predicates [11]), 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 $\text{in}_{\text{def}}(k, v)$, stating that a thread can update the key k with value v , can be used at the same time as the predicate $\text{read}(k)$, 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 $\text{read}(k)$ 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.

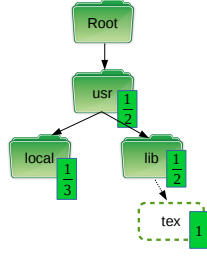
Finally, Biri and Galmiche propose a separation logic for simple tree updates with paths [3]. They use a similar tree model but without any permission structure to control resource sharing. This approach leads to non-local axioms and, as the authors admit, a frame rule that is only sound when allocation of new nodes is 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

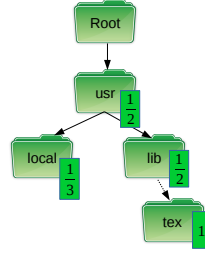
We focus on a sequential core fragment of POSIX file system commands. Even though small, it includes the basic primitive commands for manipulating the file system’s structure.

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.

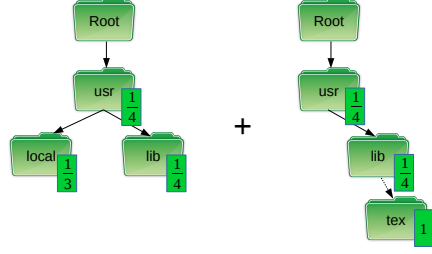
To do the update in our example, we require partial ownership of the entries in the directory tree that are required by the path `/usr/local/./lib`, and full knowledge and ownership that `tex` is not in the directory at the end of the path. To express this information, we instrument the entries of a directory tree with permissions and extend entries to also describe entry non-existence. Consider figure 2a, 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`



(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 2b by fusion.

Figure 2: Examples of partial directory trees.

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 2b 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 2a is $\text{tree}(\top[\text{usr}_{1/2}[\text{local}_{1/3}[\emptyset] * \text{lib}_{1/2}[\widehat{\text{tex}}_1]])$, where $\top$ is the root directory, $\widehat{\text{tex}}_1$ 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; [11] for non-entry). It is also convenient to use path syntax in tree assertions. For example, the assertion $\text{tree}(\text{usr}_{1/2}/\text{local}_{1/3}/\text{lib}_{1/2}/\widehat{\text{tex}}_1)$ is satisfied by the same partial directory tree as the previous assertion. The instrumented path syntax $\text{usr}_{1/2}/\text{local}_{1/3}/\text{lib}_{1/2}$ identifies the nodes of the partial directory tree and the order in which they are traversed. The $\widehat{\text{tex}}_1$ assertion is as before.

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

$$\begin{aligned} & \{ \text{var}(\mathbf{r}, -) * \text{pvar}(\text{path}, p/a_1) * \text{tree}(p/\widehat{a}_1) \} \\ & \quad \mathbf{r} := \text{mkdir}(\text{path}) \\ & \{ \text{var}(\mathbf{r}, 0) * \text{pvar}(\text{path}, p/a_1) * \text{tree}(p/a_1[\emptyset]) \} \end{aligned}$$

In the precondition, the variable assertion $\text{var}(\mathbf{r}, -)$ states that the program variable $\mathbf{r}$, 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 $\text{pvar}(\text{path}, p/a_1)$ states that the program variable `path` has a value given by the path p/a_1 . The tree assertion $\text{tree}(p/\widehat{a}_1)$ describes a partial directory tree where the nodes in the tree are described by path p and the non-entry $\widehat{a}_1$ has full permission and hence can be updated. In the postcondition, the tree assertion $\text{tree}(p/a_1[\emptyset])$ states that, under p , we now have an empty directory named $\widehat{a}_1$ 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 $\mathbf{r} := \text{rmdir}(\text{path})$ 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 2b. The command will remove the directory `tex` that we had previously created. However, with this path, the tree in figure 2b 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 2c illustrates how the instrumented directory tree in figure 2b 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 2b can be described by the assertion given by path syntax:

$$\text{tree}(\text{usr}_{1/2}/\text{local}_{1/3}/\text{'..'}/\text{lib}_{1/2}/\text{tex}_1[\emptyset])$$

This can be rewritten as the following assertion, satisfied by the partial directory trees in figure 2c:

$$\text{tree}(\text{usr}_{1/4}/\text{local}_{1/3}/\text{'..'}/\text{lib}_{1/4}[\emptyset]) \\ * \text{tree}(\text{usr}_{1/4}/\text{lib}_{1/4}/\text{tex}_1[\emptyset])$$

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 2a.

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 2c. 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 `tex0` 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 2c:

$$\text{tree}(\text{usr}_{1/4}/\text{lib}_{1/4}/\text{tex}_0/\text{'..'}/\text{tex}_1[\emptyset])$$

which is equivalent to the assertion:

$$\text{tree}(\top[\text{usr}_{1/4}[\text{lib}_{1/4}[\text{tex}_0[\emptyset] * \widehat{\text{tex}}_1]]])$$

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 `tex1[ $\emptyset$ ]` 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:

$$\text{tree}(\top[\text{usr}_{1/4}[\text{lib}_{1/4}[\text{tex}_0[\emptyset] \hat{*} \widehat{\text{tex}}_1]]])$$

The composition `tex0[ $\emptyset$ ]  $\hat{*}$   $\widehat{\text{tex}}_1$` propagates the effect of the update by discarding the now outdated `tex0[ $\emptyset$ ]`, leading to the assertion `tree( $\top[\text{usr}_{1/4}[\text{lib}_{1/4}[\widehat{\text{tex}}_1]]])$` .

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

$$\{\text{var}(\mathbf{r}, -) * \text{pvar}(\text{path}, p/a_1) * \text{tree}(p/a_1[\emptyset])\} \\ \mathbf{r} := \text{rmdir}(\text{path}) \\ \{\text{var}(\mathbf{r}, 0) * \text{pvar}(\text{path}, p/a_1) * \text{tree}(p/\widehat{a}_1)\}$$

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:

$$\text{EFFECTFRAME} \frac{\vdash \{P\} \mathcal{C} \{Q\}}{\vdash \{P * R\} \mathcal{C} \{Q \hat{*} R\}}$$

This rule is an instance of the generalised frame rule of views [10]. We prove soundness by appealing to views, as summarised in section 4.3 and given in detail in appendix B.

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

$$\text{tree}(\text{usr}_{1/4}/\text{lib}_{1/4}/\text{tex}_1[\emptyset]) \\ * \text{tree}(\text{usr}_{1/4}/\text{lib}_{1/4}/\text{tex}_0/\text{'..'}/\text{'..'}/\text{local}_{1/3}[\emptyset])$$

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

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

$$\begin{array}{l} \left\{ \begin{array}{l} \text{var}(\mathbf{r}, -) * \text{pvar}(\text{path}, p/\text{tex}_1) * \text{tree}(p/\text{tex}_1[\emptyset]) \\ * \text{tree}(p/\text{tex}_0/\text{'..'}/\text{'..'}/\text{local}_{1/3}[\emptyset]) \end{array} \right\} \\ \text{EFFECT FRAME} \left\{ \begin{array}{l} \text{var}(\mathbf{r}, -) * \text{pvar}(\text{path}, p/\text{tex}_1) * \text{tree}(p/\text{tex}_1[\emptyset]) \\ // \text{ apply the rmdir axiom} \\ \mathbf{r} := \text{rmdir}(\text{path}) \\ \text{var}(\mathbf{r}, 0) * \text{pvar}(\text{path}, p/\text{tex}_1) * \text{tree}(p/\widehat{\text{tex}}_1) \\ // \text{ consequence rule:} \\ // \text{ propagate effect on the path} \\ \text{var}(\mathbf{r}, 0) * \text{pvar}(\text{path}, p/\text{tex}_1) * \text{tree}(p/\widehat{\text{tex}}_1) \\ \left\{ \begin{array}{l} \text{var}(\mathbf{r}, 0) * \text{pvar}(\text{path}, p/\text{tex}_1) * \text{tree}(p/\widehat{\text{tex}}_1) \\ \hat{*} \text{tree}(p/\text{tex}_0/\text{'..'}/\text{'..'}/\text{local}_{1/3}[\emptyset]) \end{array} \right\} \\ // \text{ consequence rule:} \\ // \text{ propagate effect on the recombined resource} \\ \text{var}(\mathbf{r}, 0) * \text{pvar}(\text{path}, p/\text{tex}_1) * \text{tree}(p/\widehat{\text{tex}}_1) \\ * \text{tree}(p/\text{'..'}/\text{local}_{1/3}[\emptyset]) \end{array} \right\} \end{array}$$

In the precondition, the assertion `tree( $p/\text{tex}_1[\emptyset]$ )` describes all the resource required by `rmdir`. We therefore frame-off the assertion `tree( $\text{'..'}/\text{'..'}/\text{local}_{1/3}[\emptyset]$ )`. 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 `tree( $p/\widehat{\text{tex}}_1$ )`. Note the use of $\widehat{\text{}}$ 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 `tree( $p/\widehat{\text{tex}}_1$ )`. Next, we frame-on the assertion we had previously set aside, using the effect separating conjunction $\hat{*}$. Using the consequence rule, we can remove `tex0` 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 $\text{tree}(p/\hat{a}_1)$ 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 empty². 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:

$$\left\{ \begin{array}{l} \text{var}(r, -) * \text{pvar}(\text{old}, p/a_1) * \text{pvar}(\text{new}, p'/b_1) \\ * \text{tree}(p/a_1[d]) * \text{tree}(p'/b_1[\emptyset]) \end{array} \right\} \\ r := \text{rename}(\text{old}, \text{new}) \\ \left\{ \begin{array}{l} \text{var}(r, 0) * \text{pvar}(\text{old}, p/a_1) * \text{pvar}(\text{new}, p'/b_1) \\ * \text{tree}(p/\hat{a}_1) * \text{tree}(p'/b_1[d]) \end{array} \right\}$$

In the precondition, $\text{tree}(p/a_1[d])$ 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 $\text{tree}(p'/b_1[\emptyset])$ 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, $\text{tree}(p/\hat{a}_1)$ ensures that effects are propagated to p . The assertion $\text{tree}(p'/b_1[d])$ 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 $*$ 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.

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

3.1 File System

The file system structure is a directed acyclic graph consisting of a directory tree and files³. Files are uniquely identified by *inodes* from the set `INODES`, ranged over by $\iota, k, \dots$. We use the set `FNAMES`, ranged over by $a, b, \dots$, for naming both files and directories.

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

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

where $\emptyset$ 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 $\emptyset$. Well-formed directory forests have sibling distinct names. We only consider concrete directory forests that are well formed.

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

A regular file stores arbitrary binary content in the form of a sequence of bytes. As file IO involves copying of bytes between the process heap and the file it is convenient to use a heap style model for file contents. We define the contents of a file as heap from file offsets, counting the position of a single byte from the beginning of a file, to individual bytes. Note that POSIX files may have gaps. We indicate these gaps with a special distinguished value not in the set of bytes.

Definition 2 (File Byte Heaps). *Assume $\text{BYTES} = [0, 2^8]$ be the set of bytes, ranged over by α, β . The contents of a file are a heap of bytes, $\text{fh} \in \text{FBYTEHEAPS}$, mapping naturals to bytes or to the special value $\emptyset \notin \text{BYTES}$ indicating a gap.*

$$\text{fh} : \mathbb{N} \xrightarrow{\text{fin}} \text{BYTES} \cup \{\emptyset\}$$

A file byte heap is well formed if its domain is a consecutive subset of naturals: $\exists n \in \mathbb{N}. \text{dom}(\text{fh}) = [0, n]$. Throughout this paper we work with well formed byte heaps.

Definition 3 (File Heaps). *A file heap, $f \in \text{FILES}$, is a function mapping inodes to file byte heaps.*

$$f : \text{INODES} \xrightarrow{\text{fin}} \text{FBYTEHEAPS}$$

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 4 (File Systems). *File systems are defined as:*

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

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

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

Definition 5 (Concrete Paths). A concrete relative path, $\text{crp} \in \text{RELPATHS}$, is a sequence of filenames defined by:

$$\text{crp} ::= ' ' \mid ' ' \mid a \mid ' ' / \text{crp} \mid ' ' / \text{crp} \mid a / \text{crp}$$

with an equivalence relation, $\text{crp} \equiv \text{crp}'$, stating that $/$ is associative with $' '$ as identity. Absolute paths $\text{ABSPATHS} \triangleq \{ / \text{crp} \mid \text{crp} \in \text{RELPATHS} \}$, are paths that begin at the root directory. Equivalence is extended to absolute paths: $/ \text{crp} \equiv / \text{crp}' \iff \text{crp} \equiv \text{crp}'$. 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

The 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 commands: directory streams and open file descriptions. It also contains a standard program heap.

An open file description is a record holding information that controls file accesses: the inode and current offset of an open file. The commands `read`, `write`, `lseek` and `close` operate on open file descriptions. For example, every time a path is opened with `open`, a fresh *file descriptor* is allocated to store a new description that records the resolved inode and an initial 0 offset. Calls to `read` obtain bytes from the file identified by the inode, beginning at the recorded offset. The `lseek` command manipulates the offset.

Definition 6 (Open File Descriptions). An open file description is a pair of inode number identifying an open file and an offset.

$$\text{FILEDESCRIPTORS} \triangleq \text{INODES} \times \mathbb{N}$$

Let FDS be the (countable) set of file descriptors. An open file heap, $\text{of} \in \text{OFHEAPS}$, is a finite partial function mapping file descriptors to open file descriptions.

$$\text{of} : \text{FDS} \xrightarrow{\text{fin}} \text{FILEDESCRIPTORS}$$

A directory stream is an abstract data structure that captures the set of the entries in given directory and supports the `opendir`, `readdir` and `closedir` commands. For example, when `opendir(p)` is used, a fresh *directory stream address* is allocated and mapped to a directory stream, which provides a snapshot of the entries in the directory p .

Definition 7 (Directory Streams). A directory stream is a set of file names.

$$\text{DIRSTREAMS} \triangleq \mathcal{P}(\text{FNAMES})$$

Let DSADDRS be the (countable) set of directory stream addresses. An open directory heap, $\text{od} \in \text{ODHEAPS}$, is a finite partial function mapping directory stream addresses to directory streams.

$$\text{od} : \text{DSADDRS} \xrightarrow{\text{fin}} \text{DIRSTREAMS}$$

Our model of directory streams is a minor deviation from POSIX. The commands `opendir` and `readdir` return the names of entries contained within a directory. POSIX allows a high degree of non-determinism when using `readdir` on a directory whilst altering the set of entries. One may see some changes, all changes, or none. We currently adopt a *snapshot semantics*. A copy of the set of entries is taken when `opendir` is used. Calls to `readdir` return elements of that set in a non-deterministic order, but will observe no future changes to the set of entries.

A process can also manipulate normal heap memory. Traditional separation logic heaps map natural numbers to integers. Here, we restrict the codomain of process memory heaps to bytes, as in the file byte heaps of definition 2. This is both more realistic, as in real systems individual heap addresses point to a single byte, and more convenient for interacting with file byte heaps.

Definition 8 (Process Byte Heaps). A process byte heap, $\text{h} \in \text{PBYTEHEAPS}$, is a finite partial function mapping naturals to bytes

$$\text{h} : \mathbb{N} \xrightarrow{\text{fin}} \text{BYTES}$$

Definition 9 (Process Heaps). Process heaps, $\text{ph} \in \text{PH}$, are the disjoint union of open file heaps, open directory heaps and process byte heaps.

$$\text{PH} \triangleq \text{OFHEAPS} \uplus \text{ODHEAPS} \uplus \text{PBYTEHEAPS}$$

3.3 Variable Stores

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

Definition 10 (Program States). A program state, $\text{st} \in \text{STATES}$, consists of a variable store, a process heap and a file system: $\text{STATES} \triangleq \Sigma \times \text{PH} \times \text{FS}$.

3.4 Programming Language

We reason about programs written in a sequential WHILE language with calls to POSIX file system commands. The language uses *program expressions*, $\text{Expr} \in \text{EXPR}$, which consist of value literals, variables, arithmetic and boolean operations and path composition with $/$. Expression evaluation $\llbracket - \rrbracket : \text{EXPR} \rightarrow \Sigma \rightarrow \text{VALS}$ is standard. In this paper we consider the following fragment of POSIX file system

commands, $\mathbb{C}_{fs}$:

```

r := mkdir(path) | r := rmdir(path)
r := unlink(path) | r := rename(old, new)
 $\mathbb{C}_{fs} ::=$  | dir := opendir(path) | closedir(dir)
| fn := readdir(dir) | t := stat(path)
| p := realpath(path)

```

In POSIX, the commands are defined as C function interfaces. Here, we adapt them to our simple language, abstracting the details of C. The programming language is given by the following grammar:

```

var := Expr | var := [Expr] | [Expr] := Expr
ptr := malloc(size) | free(ptr)
 $\mathbb{C} ::=$  | local var in  $\mathbb{C}$  end | if Expr then  $\mathbb{C}$  else  $\mathbb{C}$  fi
| while Expr do  $\mathbb{C}$  done | skip |  $\mathbb{C}; \mathbb{C}$  | x := Expr |  $\mathbb{C}_{fs}$ 

```

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 [10], 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 [11], we add further instrumentation to *locally* describe the non-existence of entries. Otherwise, we would have to inspect all the entries in a directory to determine if something does not exist, leading to bigger footprints.

Definition 11 (Instrumented Directories). Instrumented directory forests, $d \in \text{IDIRS}$, are defined by:

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

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 accordingly⁴. 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, $a_{1/2}[\widehat{\langle \text{FNAMES} \rangle}_{1/3}]$ describes an empty directory because we know all the entry names do not exist. In contrast, $a_{1/2}[\emptyset]$, 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 $\emptyset$, 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 12 (Fusion equivalences). The fusion equivalence relation, $\equiv \subseteq \text{IDIRS} \times \text{IDIRS}$, is defined by the axioms:

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

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

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

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

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

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

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

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

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

$$a_1[d] \otimes a_0[d'] \equiv_\wedge a_1[d] \quad \text{if } d \otimes d' \neq d \quad (8)$$

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

⁴ Definition 11 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.

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.

To allow fine grained reasoning about file IO, we also instrument file byte heaps and file heaps. We instrument the file byte heaps of definition 2 by adding a distinguished value indicating the end of file. Beyond this end-of-file mark no file offset is allowed to be allocated in the byte heap.

Definition 13 (Instrumented File Byte Heaps). *An instrumented file byte heap, $\text{ifh} \in \text{IFBYTEHEAPS}$, is a finite partial function mapping naturals to bytes or the special value, $\emptyset$, indicating a gap, or the special value, $\exists$, indicating the end-of-file.*

$$\text{ifh} : \mathbb{N} \xrightarrow{\text{fin}} \text{BYTES} \cup \{\emptyset, \exists\}$$

An instrumented file byte heap is well formed when nothing is allocated beyond the end-of-file. There can be only one end-of-file value.

$$\begin{aligned} \text{wfb}(\text{ifh}) &\xleftrightarrow{\text{def}} \\ \exists n \in \text{dom}(\text{ifh}). \text{ifh}(n) = \exists &\implies \nexists m > n. m \in \text{dom}(\text{ifh}) \\ \wedge \\ \forall n, m \in \text{dom}(\text{ifh}). \text{ifh}(n) = \text{ifh}(m) = \exists &\implies n = m \end{aligned}$$

An instrumented byte heap is complete when for some integer address $n \geq 0$ all addresses less than n map to bytes or gaps and n maps to the end-of-file. Composition of instrumented file byte heaps is a standard disjoint function union, but only when the result is well formed with respect to the end-of-file.

Definition 14 (Instrumented File Byte Heap Composition). *Instrumented byte heap composition is a partial binary operator defined as the disjoint union of instrumented byte heaps as long as the result is well formed.*

$$\begin{aligned} \diamond : \text{IFBYTEHEAPS} &\rightarrow \text{IFBYTEHEAPS} \rightarrow \text{IFBYTEHEAPS} \\ \text{ifh}_1 \diamond \text{ifh}_2 &\triangleq \text{ifh}_1 \uplus \text{ifh}_2 \quad \text{iff } \text{wfb}(\text{ifh}_1 \uplus \text{ifh}_2) \end{aligned}$$

The operator is associative and commutative with unit $\{\} \mapsto \{\}$.

Instrumented file heaps are just file heaps as in definition 3 but with instrumented byte heaps.

Definition 15 (Instrumented File Heaps). *An instrumented file heap, $\text{if} \in \text{IFILEHEAPS}$ is a finite partial function mapping inode numbers to instrumented byte heaps.*

$$\text{if} : \text{INODES} \xrightarrow{\text{fin}} \text{IFBYTEHEAPS}$$

Definition 16 (Instrumented File Heap Composition). *Instrumented file heap composition is a partial binary operator on instrumented file heaps.*

$$\begin{aligned} \diamond : \text{IFILEHEAPS} &\rightarrow \text{IFILEHEAPS} \rightarrow \text{IFILEHEAPS} \\ \text{if}_1 \diamond \text{if}_2 &= \text{if} \\ &\xleftrightarrow{\text{def}} \\ \forall \iota, j. \iota \in \text{dom}(\text{if}_1) \wedge j \notin \text{dom}(\text{if}_1) \wedge \iota \notin \text{dom}(\text{if}_2) \wedge j \in \text{dom}(\text{if}_2) \\ &\quad \wedge \text{if}_1(\iota) = \text{if}(\iota) \wedge \text{if}_2(j) = \text{if}(j) \\ &\quad \wedge \\ &\quad \forall \iota. \iota \in \text{dom}(\text{if}_1) \wedge \iota \in \text{dom}(\text{if}_2) \\ &\quad \wedge \text{if}_1(\iota) = \text{ifh}_1 \wedge \text{if}_2(\iota) = \text{ifh}_2 \wedge \text{if}(\iota) = \text{ifh}_1 \diamond \text{ifh}_2 \end{aligned}$$

The operator is associative and commutative with unit $\{\} \mapsto \{\}$.

An instrumented file heap is complete when it contains complete byte heaps. A complete instrumented file heap can be related to a concrete byte heap of definition 3 by stripping away the instrumentation.

Definition 17 (File Heap Instrumentation Stripping). *A complete instrumented file heap can be viewed as a non-instrumented file heap by stripping away the end-of-file mapping.*

$$\begin{aligned} \text{stripfh} : \text{IFILEHEAPS} &\rightarrow \text{FILES} \\ \text{stripfh}(\text{if}) &= \text{f} \\ &\xleftrightarrow{\text{def}} \\ \forall \iota \in \text{dom}(\text{if}). \exists \text{ifh}. \text{if}(\iota) = \text{ifh} \\ &\quad \wedge \\ &\quad \exists n > 0. \forall n' \leq n. \\ &\quad \text{ifh}(n') \in \text{BYTES} \cup \{\emptyset\} \wedge \text{ifh}(n+1) = \exists \wedge \text{f}(\iota) = \text{ifh} \upharpoonright [0, n] \\ &\quad \wedge \\ &\quad (\text{dom}(\text{ifh}) = \{0\} \wedge \text{ifh}(0) = \exists \wedge \text{f}(\iota) = \{\} \mapsto \{\}) \end{aligned}$$

A partial instrumented file heap can be related to a set of concrete file heaps by considering all possible completions. An instrumented file heap completes another when if composed together they form a complete instrumented file heap.

Definition 18 (Instrumented File Heap Completion Set). *The function $\text{completions}_F : \text{IFILEHEAPS} \rightarrow \text{FILES}$ returns the set of all possible non-instrumented file heaps completed from the given instrumented file heap.*

$$\begin{aligned} \text{completions}_F(\text{if}) & \\ &\triangleq \\ \{\text{f} \in \text{FILES} \mid \text{if}' \in \text{IFILEHEAPS}. \text{stripfh}(\text{if} \diamond \text{if}') = \text{f}\} \end{aligned}$$

Definition 19 (Instrumented Program States). *Instrumented filesystems are defined as: $\text{IFS} \triangleq \text{IDTREES} \times \text{IFILEHEAPS}$. 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 20 (Reification). *Instrumented program states are reified to program states via the reification function, $[-] : \text{ISTATES} \rightarrow \mathcal{P}(\text{STATES})$, defined as:*

$$\{(\sigma, \text{ph}, (\top[\text{cd}], \text{f}')) \mid \text{strip}(d) = \text{cd} \wedge \text{f}' \in \text{completions}_F(\text{f})\}$$

where the function $\text{strip} : \text{IDIRS} \rightarrow \text{DIRS}$ is defined by stripping permissions and discarding non-existing entries when the result is a well-formed directory forest.

Reification expects that all outstanding effects have been propagated 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 4.

Finally, we also instrument pathnames to use them as convenient syntax for describing the tree footprints of paths used by programs as described in section 2. The instrumentation simply adds permissions to the entry name path component.

Definition 21 (Instrumented Pathnames). *Instrumented relative pathnames, $rp \in \text{IRELPATHS}$, 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, $\text{IABSPATHS} \triangleq \{./rp \mid rp \in \text{IRELPATHS}\}$, 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 $\text{IPATHS} \triangleq \text{IRELPATHS} \cup \text{IABSPATHS}$.

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

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

where $\equiv_{..}$ extends $\equiv$ on paths with $a_\pi / .. / rp \equiv_{..} rp$ and $./rp \equiv_{..} ./rp$. Finally, we use the function $\text{strippath} : \text{IPATHS} \rightarrow \text{PATHS}$, 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 2. 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 $./\text{usr}_{1/2}/\text{local}_{1/3}/../\text{lib}_{1/2}/\text{tex}_1[\emptyset]$ the absolute instrumented path $./\text{usr}_{1/2}/\text{local}_{1/3}/../\text{lib}_{1/2}$ corresponds to the path footprint and the instrumented directory $\text{tex}_1[\emptyset]$ corresponds to the update footprint. The path footprint is the natural one-holed context arising from the path. For example, the instrumented path $./\text{usr}_{1/2}/\text{local}_{1/3}$ gives rise to the path footprint $\top[\text{usr}_{1/2}[\text{local}_{1/3}[-]]]$, where $-$ is the

context hole, and $./\text{usr}_{1/2}/\text{local}_{1/3}/../\text{lib}_{1/2}$ gives rise to the path footprint $\top[\text{usr}_{1/2}[\text{local}_{1/3}[\emptyset] \otimes \text{lib}_{1/2}[-]]]$. The notation $./\text{usr}_{1/2}/\text{local}_{1/3}/../\text{lib}_{1/2}/\text{tex}_1[\emptyset]$ thus describes the tree footprint given by the composition of the path footprint context with the update footprint directory:

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

Definition 22. *One-holed instrumented directory contexts, $c \in \text{ICDIRS}$, are defined by*

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

with an equivalence relation, $c \equiv c'$, stating that $\otimes$ is associative and commutative with identity $\emptyset$. The same well-formedness conditions of instrumented directories (definition 11) apply. Context application, $\circ : \text{ICDIRS} \times \text{ICDIRS} \rightarrow \text{ICDIRS}$, 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 23 (Path-to-context reduction).

$$\downarrow \subseteq (\text{ICDIRS} \times \text{IPATHS}) \times (\text{ICDIRS} \times \text{IPATHS})$$

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

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 /usr_{1/2}/local_{1/3}/../lib_{1/2}:

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

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 24 (Path/tree notation).

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

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 $../\emptyset$ is not defined, but $\text{usr}_{1/2}/\text{local}_{1/3}/../\emptyset$ 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, $e \in \text{LENVS}$, to map logical variables to the extended values. Logical expressions, $E \in \text{LEXPRS}$, 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, $(\llbracket - \rrbracket_- : \text{LEXPRS} \rightarrow \text{LENVS} \rightarrow \text{VALS})$, is standard. We treat all variables in assertions that are not program variables as logical variables. We slightly abuse notation and use $p, q, \dots$ for logical instrumented path variables, $a, b, \dots$ for logical entry name variables, $\iota, j, \dots$ for logical inode variables and $d, d', \dots$ for logical instrumented directory tree variables.

Assertions $P, Q \in \text{ASRTS}$ are constructed from standard logical connectives and quantifiers and the left-hand-side assertions of figure 3. The assertion $\text{tree}(E/\Delta)$ 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 $\text{tree}(E//\Delta)$ 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 tree($\top[\Delta]$) which describes the root directory with contents given by the directory assertion Δ . The assertion $\text{fd}(\text{fd}, \iota, E)$ describes the open file description associated with the file descriptor given by the program variable fd , for the file with inode ι and offset given by the value of the logical expression E . The assertion $\text{fh}(\iota, E, E')$ describes the a file with inode ι which at offset given by the value of the logical expression E has a byte value given by E' . The assertion $\text{ds}(\text{dir}, E)$ describes a directory stream with address dir containing the set of unread entries given by the logical expression E . The assertion $\text{ptr}(x, E)$ describes a heap cell at address given by the program variable x with value given by E . The assertion $\text{var}(x, E)$ 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* $P \hat{*} Q$ extends $*$, by propagating effects of P and Q to each other, with emp being the identity of both $*$ and $\hat{*}$. Finally, directory tree assertions, $\Delta \in \text{IDIRECTORYASRTS}$, are constructed from standard logical connectives and quantifiers and the right-hand-side assertions of figure 3. These essentially lift instrumented directory trees to the assertion language. We overload $*$ and $\hat{*}$ to describe fusion (definition 12). Instrumented program states satisfying the assertions are given by satisfaction relations, $\models : (\text{LENVS} \times \text{ISTATES}) \times \text{ASRTS}$, and, $\models' : (\text{LENVS} \times \text{IDIRS}) \times \text{IDIRECTORYASRTS}$, defined in figure 4, 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 $\hat{*}$: $P * Q \implies P \hat{*} Q$. 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 12. These are directly lifted to the logic. For example, from equation (6) in definition 12 we have the logical axiom: $\langle \widehat{\{a\}} \rangle_1 \hat{*} a_0[d] \iff \langle \widehat{\{a\}} \rangle_1$.

We also define the following derived assertions in our specifications:

$$\begin{aligned}
& \textbf{Derived assertions} \\
\hat{a}_\pi &\triangleq \langle \widehat{\{a\}} \rangle_\pi & \text{ent}_\pi(a) &\triangleq (\exists \iota. a_\pi : \iota) \vee a_\pi[\text{true}] \\
\bigotimes_{x \in \emptyset} \Delta &\triangleq \emptyset & \bigotimes_{x \in \{n\}} \Delta &\triangleq \Delta[n/x] \\
\bigotimes_{x \in \{n\} \uplus S} \Delta &\triangleq \Delta[n/x] \bigotimes_{x \in S} \Delta
\end{aligned}$$

Assertions P, Q		Directory Assertions Δ, Δ'	
Directory Tree	$\text{tree}(E/\Delta)$	File Type Entry	$E_{E'} : \iota$
Effect Directory Tree	$\text{tree}(E//\Delta)$	Directory Type Entry	$E_{E'}[\Delta]$
Root Directory	$\text{tree}(\top[\Delta])$	Non Existing Entries	$\langle E \rangle_{E'}$
File Descriptor Cell	$\text{fd}(\text{fd}, \iota, E)$	Fusion	$\Delta * \Delta'$
File Heap Cell	$\text{fh}(\iota, E, E)$	Effect fusion	$\Delta \hat{*} \Delta'$
Directory Stream	$\text{ds}(\text{dir}, E)$	Logical Expression	E
Heap Cell	$\text{ptr}(\mathbf{x}, E)$	Empty Entry	$\emptyset$
Program Variable Value	$\text{var}(\mathbf{x}, E)$		
Separating Conjunction	$P * Q$		
Effect Separating Conjunction	$P \hat{*} Q$		
Logical Expression	E		
Empty	emp		

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

Figure 3: Fusion logic assertions.

$$\begin{aligned}
e, (\emptyset, \emptyset, (\top[d], \emptyset)) \models \text{tree}(E/\Delta) &\iff \exists d', d''. e, d' \models' \Delta \wedge \top[d''] = \langle E \rangle_e / d' \wedge d'' \equiv d \\
e, (\emptyset, \emptyset, (\top[d], \emptyset)) \models \text{tree}(E//\Delta) &\iff \exists d', d''. e, d' \models' \Delta \wedge \top[d''] = \langle E \rangle_e / d' \wedge d'' \equiv_{\wedge} d \\
e, (\emptyset, \emptyset, (\top[d], \emptyset)) \models \text{tree}(\top[\Delta]) &\iff e, d \models' \Delta \\
e, (\sigma, \text{ph}, (\emptyset, \emptyset)) \models \text{fd}(\text{fd}, \iota, E) &\iff \exists fd. \text{ph} = fd \mapsto (e(\iota), \langle E \rangle_e) \wedge \sigma = \text{fd} \mapsto fd \\
e, (\emptyset, \emptyset, (\emptyset, \text{if})) \models \text{fh}(\iota, E, E') &\iff \text{if} = e(\iota) \mapsto (\langle E \rangle_e \mapsto \langle E' \rangle_e) \\
e, (\sigma, \text{ph}, (\emptyset, \emptyset)) \models \text{ds}(\text{dir}, E) &\iff \exists ds. \text{ph} = ds \mapsto \langle E \rangle_e \wedge \sigma = \text{dir} \mapsto ds \\
e, (\sigma, \emptyset, (\emptyset, \emptyset)) \models \text{var}(\mathbf{x}, E) &\iff \sigma = \text{var} \mapsto \langle E \rangle_e \\
\\
e, (\sigma, \text{ph}, (\top[d], f)) \models P * Q &\iff \begin{aligned} &\exists \sigma', \sigma'', \text{ph}', \text{ph}'', d', d'', f', f''. (\sigma, \text{ph}, (\top[d], f)) = (\sigma' \uplus \sigma'', \text{ph}' \uplus \text{ph}'', (\top[d], f' \uplus f')) \\ &\wedge d \equiv d' \otimes d'' \wedge e, (\sigma', \text{ph}', (\top[d'], f')) \models P \wedge e, (\sigma'', \text{ph}'', (\top[d''], f'')) \models Q \end{aligned} \\
e, (\sigma, \text{ph}, (\top[d], f)) \models P \hat{*} Q &\iff \begin{aligned} &\exists \sigma', \sigma'', \text{ph}', \text{ph}'', d', d'', f', f''. (\sigma, \text{ph}, (\top[d], f)) = (\sigma' \uplus \sigma'', \text{ph}' \uplus \text{ph}'', (\top[d], f' \uplus f')) \\ &\wedge d \equiv_{\wedge} d' \otimes d'' \wedge e, (\sigma', \text{ph}', (\top[d'], f')) \models P \wedge e, (\sigma'', \text{ph}'', (\top[d''], f'')) \models Q \end{aligned} \\
e, (\emptyset, \emptyset, (\top[d], \emptyset)) \models E &\iff \top[d] = \langle E \rangle_e \\
e, (\emptyset, \emptyset, (\emptyset, \emptyset)) \models \text{emp} &\iff \text{true} \\
\\
e, d \models' E_{E'} : \iota &\iff d \equiv \langle E \rangle_{e \langle E' \rangle_e} : e(\iota) \\
e, d \models' \langle E \rangle_{E'} &\iff d \equiv \langle \langle E \rangle_e \rangle_{\langle E' \rangle_e} \\
e, d \models' E_{E'}[\Delta] &\iff \exists d'. d \equiv \langle E \rangle_{e \langle E' \rangle_e} [d'] \wedge e, d' \models' \Delta \\
e, d \models' \Delta * \Delta' &\iff \exists d', d''. d \equiv d' \otimes d'' \wedge e, d' \models' \Delta \wedge e, d'' \models' \Delta' \\
e, d \models' \Delta \hat{*} \Delta' &\iff \exists e, d', d''. d \equiv_{\wedge} d' \otimes d'' \wedge e, d' \models' \Delta \wedge e, d'' \models' \Delta' \\
e, d \models' E &\iff d \equiv \langle E \rangle_e \\
e, \emptyset \models' \emptyset &\iff \text{true}
\end{aligned}$$

Figure 4: Satisfaction relation.

$$\text{pvar}(\text{path}, p) \triangleq \exists cp. \text{var}(\text{path}, cp) \wedge \text{strippath}(p) = cp$$

$$\begin{aligned}
\text{all}(d) &\triangleq d \wedge \exists S, S'. (\text{FNAMES} = S \uplus S') \\
&\wedge \left(\bigotimes_{n \in S} \exists \pi. \text{ent}_{\pi}(n) \right) * \exists \pi. \langle S' \rangle_{\pi}
\end{aligned}$$

$$\text{names}(S) \triangleq \forall a. (a \in S \iff \text{true} * \exists \pi. \text{ent}_{\pi}(a))$$

$$E \doteq E' \triangleq E = E' \wedge \text{emp}$$

where $\text{strippath}(p)$ is the strippath function from definition 21 lifted to the assertion language. The assertion $\hat{a}_{\pi}$ is derived notation for a singleton set of a non-existing en-

try with name a and permission π . $\text{ent}_{\pi}(a)$ describes a file or directory with name a and permission π . $\bigotimes_{x \in S} \Delta$ is the iterative version of $*$ on elements of the set S . The assertion $\text{pvar}(\text{path}, p)$ states that the path value of the program variable path matches the instrumented pathname p once permissions are stripped. The predicate $\text{all}(d)$ describes a complete list of entries d , requiring ownership of all possible entries either existing (set S) or non-existing (set S'). $\text{names}(S)$ describes entries for every file name in the set S and finally, $E \doteq E'$ provides a $*$ -separated version of equals.

Consider the following two instrumented directories: $a_1[\emptyset]$ and $a_1[\widehat{b}_1]$. By reification (definition 20), 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 [10].

Definition 25 (Semantic Entailment). *The semantic entailment relation, $\preceq \subset \text{ASRTS} \times \text{ASRTS}$, is defined as:*

$$\begin{aligned} P \preceq Q &\iff \\ \forall R \in \text{ASRTS}, e \in \text{LENVS}. & \llbracket P * R \rrbracket_e \subseteq \llbracket Q * R \rrbracket_e \end{aligned}$$

where $\llbracket P \rrbracket_e \triangleq \{is \in \text{ISTATES} \mid e, is \models P\}$ denotes the set of all instrumented states satisfying P and $\llbracket - \rrbracket$ is the reification function from definition 20.

With semantic entailment we can generalise the previous example: $a_1[\Delta \wedge \text{notnamed}(S)] \preceq a_1[\Delta * \langle S \rangle_1]$ where $\text{notnamed}(S)$ asserts that the names in S are not mentioned as top level entries in the directory and is defined as:

$$\begin{aligned} \text{notnamed}(S) &\triangleq \\ \forall a. a \in S &\iff \\ \text{true} * \exists \pi. \text{ent}_\pi(a) & \\ \neg \left(\bigvee \left(\exists S'. \widehat{\langle S' \rangle}_\pi \wedge a \in S' \right) \right) & \end{aligned}$$

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

When $\pi < 1$, the assertion $a_\pi[\emptyset]$ 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 $a_\pi[\text{true}]$ 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 $a_{1/2}[\langle \text{FNAMES} \rangle_{1/2}]$.

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 [10]; 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 . We build a Hoare-style program logic comprising the standard Hoare inference rules for control flow commands, such as **if** and **while**, the semantic consequence rule of views [10], the new *effect frame* rule, standard axioms for **skip** and assignment from separation logic and the small axioms of figure 5. We adopt a fault avoiding

interpretation of Hoare triples $\{P\} \mathbb{C} \{Q\}$. Running $\mathbb{C}$ on states reified from P will not fault, and if it terminates the resulting states are those arising from the reification of Q .

The small axioms for our POSIX fragment are given in figure 5. 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 `pvar(path, p/a1)` states that `path` has a path-prefix `p` and ends with the name `a` with full permission 1. Assertion `tree(p/a1)` states that `a` does not exist at the end of `p` and a new entry can be created as the permission is 1. Assertion `var(x, -)` states that variable `x` has an undeclared value using the variables-as-resource interpretation [5]. In the postcondition, assertion `tree(p/a1[ $\emptyset$ ])` states that a new empty directory, named `a` with permission 1, has been created at the end of path `p`. Assertion `var(x, 0)` states that the return value captured by `x` is set to 0 indicating a successful update.

Now consider `rmdir(path)` which removes the empty directory identified by `path`. In the precondition, assertion `tree(p/a1[ $\emptyset$ ])` 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 `tree(p//a1)` states that the entry `a1` 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 5 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 `tree(p//a1)`, 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:

$$\text{EFFECTFRAME} \frac{\vdash \{P\} \mathbb{C} \{Q\}}{\vdash \{P * R\} \mathbb{C} \{Q \hat{*} R\}}$$

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

$$\begin{array}{ll}
\left\{ \begin{array}{l} \text{var}(\mathbf{r}, -) * \text{pvar}(\text{path}, p/a_\pi) * \text{lin}(p/a_\pi) \doteq lp \\ \quad * \text{tree}(p/(d \wedge \text{ent}_\pi(a))) \\ \mathbf{r} := \text{realpath}(\text{path}) \\ \text{pvar}(\mathbf{r}, lp) * \text{pvar}(\text{path}, p/a_\pi) * \text{lin}(p/a_\pi) \doteq lp \\ \quad * \text{tree}(p/d) \end{array} \right\} & \left\{ \begin{array}{l} \text{var}(\mathbf{r}, -) * \text{pvar}(\text{old}, p/a_1) * \text{pvar}(\text{new}, p'/b_1) \\ \quad * \text{tree}(p/a_1[d]) * \text{tree}(p'/\widehat{b}_1) \\ \mathbf{r} := \text{rename}(\text{old}, \text{new}) \\ \text{var}(\mathbf{r}, 0) * \text{pvar}(\text{old}, p/a_1) * \text{pvar}(\text{new}, p'/b_1) \\ \quad * \text{tree}(p/\widehat{a}_1) \hat{*} \text{tree}(p'/b_1[d]) \end{array} \right\} \\
\left\{ \begin{array}{l} \text{var}(\mathbf{r}, -) * \text{pvar}(\text{path}, p/a_1) * \text{tree}(p/\widehat{a}_1) \\ \mathbf{r} := \text{mkdir}(\text{path}) \\ \text{var}(\mathbf{r}, 0) * \text{pvar}(\text{path}, p/a_1) * \text{tree}(p/a_1[\emptyset]) \end{array} \right\} & \left\{ \begin{array}{l} \text{var}(\mathbf{r}, -) * \text{pvar}(\text{old}, p/a_\pi) * \text{pvar}(\text{new}, p'/a_\pi) \\ \quad * \text{lin}(p/a_1) \doteq \text{lin}(p'/a_1) * \text{tree}(p/(d \wedge \text{ent}_\pi(a))) \\ \mathbf{r} := \text{rename}(\text{old}, \text{new}) \\ \text{var}(\mathbf{r}, 0) * \text{pvar}(\text{old}, p/a_\pi) * \text{pvar}(\text{new}, p'/a_\pi) \\ \quad * \text{lin}(p/a_1) \doteq \text{lin}(p'/a_1) * \text{tree}(p/d) \end{array} \right\} \\
\left\{ \begin{array}{l} \text{var}(\mathbf{r}, -) * \text{pvar}(\text{path}, p/a_1) * \text{tree}(p/a_1[\emptyset]) \\ \mathbf{r} := \text{rmdir}(\text{path}) \\ \text{var}(\mathbf{r}, 0) * \text{pvar}(\text{path}, p/a_1) * \text{tree}(p/\widehat{a}_1) \end{array} \right\} & \left\{ \begin{array}{l} \text{var}(\mathbf{t}, -) * \text{pvar}(\text{path}, p) * \text{tree}(p/\emptyset) \\ \mathbf{t} := \text{stat}(\text{path}) \\ \text{var}(\mathbf{t}, D) * \text{pvar}(\text{path}, p) * \text{tree}(p/\emptyset) \end{array} \right\} \\
\left\{ \begin{array}{l} \text{var}(\mathbf{r}, -) * \text{pvar}(\text{path}, p/a_1) * \text{tree}(p/a_1:\iota) \\ \mathbf{r} := \text{unlink}(\text{path}) \\ \text{var}(\mathbf{r}, 0) * \text{pvar}(\text{path}, p/a_1) * \text{tree}(p/\widehat{a}_1) \end{array} \right\} & \left\{ \begin{array}{l} \text{var}(\mathbf{t}, -) * \text{pvar}(\text{path}, p/a_\pi) * \text{tree}(p/a_\pi:\iota) \\ \mathbf{t} := \text{stat}(\text{path}) \\ \text{var}(\mathbf{t}, F) * \text{pvar}(\text{path}, p/a_\pi) * \text{tree}(p/a_\pi:\iota) \end{array} \right\} \\
\left\{ \begin{array}{l} \text{var}(\mathbf{r}, -) * \text{pvar}(\text{old}, p/a_1) * \text{pvar}(\text{new}, p'/b_1) \\ \quad * \text{tree}(p/a_1:\iota) * \text{tree}(p'/b_1:\iota') \\ \mathbf{r} := \text{rename}(\text{old}, \text{new}) \\ \text{var}(\mathbf{r}, 0) * \text{pvar}(\text{old}, p/a_1) * \text{pvar}(\text{new}, p'/b_1) \\ \quad * \text{tree}(p/\widehat{a}_1) * \text{tree}(p'/b_1:\iota) \end{array} \right\} & \left\{ \begin{array}{l} \text{var}(\mathbf{dir}, -) * \text{pvar}(\text{var}, p) * \text{tree}(p/\text{all}(d)) \\ \mathbf{dir} := \text{opendir}(\text{path}) \\ \left\{ \begin{array}{l} \text{ds}(\mathbf{dir}, S) * \text{pvar}(\text{var}, p) \\ * \text{tree}(p/(d \wedge \text{names}(S))) \end{array} \right\} \end{array} \right\} \\
\left\{ \begin{array}{l} \text{var}(\mathbf{r}, -) * \text{pvar}(\text{old}, p/a_1) * \text{pvar}(\text{new}, p'/b_1) \\ \quad * \text{tree}(p/a_1:\iota) * \text{tree}(p'/\widehat{b}_1) \\ \mathbf{r} := \text{rename}(\text{old}, \text{new}) \\ \text{var}(\mathbf{r}, 0) * \text{pvar}(\text{old}, p/a_1) * \text{pvar}(\text{new}, p'/b_1) \\ \quad * \text{tree}(p/\widehat{a}_1) * \text{tree}(p'/b_1:\iota) \end{array} \right\} & \left\{ \begin{array}{l} \text{var}(\mathbf{fn}, -) * \text{ds}(\mathbf{dir}, S) \wedge S \neq \emptyset \\ \mathbf{fn} := \text{readdir}(\mathbf{dir}) \\ \text{var}(\mathbf{fn}, a) * \text{ds}(\mathbf{dir}, (S \setminus \{a\})) \wedge a \in S \end{array} \right\} \\
\left\{ \begin{array}{l} \text{var}(\mathbf{r}, -) * \text{pvar}(\text{old}, p/a_1) * \text{pvar}(\text{new}, p'/b_1) \\ \quad * \text{tree}(p/a_1[d]) * \text{tree}(p'/b_1[\emptyset]) \\ \mathbf{r} := \text{rename}(\text{old}, \text{new}) \\ \text{var}(\mathbf{r}, 0) * \text{pvar}(\text{old}, p/a_1) * \text{pvar}(\text{new}, p'/b_1) \\ \quad * \text{tree}(p/\widehat{a}_1) \hat{*} \text{tree}(p'/b_1[d]) \end{array} \right\} & \left\{ \begin{array}{l} \text{var}(\mathbf{fn}, -) * \text{ds}(\mathbf{dir}, \emptyset) \\ \mathbf{fn} := \text{readdir}(\mathbf{dir}) \\ \text{var}(\mathbf{fn}, 0) * \text{ds}(\mathbf{dir}, \emptyset) \end{array} \right\} \\
& \left\{ \begin{array}{l} \text{ds}(\mathbf{dir}, S) \\ \text{closedir}(\mathbf{dir}) \\ \text{var}(\mathbf{dir}, -) \end{array} \right\}
\end{array}$$

Figure 5: Small axioms for the POSIX file-system fragment.

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 [10] 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, $\mathbb{C}_{\text{fs}}$ (§3.4); the concrete states, STATES (definition 10); the separation algebra (ISTATES, $\bullet$, $(\emptyset, \emptyset, (\emptyset, \emptyset))$); the axioms of figure 5; and the reification function $\lfloor - \rfloor$ of definition 20. 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 $e \in \text{LENVS}$, $st \in \llbracket P \rrbracket_e$ and $st' \in \text{STATES}$, if $\mathbb{C}, st \rightsquigarrow^* \text{skip}$, st' in the operational semantics then $st' \in \llbracket Q \rrbracket_e$.*

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 6a. 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.

```

1: rmr(path)  $\triangleq$ 
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  $\neq$  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)  $\triangleq$ 
2:   local lp in
3:     lp := realpath(path)
4:     rmr(lp)
5:   end

```

(b) A safe recursive remove.

Figure 6: A naive recursive remove (top), that fails to remove a directory completely, and the suggested fix (bottom).

Formally, the specification is:

$$\{ \text{pvar}(\text{path}, p/a_1) * \text{tree}(p/\text{ent}_1(a)) \} \\ \text{rmr}(\text{path}) \\ \{ \text{pvar}(\text{path}, p/a_1) * \text{tree}(p/\widehat{a}_1) \}$$

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.

The implementation shown in figure 6a 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 6a will reach line 3 and invoke `stat` using a non-existing path. The axioms of figure 5 require paths to exist. In POSIX, commands fail with an error code when invalid paths are used. We specify such error cases in appendix C. 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:

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

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

$$\{ \text{var}(t, -) * \text{var}(\text{errno}, -) * \text{E_NOENT}(\text{path}, v) \wedge t \} \\ t := \text{stat}(\text{path}) \\ \{ \text{var}(t, -1) * \text{var}(\text{errno}, \text{E_NOENT}) * \text{var}(\text{path}, v) * t \}$$

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 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 6a is:

$$\{ \text{var}(\text{errno}, -) * \text{pvar}(\text{path}, p/a_1) * \text{tree}(p/\text{ent}_1(a)) \} \\ \text{rmr}(\text{path}) \\ \{ \text{var}(\text{errno}, e) * \text{pvar}(\text{path}, p/a_1) \\ * \text{tree}(p/\widehat{a}_1 \vee (e = \text{E_NOENT} \wedge a_1[\text{true}])) \}$$

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. In figure 7 we prove that the naive implementation satisfies the specification given above.

In figure 7, the implementation starts by checking the type of the entry on line 5, with `stat`. Here, and also throughout the proof, we are implicitly using the frame rule to frame off resource not affected. If the entry is a file it removed on line 9. This is a straightforward application of the command's small axiom and the update has no effects to consider. Note that this branch behaves sensibly.

The problem with the implementation arises in the case path identifies a directory. In the branch handling this case we first open the directory for reading (line 13) and start iterating over the contents (lines 15 - 24) recursively invoking the operation for each entry read. Intuitively, the problem lies in how the recursive call is made. Note that we append the entry read (`fn` variable) to path and using this as argument for the recursive call. Assuming the operation succeeds, the entry identified by path/fn is removed, which has an effect on path if they overlap. This means that at some point the pathname used in the recursive call will no longer be valid.

Consider the loop invariant on line 17. The main assertion is the one of the form $\text{tree}(p/\widehat{a}_1[\cdot \cdot])$, stating that the directory we want removed (a) either is empty, in which case the iteration will stop, or there is at least one entry. Note that this uses the $\widehat{\cdot}$ -pathname assertion to indicate that removing sub-directories of a has effects on p . Also, an error may have already been encountered. Effectively, the invariant is saying that we do not know whether pathname p is valid or not.

```

1: {var(errno, -) * pvar(path, p/a1) * tree(p/ent1(a))}
2: rmr(path)  $\triangleq$ 
3:   local t, dir, fn in
4:     {var(errno, -) * var(t, -) * var(dir, -) * var(fn, -) * pvar(path, p/a1) * tree(p/ent1(a))}
5:     t := stat(path)
6:     {var(errno, -) * var(t, t) * var(dir, -) * var(fn, -) * pvar(path, p/a1) * tree(p/(a1[X]  $\wedge$  t = D  $\vee$  a1: $\iota$   $\wedge$  t = F))}
7:     if t = F then
8:       {var(errno, -) * var(t, F) * var(dir, -) * var(fn, -) * pvar(path, p/a1) * tree(p/a1: $\iota$ )}
9:       unlink(path)
10:      {var(errno, -) * var(t, F) * var(dir, -) * var(fn, -) * pvar(path, p/a1) * tree(p/ $\widehat{a}_1$ )}
11:     else if t = D then
12:       {var(errno, -) * var(t, D) * var(dir, -) * var(fn, -) * pvar(path, p/a1) * tree(p/a1[X])}
13:       dir := opendir(path)
14:       { $\exists S$ . var(errno, -) * var(t, D) * ds(dir, S) * var(fn, -) * pvar(path, p/a1) * tree(p/a1[X  $\wedge$  names(S)])}
15:       fn := readdir(dir)
16:       { $\exists S, b$ . var(errno, -) * var(t, D) * ds(dir, S) * var(fn, b) * pvar(path, p/a1) * tree(p/a1[X  $\wedge$  names(S)  $\wedge$  (( $\emptyset \wedge b = 0 \wedge S = \emptyset$ )  $\vee b \in S$ ))]}
17:       { $\left. \begin{array}{l} \exists S, b, X', err. \\ \text{var(errno, err) * var(t, D) * ds(dir, S) * var(fn, b) * pvar(path, p/a}_1\text{)} \\ * \text{tree}\left(\frac{p}{a_1} \left[ \begin{array}{l} (b = 0 \wedge S = \emptyset \wedge (\emptyset \vee (err = E\_NOENT \wedge \exists b'. \text{ent}_1(b) * \text{true}))) \\ \vee (\text{ent}_1(b) * (X' \wedge \text{names}(S \setminus \{b\}))) \end{array} \right] \right) \end{array} \right\}$ }
18:       while fn  $\neq$  0 do
19:         { $\exists S, b, X'. \text{var(errno, -) * ds(dir, S) * var(fn, b) * pvar(path, p/a}_1\text{)} * \text{tree}\left(\frac{p}{a_1} \left[ \text{ent}_1(b) * (X' \wedge \text{names}(S \setminus \{b\})) \right] \right)$ }
20:         { $\exists S, b, X'. \text{var(errno, -) * ds(dir, S) * var(fn, b) * pvar(path, p/a}_1\text{)} * \text{tree}\left(\frac{p}{a_1} \left[ \text{ent}_1(b) * (X' \wedge \text{names}(S \setminus \{b\})) \right] \right) \vee \text{tree}\left(\frac{ram(p)}{a_1} \left[ \text{ent}_1(b) * (X' \wedge \text{names}(S \setminus \{b\})) \right] \right)$ }
21:         rmdir(path/fn)
22:         { $\exists S, b, X'. \text{var(errno, -) * ds(dir, S) * var(fn, b) * pvar(path, p/a}_1\text{)} * \text{tree}\left(\frac{p}{a_1} \left[ \begin{array}{l} \widehat{b}_1 * (X' \wedge \text{names}(S \setminus \{b\})) \\ \vee (\text{ent}_1(b) * X' \wedge \text{names}(S \setminus \{b\})) \end{array} \right] \right)$ }
23:         fn := readdir(dir)
24:       done
25:       {var(errno, err) * var(t, D) * ds(dir,  $\emptyset$ ) * var(fn, 0) * pvar(path, p/a1) * tree(p/a1[ $\emptyset \vee (\exists b. \text{ent}_1(b) * \text{true} \wedge err = E\_NOENT)$ ])}
26:       closedir(dir)
27:       {var(errno, -) * var(t, D) * var(dir, -) * var(fn, 0) * pvar(path, p/a1) * tree(p/a1[ $\emptyset$ ]  $\vee$  tree(ram(p)/a1[true]))}
28:       rmdir(path)
29:       {var(errno, err) * var(t, D) * var(dir, -) * var(fn, 0) * pvar(path, p/a1) * tree(p/( $\widehat{a}_1 \vee (err = E\_NOENT \wedge a_1[\text{true}])$ ))}
30:     fi
31:     {var(errno, e) * var(t, -) * var(dir, -) * var(fn, -) * pvar(path, p/a1) * tree(p/( $\widehat{a}_1 \vee (e = E\_NOENT \wedge a_1[\text{true}])$ ))}
32:   end
33: {var(errno, e) * pvar(path, p/a1) * tree(p/( $\widehat{a}_1 \vee (e = E\_NOENT \wedge a_1[\text{true}])$ ))}

```

Figure 7: Proof derivation of a naive implementation of recursive delete.

Therefore, before making the recursive call we consider two cases: one when the p does not need effects to be propagated (after propagating effects p is the same), and one where it does. To express the second case we use the following abstraction:

$$\text{tree}(\text{ram}(p)/\Delta) \triangleq \exists p'. p \not\equiv p' \wedge \text{tree}(p/\Delta) \wedge \text{tree}(p'/\Delta)$$

In both cases, we get to the postcondition on line 22.

When we exit the loop, we close the directory. If no error has been generated in the loop then we definitely know that the directory is empty as is specified on line 27. However, the pathname may still require effect updates, which means that the next `rmdir` may produce an `E_NOENT` error.

Finally, we exit the directory handling branch and combine the postconditions of the two branches to produce the final postcondition.

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

$$\begin{aligned} & \{ \text{pvar}(\text{path}, p/a_1) * \text{islin}(p/a_1) * \text{tree}(p/\text{ent}_1(a)) \} \\ & \quad \text{rmr}(\text{path}) \\ & \{ \text{pvar}(\text{path}, p/a_1) * \text{islin}(p) * \text{tree}(p/\widehat{a}_1) \} \end{aligned}$$

where $\text{islin}(p) \triangleq \text{lin}(p) \doteq p$ 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 8.

```

1: {pvar(path, p/a1) * islin(p/a1) * tree(p/ent1(a))}
2: rmr(path)  $\triangleq$ 
3:   local t, dir, fn in
4:     {var(t, -) * var(dir, -) * var(fn, -) * pvar(path, p/a1) *
      * islin(p/a1) * tree(p/ent1(a))}
5:     t := stat(path)
6:     { $\exists d, t. \text{var}(t, t) * \text{var}(\text{dir}, -) * \text{var}(\text{fn}, -) * \text{pvar}(\text{path}, p/a_1) *
      * \text{islin}(p/a_1) * \text{tree}(p / ((a_1 : t \wedge t = F) \vee (a_1[d] \wedge t = D)))$ }
7:     if t = F then
8:       {var(t, F) * pvar(path, p/a1) * islin(p/a1) * tree(p/a1:t)}
9:       unlink(path)
10:      {var(t, F) * pvar(path, p/a1) * islin(p) * tree(p/a1) }
11:     else if t = D then
12:       {var(t, D) * var(dir, -) * var(fn, -) * pvar(path, p/a1) *
        * islin(p/a1) * tree(p/a1[d]) }
13:       dir := opendir(path); fn := readdir(dir)
14:       { $\exists S, b, d'. \text{ds}(\text{dir}, S) * \text{var}(\text{fn}, b) *
        \text{pvar}(\text{path}, p/a_1) * \text{islin}(p/a_1) *
        * \text{tree}(p/a_1 [(\vee (\text{ent}_1(b) * d' \wedge \text{names}(S \setminus \{b\})))])$ }
15:       while fn  $\neq$  0 do
16:         rmr(path/fn); fn := readdir(dir)
17:       done
18:       {ds(dir, 0) * var(fn, 0) * pvar(path, p/a1) * islin(p/a1) *
        * tree(p/a1[0]) }
19:       closedir(dir); rmdir(path)
20:     fi
21:   end
22: {pvar(path, p/a1) * islin(p) * tree(p/a1) }

```

Figure 8: Proof derivation on a naive recursive remove implementation when we specifically use a linear pathname.

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 6b we demonstrate such an implementation that meets the intended specification. Note that we reuse the naive `rmr` of figure 6a.

The proof is given in figure 9. 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 $\text{tree}(p_l/\text{ent}_1(a))$ 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 $\text{tree}(p'/\emptyset)$. 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 $\text{tree}(p'/\emptyset)$. Note the use of $\hat{*}$ on the combined state. We propagate the effects on line 13, where the assertion $\text{tree}(p/\hat{a}_1)$ 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 be-

```

1: {pvar(path, p/a1) * tree(p/ent1(a))}
2: rmrSafe(path)  $\triangleq$ 
3:   local lp in
4:     {var(lp, -) * pvar(path, p/a1) * tree(p/ent1(a))}
5:     { $\exists p_l. \text{var}(lp, -) * \text{pvar}(\text{path}, p/a_1) *
      * \text{tree}(p/\text{ent}_1(a)) * \text{lin}(p) \doteq p_l$ }
6:     lp := realpath(path)
7:     { $\exists p_l. \text{pvar}(lp, p_l) * \text{pvar}(\text{path}, p/a_1) *
      * \text{tree}(p/\text{ent}_1(a)) * \text{lin}(p) \doteq p_l$ }
8:     { $\exists p_l, p'. \text{pvar}(lp, p_l) * \text{pvar}(\text{path}, p/a_1) * \text{lin}(p) \doteq p_l *
      * \text{tree}(p/\emptyset) \iff \text{tree}(p_l/\emptyset) * \text{tree}(p'/\emptyset) *
      * \text{tree}(p_l/\text{ent}_1(a)) * \text{tree}(p'/\emptyset)$ }
9:     {pvar(lp, p_l) * islin(p_l) * tree(p_l/ent1(a))}
10:    rmr(lp)
11:    {pvar(lp, p_l) * islin(p_l) * tree(p_l/a1)}
12:    { $\exists p_l, p'. \text{pvar}(lp, p_l) * \text{pvar}(\text{path}, p/a_1) * \text{lin}(p) \doteq p_l *
      * \text{tree}(p/\emptyset) \iff \text{tree}(p_l/\emptyset) * \text{tree}(p'/\emptyset) *
      * \text{tree}(p_l/\hat{a}_1) * \text{tree}(p'/\emptyset)$ }
13:    { $\exists p_l. \text{pvar}(lp, p_l) * \text{pvar}(\text{path}, p/a_1) * \text{lin}(p) \doteq p_l *
      * \text{tree}(p/\hat{a}_1)$ }
14:  end
15: {pvar(path, p/a1) * tree(p/\hat{a}_1)}

```

Figure 9: Proof derivation of the fixed recursive delete.

ing 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 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 the readers to appendix E for the 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 11) with symbolic links: $d ::= \dots \mid a_\pi @ \text{cap}$, where $a_\pi @ \text{cap}$ is a symbolic link file named a with path value cap . Instrumented directory contexts (definition 22) and concrete directories (definition 1) are extended similarly. Next, we extend the fusion equivalences of definition 12 with:

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

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: $rp ::= \dots \mid a_\pi @ ap$. This provides information about the footprint of paths with symbolic links. For example, given the instrumented pathname $/usr_1/4/(lib_1@/usr_1/4/local_1/2/lib_1/2)/tex_1$ 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:

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

Note that symbolic link loops may occur in the directory structure, as in $\top[usr[la@/usr/lb \otimes lb@/usr/la]]$. 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.

```
{var(r, -) * var(target, cp) * pvar(link, p/a1) * tree(p/a1)}
  r := symlink(target, link)
  {var(r, -) * var(target, cp) * pvar(link, p/a1)}
    * tree(p/a1@cp)
```

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.

References

- [1] POSIX.1-2008, IEEE 1003.1-2008, The Open Group Base Specifications Issue 7.
- [2] Konstantine Arkoudas, Karen Zee, Viktor Kuncak, and Martin Rinard. Verifying a File System Implementation. In *LNCIS: Formal Methods and Software Engineering*. Springer Berlin Heidelberg, 2004.
- [3] Nicolas Biri and Didier Galmiche. Models and separation logics for resource trees. *Journal of Logic and Computation*, 17(4):687–726, 2007.
- [4] Richard Bornat, Cristiano Calcagno, Peter O’Hearn, and Matthew Parkinson. Permission Accounting in Separation Logic. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, 2005.
- [5] Richard Bornat, Cristiano Calcagno, and Hongseok Yang. Variables as Resource in Separation Logic. *Electronic Notes in Theoretical Computer Science*, 155:247–276, 2006.
- [6] John Boyland. Checking Interference with Fractional Permissions. In *Static Analysis*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003.
- [7] Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. Context Logic and Tree Update. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, 2005.
- [8] Luca Cardelli and Andrew D Gordon. Ambient logic. *Mathematical Structures in Computer Science*, 2003.
- [9] Pedro da Rocha Pinto, Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, and Mark 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] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. Views: Compositional Reasoning for Concurrent Programs. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, 2013.
- [11] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, MatthewJ. Parkinson, and Viktor Vafeiadis. Concurrent Abstract Predicates. In *ECOOP*. Springer Berlin Heidelberg, 2010.
- [12] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor 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] Gidon Ernst, Gerhard Schellhorn, Dominik Haneberg, Jörg Pfähler, and Wolfgang Reif. Verification of a Virtual Filesystem Switch. In *Verified Software: Theories, Tools, Experi-*

ments, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014.

- [14] Kathleen Fisher, Nate Foster, David Walker, and Kenny 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] Leo Freitas, Zheng Fu, and Jim Woodcock. POSIX File Store in Z/Eves: An Experiment in the Verified Software Repository. *Engineering of Complex Computer Systems, IEEE International Conference*, 2007.
- [16] Leo Freitas, Jim Woodcock, and Andrew Butterfield. POSIX and the Verification Grand Challenge: A Roadmap. *2014 19th International Conference on Engineering of Complex Computer Systems*, 0:153–162, 2008.
- [17] Philippa Gardner, Sergio Maffeis, and Gareth David 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] Philippa Gardner, Gian Ntzik, and Adam Wright. Local Reasoning for the POSIX File System. In *Programming Languages and Systems, Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2014.
- [19] Wim H. Hesselink and M.I. Lali. Formalizing a Hierarchical File System. *REFINE*, 2009.
- [20] Aquinas Hobor and Jules 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] Rajeev Joshi and Gerard J. Holzmann. A Mini Challenge: Build a Verifiable Filesystem. *Formal Aspects of Computing*, 19(2):269–272, 2007.
- [22] Carroll Morgan and Bernard Sufrin. Specification of the UNIX Filing System. *Software Engineering, IEEE Transactions on*, 1984.
- [23] Gian Ntzik and Philippa Gardner. Reasoning about the POSIX File System: Local Update and Global Pathnames. Technical report, Imperial College London, 2015.
- [24] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, 2002.
- [25] Adam Wright. *Structural Separation Logic*. PhD thesis, Imperial College London, 2013.

A. Software 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 12. 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:

$$P_i \triangleq \\ W \wedge \text{var}(\mathbf{r}, -) * \text{var}(\text{errno}, -) * \text{pvar}(\text{il}, \text{il}) \\ * \text{srcPre} * \text{binPre} * \text{optPre}$$

captures the state of the file system in the variable `W` describes the initial states of variables, the installation source directory `srcPre`, the `/usr/bin` directory `binPre` and the program data directory `optPre` with their definitions given in figure 10. 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 `/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

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

Figure 10: Components of the installer's precondition.

unmodified. This is captured by the postcondition:

$$\begin{aligned}
Q_i &\triangleq \\
&\exists r, \iota', j'. \text{var}(r, r) * \text{var}(\text{errno}, -) \\
&\quad \wedge (r = -1 \implies W) \\
&\quad \wedge \left(r = 0 \implies \begin{array}{c} src_{Post} * bin_{Post}(\iota') * opt_{Post}(j') \\ * files_{Post}(\iota', j') \end{array} \right)
\end{aligned}$$

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 11. 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:

$$\{P_i\} \text{installWidget} \{Q_i\}$$

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 5, extended with the following axiom for handling a symbolic link:

$$\begin{aligned}
&\{\text{var}(t, -) * \text{pvar}(\text{path}, p/a_\pi) * \text{tree}(p/a_\pi @ cp) \} \\
&\quad t := \text{lstat}(\text{path}) \\
&\{\text{var}(t, S) * \text{pvar}(\text{path}, p/a_\pi) * \text{tree}(p/a_\pi @ cp) \}
\end{aligned}$$

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:

$$\begin{aligned}
&\{\text{var}(t, -) * \text{pvar}(\text{path}, p) * \text{tree}(p/sa_\pi : \iota) \} \\
&\quad t := \text{stat}(\text{path}) \\
&\{\text{var}(t, F) * \text{pvar}(\text{path}, p) * \text{tree}(p/sa_\pi : \iota) \} \\
&\{\text{var}(t, -) * \text{pvar}(\text{path}, p) * \text{tree}(p/sa_\pi [\emptyset]) \} \\
&\quad t := \text{stat}(\text{path}) \\
&\{\text{var}(t, D) * \text{pvar}(\text{path}, p) * \text{tree}(p/sa_\pi [\emptyset]) \}
\end{aligned}$$

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

$$\begin{aligned}
\text{tree}(p/a_\pi / s \Delta) &\triangleq \\
&\text{tree}(p/a_\pi / \Delta) \\
&\vee \exists p'. \text{tree}(p/a_\pi @ \text{strippath}(p')) * \text{tree}(p' / s \Delta)
\end{aligned}$$

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

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

Our example installer, along with a proof sketch that it meets its specification: $\{P_i\} \text{installWidget} \{Q_i\}$, is given in figure 12.

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.

$$\begin{aligned}
srcPost &\triangleq srcP \\
binPost(\iota) &\triangleq tree\left(\text{'usr'}_{\pi_1} / \text{'bin'}_{\pi_2} / \left(\text{'widget'}_1 @ (\text{'usr'} / \text{'bin'} / \text{'widget-2.0'}) \right) \right. \\
&\quad \left. * \text{'widget-2.0'}_1 : \iota \right) \\
optPost(j) &\triangleq tree\left(\text{'opt'}_{\pi_3} / \left(\text{'widget'}_1 @ (\text{'opt'} / \text{'widget-2.0'}) \right) \right. \\
&\quad \left. * \text{'widget-2.0'}_1 [\text{'data'}_1 : j] \right) \\
fileSPost(\iota, j) &\triangleq file(\iota, \beta) * file(j, \beta')
\end{aligned}$$

Figure 11: Components of installer's postcondition.

```

1: {Pi}
2: r := installWidget  $\triangleq$ 
3:   local t1, t2 in
4:     {var(t1, -) * var(t2, -) * var(errno, -) * binPre * optPre}
5:     t1 := lstat('/usr/bin/widget');
6:     t2 := lstat('/opt/widget');
7:     {var(t1, t1)  $\wedge$  t1 = D  $\vee$  F  $\vee$  S  $\vee$  -1 * var(t2, t2)  $\wedge$  t2 = D  $\vee$  F  $\vee$  S  $\vee$  -1 }
8:     {var(errno, e)  $\wedge$  (t1 = -1  $\vee$  t2 = -1  $\implies$  e = E.NOENT) * binPre * optPre}
9:   if t1  $\neq$  S  $\vee$  t2  $\neq$  S then
10:    r := -1
11:   else
12:     {
13:       var(t1, S) * var(t2, S) * var(errno, -)
14:       * tree('/usr'_{\pi_1} / 'bin'_{\pi_2} / ((\exists cp. 'widget'_1 @ cp) * ent_1('widget-2.0')  $\vee$  'widget-2.0'_1))
15:       * tree('/opt'_{\pi_3} / ((\exists cp. 'widget'_1 @ cp) * ent_1('widget-2.0')  $\vee$  'widget-2.0'_1))
16:     }
17:     t1 := lstat('/opt/widget-2.0');
18:     t2 := lstat('/usr/bin/widget-2.0');
19:     {
20:       var(t1, t1)  $\wedge$  t1 = D  $\vee$  F  $\vee$  S  $\vee$  -1 * var(t2, t2)  $\wedge$  t2 = D  $\vee$  F  $\vee$  S  $\vee$  -1
21:       * var(errno, e)  $\wedge$  (t1 = -1  $\vee$  t2 = -1  $\implies$  e = E.NOENT)
22:     }
23:     {
24:       * tree('/usr'_{\pi_1} / 'bin'_{\pi_2} / ((\exists cp. 'widget'_1 @ cp) * ent_1('widget-2.0')  $\vee$  'widget-2.0'_1))
25:       * tree('/opt'_{\pi_3} / ((\exists cp. 'widget'_1 @ cp) * ent_1('widget-2.0')  $\vee$  'widget-2.0'_1))
26:     }
27:   if t1  $\neq$  -1  $\vee$  t2  $\neq$  -1 then
28:    r := -1
29:   else
30:     {
31:       var(t1, -1) * var(t2, -1) * var(errno, -)
32:       * tree('/usr'_{\pi_1} / 'bin'_{\pi_2} / ((\exists cp. 'widget'_1 @ cp) * 'widget-2.0'_1))
33:       * tree('/opt'_{\pi_3} / ((\exists cp. 'widget'_1 @ cp) * 'widget-2.0'_1))
34:     }
35:     r := mkdir('/opt/widget-2.0')
36:     {
37:       var(t1, -1) * var(t2, -1) * var(errno, -)
38:       * tree('/usr'_{\pi_1} / 'bin'_{\pi_2} / ((\exists cp. 'widget'_1 @ cp) * 'widget-2.0'_1))
39:       * tree('/opt'_{\pi_3} / ((\exists cp. 'widget'_1 @ cp) * 'widget-2.0'_1[\emptyset]))
40:     }
41:     r := fileCopy(il/'widget', '/bin/widget-2.0');
42:     r := fileCopy(il/'data', '/opt/widget-2.0/data');
43:     {
44:       \exists \iota', j'. var(t1, -1) * var(t2, -1) * var(errno, -)
45:       * tree('/usr'_{\pi_1} / 'bin'_{\pi_2} / ((\exists cp. 'widget'_1 @ cp) * 'widget-2.0'_1 : \iota'))
46:       * tree('/opt'_{\pi_3} / ((\exists cp. 'widget'_1 @ cp) * 'widget-2.0'_1 ['data'_1 : j]))
47:       * fileSPost(\iota', j')
48:     }
49:     r := unlink('/usr/bin/widget');
50:     r := unlink('/opt/widget');
51:     {
52:       \exists \iota', j'. var(t1, -1) * var(t2, -1) * var(errno, -)
53:       * tree('/usr'_{\pi_1} / 'bin'_{\pi_2} / ('widget'_1 * 'widget-2.0'_1 : \iota'))
54:       * tree('/opt'_{\pi_3} / ('widget'_1 * 'widget-2.0'_1 ['data'_1 : j]))
55:       * fileSPost(\iota', j')
56:     }
57:     r := symlink('/usr/bin/widget-2.0', '/usr/bin/widget');
58:     r := symlink('/opt/widget-2.0', '/opt/widget')
59:     { \exists \iota', j'. var(t1, -1) * var(t2, -1) * var(errno, -) * binPost * optPost * fileSPost(\iota', j') }
60:   fi
61: end
62: {Qi}

```

Figure 12: Implementation and proof sketch of an installer for “widget” 2.0.

B. Soundness

Views [10] 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.

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

Definition 26 (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 10.
3. **Interpretation of Atomic Commands:** A function $[-](-) : \mathbb{C}_{fs} \rightarrow \text{STATES} \rightarrow \mathcal{P}(\text{STATES})$ that associates each atomic command with a state transformer. Given in definition 30.
4. **Separation Algebra:** The partial commutative monoid $(\text{ISTATES}, \bullet, (\emptyset, \emptyset, (\emptyset, \emptyset)))$, defined using disjoint function union for variables stores, process heaps and file heaps, and $\otimes$ for directory trees. Induces a separation view monoid $(\mathcal{P}(\text{ISTATES}), *_v, U)$, where $U = \{(\emptyset, \emptyset, (\emptyset, \emptyset))\}$ and $p_1 *_v p_2 \triangleq \{m_1 \bullet m_2 \mid m_1 \in p_1, m_2 \in p_2\}$.
5. **Axiomatisation:** The axioms of figure 5, which we denote with the set *Axioms*.
6. **Reification:** The reification function in definition 31.
7. **Disjunction:** A total function, $\bigvee : \mathcal{P}(\mathcal{P}(\text{ISTATES})) \rightarrow \mathcal{P}(\text{ISTATES})$, which we define as set union.

Views provides a general operational semantics in the form of an operational transition relation. We use this relation as the basis of the operational semantics for our programming language.

Definition 27 (Operational Semantics). *The multi-step operational transition relation, $-, - \rightsquigarrow^* -, - : (\mathbb{C} \times \text{STATES}) \times (\mathbb{C} \times \text{STATES} \cup \{\zeta\})$, relates programs and their initial states to either a terminal state or a distinguished fault state ζ , where $\zeta \notin \text{STATES}$. Command/state pairs not in the relation are deemed to be divergent.*

In views the operational transition relation is parameterised the interpretation function (parameter 3) which provides semantics for the primitive commands (parameter 1). In order to define the interpretation function, we first define a concrete semantics for pathname resolution and directory tree update.

In the following, assume $\vec{S}$ to be the set of standard cons-lists of elements belonging to the set S . Non-linear pathnames do not follow nicely the inductive directory tree structure. Therefore, pathname resolution must be able to go back up the structure, with $'..'$, or make an arbitrary jump, with symbolic links. As such, we define the pathname

resolution function in terms of auxiliary state, which tracks all the previously visited directories.

Definition 28 (Pathname Resolution). *The function:*

$\text{resolve} : \text{DIRS} \cup \text{DTREES} \rightarrow \text{PATHS} \rightarrow \overrightarrow{\text{DIRS}} \rightarrow \text{DIRS} \cup \text{DTREES}$

traverses a given directory hierarchy following a pathname and returns the identified entry. It is defined in figure 13.

$$\begin{aligned}
 \text{resolve}(cd, '.. ', l) &\triangleq cd \\
 \text{resolve}(cd, '.. ', cd' :: t) &\triangleq cd' \\
 \text{resolve}(cd \otimes a[cd'], a, l) &\triangleq a[cd'] \\
 \text{resolve}(cd \otimes a : \iota, a, l) &\triangleq a : \iota \\
 \text{resolve}(cd \otimes a@cp, a, l) &\triangleq \text{resolve}(\top[\text{last}(l)], cp, []) \\
 \text{resolve}(cd, '.. '/cp, l) &\triangleq \text{resolve}(cd, cp, l) \\
 \text{resolve}(cd, '.. '/cp, cd' :: t) &\triangleq \text{resolve}(cd', cp, cd' :: t) \\
 \text{resolve}(cd, '.. '/cp, cd' :: []) &\triangleq \text{resolve}(cd', cp, cd' :: []) \\
 \text{resolve}(cd \otimes a[cd'], a/cp, l) &\triangleq \text{resolve}(cd', cp, (cd \otimes a[cd']) :: l) \\
 \text{resolve}(cd \otimes a@cp', a/cp, l) &\triangleq \text{resolve}(\top[\text{last}(l)], cp'/cp, []) \\
 \text{resolve}(\top[cd], /'.. ', []) &\triangleq \top[cd] \\
 \text{resolve}(\top[cd], /'.. ', []) &\triangleq \top[cd] \\
 \text{resolve}(\top[cd], /a, []) &\triangleq \text{resolve}(cd, a, cd :: []) \\
 \text{resolve}(\top[cd], /'.. '/cp, []) &\triangleq \text{resolve}(\top[cd], /cp, []) \\
 \text{resolve}(\top[cd], /'.. '/cp, []) &\triangleq \text{resolve}(\top[cd], /cp, []) \\
 \text{resolve}(\top[cd], /a/cp, []) &\triangleq \text{resolve}(cd, a/cp, cd :: [])
 \end{aligned}$$

where we use: the traditional cons-list syntax $[]$ to mean an empty list; $h :: t$ to mean a list with head h and tail t ; and $\text{last}(l)$ to be the last element in the list l . In all other cases the result is **undefined**.

Figure 13: Definition of resolve.

We define directory tree update in terms of two functions: update and remove. We use the first to replace an entry identified by a path with a new one. We use the second to remove an entry identified by a path.

Definition 29 (Directory Entry Update and Restriction). *The partial function, update, traverses a given directory hierarchy following a pathname and updates the identified entry or inserts a new one.*

$\text{update} : \text{DIRS} \cup \text{DTREES} \rightarrow \text{PATHS} \rightarrow \text{DIRS} \rightarrow \text{DIRS} \cup \text{DTREES} \cup \text{PATHS}$

It is defined in figure 14.

The partial function, remove, traverses a given directory hierarchy following a pathname and removes the identified entry.

$\text{remove} : \text{DIRS} \cup \text{DTREES} \rightarrow \text{PATHS} \rightarrow \text{DIRS} \cup \text{DTREES} \cup \text{PATHS}$

It is defined in figure 15.

As in the rest of this paper, for simplicity, the previous definitions require that symbolic link values are absolute pathnames. Modifying the definitions for relative path symbolic link values is trivial.

$$\begin{aligned}
& \text{update}(cd, \cdot', cd') \triangleq \cdot' \\
& \text{update}(cd, \cdot'', cd') \triangleq \cdot'' \\
& \text{update}(cd \otimes a[cd'], a, cd'') \triangleq cd \otimes cd'' \\
& \text{update}(cd \otimes a : \iota, a, cd') \triangleq cd \otimes cd' \\
& \text{update}(cd \otimes a @ cp, a, cd') \triangleq cd \otimes cd' \\
& \text{update}(cd, a, cd') \triangleq cd \otimes cd' \quad \text{if } \text{notentry}(a, cd) \\
& \text{update}(cd, \cdot' / cp, cd') \triangleq \text{update}(cd, cp, cd') \\
& \text{update}(cd, \cdot'' / cp, cd'') \triangleq cp \\
& \text{update}(cd \otimes a[cd'], a / cp, cd'') \triangleq \begin{cases} cd \otimes a[\text{update}(cd', cp, cd'')] & \text{if } \text{update}(cd', cp, cd'') \in \text{DIRS} \\ \text{update}(cd \otimes a[cd'], cp', cd'') & \text{if } \text{update}(cd', cp, cd'') = cp' \wedge cp' \in \text{RELPATHS} \\ cp' & \text{if } \text{update}(cd', cp, cd'') = cp' \wedge cp' \in \text{ABSPATHS} \end{cases} \\
& \text{update}(cd \otimes a @ cp', a / cp, cd'') \triangleq cp' / cp \\
& \text{update}(\top[cd], /cp, cd') \triangleq \begin{cases} \top[\text{update}(cd, cp, cd')] & \text{if } \text{update}(cd, cp, cd') \in \text{DIRS} \\ \top[\text{update}(cd, cp', cd')] & \text{if } \text{update}(cd, cp, cd') = cp' \wedge cp' \in \text{RELPATHS} \\ \text{update}(cd, cp', cd') & \text{if } \text{update}(cd, cp, cd') = cp' \wedge cp' \in \text{ABSPATHS} \end{cases}
\end{aligned}$$

Where $\text{notentry}(a, cd) \triangleq \neg \exists cd'', \iota. cd \equiv a[cd''] \otimes cd'' \vee cd \equiv a : \iota \otimes cd'$ and in all other cases the result is **undefined**.

Figure 14: Definition of update.

$$\begin{aligned}
& \text{remove}(cd, \cdot') \triangleq \cdot' \\
& \text{remove}(cd, \cdot'') \triangleq \cdot'' \\
& \text{remove}(cd \otimes a[cd'], a) \triangleq cd \\
& \text{remove}(cd \otimes a : \iota, a) \triangleq cd \\
& \text{remove}(cd \otimes a @ cp, a) \triangleq cd \\
& \text{remove}(cd, \cdot' / cp) \triangleq \text{remove}(cd, cp) \\
& \text{remove}(cd, \cdot'' / cp) \triangleq cp \\
& \text{remove}(cd \otimes a[cd'], a / cp) \triangleq \begin{cases} cd \otimes a[\text{remove}(cd', cp)] & \text{if } \text{remove}(cd', cp) \in \text{DIRS} \\ \text{remove}(cd \otimes a[cd'], cp') & \text{if } \text{remove}(cd', cp) = cp' \wedge cp' \in \text{RELPATHS} \\ cp' & \text{if } \text{remove}(cd', cp) = cp' \wedge cp' \in \text{ABSPATHS} \end{cases} \\
& \text{remove}(cd \otimes a @ cp', a / cp) \triangleq cp' / cp \\
& \text{remove}(\top[cd], /cp) \triangleq \begin{cases} \top[\text{remove}(cd, cp)] & \text{if } \text{remove}(cd, cp) \in \text{DIRS} \\ \top[\text{remove}(cd, cp')] & \text{if } \text{remove}(cd, cp) = cp' \wedge cp' \in \text{RELPATHS} \\ \text{remove}(\top[cd], cp') & \text{if } \text{remove}(cd, cp) = cp' \wedge cp' \in \text{ABSPATHS} \end{cases}
\end{aligned}$$

In all other cases the result is **undefined**.

Figure 15: Definition of remove.

Definition 30 (Interpretation of Atomic Commands). *The interpretation of the commands of our primitive commands is given with the following function:*

$$[-](-) : \mathbb{C}_{fs} \rightarrow \text{STATES} \rightarrow \mathcal{P}(\text{STATES}) \cup \{\dagger\}$$

We amend the reification function of definition 20 to take update effects into account. Effect reification is parameterised by the instrumented pre-states, which are used to detect if effect propagation is required. If it is not required, then we reify directly. Otherwise, we force update effects to be propagated via the effect fusion equivalence, $\equiv_{\wedge}$ (definition 12).

Definition 31 (Effect Reification). *The reification function, $[-]_- : \text{ISTATES} \rightarrow \text{ISTATES} \rightarrow \mathcal{P}(\text{STATES})$, is defined as:*

$$\begin{aligned}
& [(\sigma, \text{ph}, (\top[d], f))]_{is} \triangleq \\
& \begin{cases} [(\sigma, \text{ph}, (\top[d], f))] & \text{if } is = (\sigma, \text{ph}, (\top[d], f)) \\ [\{ (\sigma, \text{ph}, (\top[d'], f)) \mid d \equiv_{\wedge} d' \}] & \text{otherwise} \end{cases}
\end{aligned}$$

where $[-]$ is the reification from definition 31.

The properties required by the views framework are all expressed in terms of sets of views, which from definition 26 are sets of instrumented program states, and the separation view monoid composition $*_{\text{v}}$, which according to defini-

$$\begin{aligned}
& [\mathbf{r} := \mathbf{mkdir}(\mathbf{path})]((\sigma, \text{ph}, (t, f))) \\
& = \\
& \left\{ \begin{array}{ll} \{(\sigma[\mathbf{r} \mapsto 0], \text{ph}, (\text{update}(t, cp/a, a[\emptyset]), f))\} & \text{iff } \llbracket \mathbf{path} \rrbracket_\sigma \equiv cp/a \\ & \wedge \text{resolve}(t, cp, []) \neq \mathbf{undefined} \\ & \wedge \text{resolve}(t, cp/a, []) = \mathbf{undefined} \\ \{\{\}\} & \text{otherwise} \end{array} \right. \\
& [\mathbf{r} := \mathbf{rmdir}(\mathbf{path})]((\sigma, \text{ph}, (t, f))) \\
& = \\
& \left\{ \begin{array}{ll} \{(\sigma[\mathbf{r} \mapsto 0], \text{ph}, (\text{remove}(t, cp/a), f))\} & \text{iff } \llbracket \mathbf{path} \rrbracket_\sigma \equiv cp/a \wedge \text{resolve}(t, cp, []) = a[\emptyset] \\ \{\{\}\} & \text{otherwise} \end{array} \right. \\
& [\mathbf{r} := \mathbf{unlink}(\mathbf{path})]((\sigma, \text{ph}, (t, f))) \\
& = \\
& \left\{ \begin{array}{ll} \{(\sigma[\mathbf{r} \mapsto 0], \text{ph}, (\text{remove}(t, cp/a), f))\} & \text{iff } \llbracket \mathbf{path} \rrbracket_\sigma \equiv cp/a \wedge \text{resolve}(t, cp, []) \in \{a:\iota, a@cp'\} \\ \{\{\}\} & \text{otherwise} \end{array} \right. \\
& [\mathbf{r} := \mathbf{link}(\mathbf{existing}, \mathbf{new})]((\sigma, \text{ph}, (t, f))) \\
& = \\
& \left\{ \begin{array}{ll} \{(\sigma[\mathbf{r} \mapsto 0], \text{ph}, (\text{update}(t, cp'/b, b:\iota), f))\} & \text{iff } \sigma(\mathbf{existing}) = cp/a \wedge \sigma(\mathbf{new}) = cp'/b \\ & \wedge \text{resolve}(t, cp/a, []) = a:\iota \wedge \text{resolve}(t, cp'/b, []) = \mathbf{undefined} \wedge \text{resolve}(t, cp'/b, []) \neq \mathbf{undefined} \\ \{\{\}\} & \text{otherwise} \end{array} \right. \\
& [\mathbf{r} := \mathbf{rename}(\mathbf{old}, \mathbf{new})]((\sigma, \text{ph}, (t, f))) \\
& = \\
& \left\{ \begin{array}{ll} \{(\sigma[\mathbf{r} \mapsto 0], \text{ph}, (\text{update}(t, cp'/b, b:\iota'), f))\} & \text{iff } \sigma(\mathbf{old}) = cp/a \wedge \sigma(\mathbf{new}) = cp'/b \\ & \wedge \text{resolve}(t, cp/a, []) = a:\iota \wedge \text{resolve}(t, cp') \neq \mathbf{undefined} \\ & \wedge (\text{resolve}(t, cp'/b, []) = \mathbf{undefined} \vee \text{resolve}(t, cp'/b, []) = b:\iota') \\ \{(\sigma[\mathbf{r} \mapsto 0], \text{ph}, (\text{update}(t, cp'/b, b[cd]), f))\} & \text{iff } \sigma(\mathbf{old}) = cp/a \wedge \sigma(\mathbf{new}) = cp'/b \\ & \wedge \text{resolve}(t, cp/a, []) = a[cd] \\ & \wedge (\text{resolve}(t, cp'/b, []) = b[\emptyset] \vee \text{resolve}(t, cp'/b, []) = \mathbf{undefined}) \\ \{(\sigma[\mathbf{r} \mapsto 0], \text{ph}, (t, f))\} & \text{iff } \sigma(\mathbf{old}) = cp/a \wedge \sigma(\mathbf{new}) = cp'/b \\ & \wedge \text{resolve}(t, cp/a, []) \neq \mathbf{undefined} \wedge \text{resolve}(t, cp'/b, []) \neq \mathbf{undefined} \wedge \text{lin}(cp/a) = \text{lin}(cp'/b) \\ \{\{\}\} & \text{otherwise} \end{array} \right.
\end{aligned}$$

Figure 16: Interpretation function for the primitive file system commands.

tion 26 is the composition of instrumented program states ($\bullet$) lifted to sets. On the other hand, our program logic is based on assertions and use logical connectives, instead of working directly with sets. We interpret assertions as sets of instrumented program states, with $\llbracket - \rrbracket : \text{ASRTS} \rightarrow \mathcal{P}(\text{ISTATES})$, as given in definition 25. This allows us to establish an equivalence between assertions and instrumented program states. The following lemma establishes an equivalence between the logical connective $*$ and the set composition operator $*_{\mathbf{v}}$.

Lemma 1 ($*$ equivalence). *For all $P, Q \in \text{ASRTS}$, it is the case that:*

$$\llbracket P * Q \rrbracket = \llbracket P \rrbracket *_{\mathbf{v}} \llbracket Q \rrbracket$$

Proof. Calculating using the appropriate definitions:

$$\begin{aligned}
& \llbracket P * Q \rrbracket \\
& = \{is \in \text{ISTATES} \mid is \models P * Q\} \\
& = \left\{ \begin{array}{l} (\sigma, \text{ph}, (it, f)) \bullet (\sigma', \text{ph}', (it', f')) \mid \\ \sigma, \text{ph}, (it, f) \models P \wedge \sigma', \text{ph}', (it', f') \models Q \end{array} \right\} \\
& = \{p \bullet q \mid p \in \llbracket P \rrbracket, q \in \llbracket Q \rrbracket\} \\
& = \llbracket P \rrbracket *_{\mathbf{v}} \llbracket Q \rrbracket
\end{aligned}$$

□

Definition 32 (Effect Instrumented State Composition). *We define the following partial binary composition on instrumented program states:*

$$\bullet_{\wedge} : \text{ISTATES} \times \text{ISTATES} \rightarrow \text{ISTATES}$$

$$\begin{aligned}
& (\sigma, \text{ph}, (\top[d], f)) \bullet_{\wedge} (\sigma', \text{ph}', (\top[d'], f')) \triangleq \\
& (\sigma \uplus \sigma', \text{ph} \uplus \text{ph}', (\top[d \wedge d'], f \uplus f'))
\end{aligned}$$

where:

$$+_{\wedge} : \text{IDIRS} \times \text{IDIRS} \rightarrow \text{IDIRS}$$

$$d +_{\wedge} d' = d'' \stackrel{\text{def}}{\iff} d \otimes d' \equiv_{\wedge} d'' \wedge \text{canonical}(d'')$$

where the predicate *canonical* is defined as follows:

$$\begin{aligned}
& \text{canonical}(\emptyset) \triangleq \mathbf{true} \\
& \text{canonical}(a_{\pi}[d]) \triangleq \text{canonical}(d) \\
& \text{canonical}(a_{\pi}:\iota) \triangleq \mathbf{true} \\
& \text{canonical}(\langle S \rangle_{\pi}) \triangleq \mathbf{true} \\
& \text{canonical}(d \otimes d') \triangleq \text{canonical}(d) \wedge \text{canonical}(d') \\
& \quad \wedge \text{names}(d) \cap \text{names}(d') = \emptyset
\end{aligned}$$

in all other cases canonical is **false**, and:

$$\text{names} : \text{IDIRS} \rightarrow \text{FNAMES}$$

$$\begin{aligned} \text{names}(\emptyset) &\triangleq \emptyset \\ \text{names}(a_\pi[d]) &\triangleq \{a\} \\ \text{names}(a_\pi:\iota) &\triangleq \{a\} \\ \text{names}(\widehat{\langle S \rangle}_\pi) &\triangleq \{S\} \\ \text{names}(d \otimes d') &\triangleq \text{names}(d) \cup \text{names}(d') \end{aligned}$$

$\bullet_\wedge$ induces the following composition on sets of instrumented states:

$$p_1 \hat{*}_v p_2 \triangleq \{m_1 \bullet_\wedge m_2 \mid m_1 \in p_1, m_2 \in p_2\}$$

Lemma 2 ($\hat{*}$ equivalence). *For all $P, Q \in \text{ASRTS}$, it is the case that:*

$$\llbracket P \hat{*} Q \rrbracket = \llbracket P \rrbracket \hat{*}_v \llbracket Q \rrbracket$$

Proof. Calculating using the appropriate definitions:

$$\begin{aligned} \llbracket P \hat{*} Q \rrbracket &= \{is \in \text{ISTATES} \mid is \models P \hat{*} Q\} \\ &= \left\{ \begin{array}{l} (\sigma, \text{ph}, (it, f)) \bullet_\wedge (\sigma', \text{ph}', (it', f')) \mid \\ \sigma, \text{ph}, (it, f) \models P \wedge \sigma', \text{ph}', (it', f') \models Q \end{array} \right\} \\ &= \{p \bullet q \mid p \in \llbracket P \rrbracket, q \in \llbracket Q \rrbracket\} \\ &= \llbracket P \rrbracket \hat{*}_v \llbracket Q \rrbracket \end{aligned}$$

□

Given the parameters of definition 26, we establish the following properties as required by the framework. We implicitly use lemma 1 to relate the properties as stated for views to the assertions of our program logic.

Definition 33 (Views Properties). *The Views framework requires the following properties to be satisfied:*

1. **Atomic Soundness:** *It is sufficient to show that for every $(P, \mathcal{A}, Q) \in \text{AXIOMS}$, where $\mathcal{A} \in \mathbb{C}_{fs}$, and every $R \in \text{ASRTS}$,*

$$[\mathcal{A}] \left(\llbracket \llbracket P \rrbracket *_{\text{v}} \llbracket R \rrbracket \rrbracket_{\llbracket P \rrbracket *_{\text{v}} \llbracket R \rrbracket} \right) \subseteq \llbracket \llbracket Q \rrbracket *_{\text{v}} \llbracket R \rrbracket \rrbracket_{\llbracket P \rrbracket *_{\text{v}} \llbracket R \rrbracket}$$

2. **Join Distributivity:** $p *_{\text{v}} \bigvee \{q_i\}_{i \in I} = \bigvee \{p *_{\text{v}} q_i\}_{i \in I}$ where $p, q \in \text{ISTATES}$.
3. **Join Morphism:** $\llbracket \bigvee \{p_i\}_{i \in I} \rrbracket = \bigcup \{ \llbracket p_i \rrbracket \}_{i \in I}$ where $p \in \text{ISTATES}$.

We establish the properties with the following two lemmas.

Lemma 3 (Atomic Soundness). *The atomic soundness property holds.*

Proof. First, consider the `mkdir` axiom from figure 5. For the precondition we have:

$$\begin{aligned} &\llbracket \text{var}(\mathbf{r}, -) * \text{pvar}(\text{path}, p/a_1) * \text{tree}(p/\widehat{a}_1) \rrbracket \\ &= \\ &\left\{ \begin{array}{l} \sigma, \emptyset, (it, \emptyset) \in \text{ISTATES} \mid \\ \exists \sigma_1, \sigma_2, v. \sigma_1 = \mathbf{r} \mapsto v \wedge \sigma_2 = \text{path} \mapsto \text{strippath}(p/a_1) \\ \wedge \emptyset, \emptyset, (it, \emptyset) \models \text{tree}(p/\widehat{a}_1) \wedge \sigma = \sigma_1 \uplus \sigma_2 \end{array} \right\} \\ &= p \end{aligned}$$

For every, $R \in \text{ASRTS}$

$$\begin{aligned} &p *_{\text{v}} \llbracket R \rrbracket \\ &= \\ &\left\{ \begin{array}{l} (\sigma, \text{ph}, ifs) \in \text{ISTATES} \mid \\ is \in \llbracket R \rrbracket, \\ \exists \sigma_1, \sigma_2, v. \sigma_1 = \mathbf{r} \mapsto v \wedge \sigma_2 = \text{path} \mapsto \text{strippath}(p/a_1) \\ \wedge \emptyset, \emptyset, (it, \emptyset) \models \text{tree}(p/\widehat{a}_1) \\ \wedge (\sigma, \text{ph}, ifs) = (\sigma_1 \uplus \sigma_2, \emptyset, (it, \emptyset)) \bullet is \end{array} \right\} \\ &= iss \end{aligned}$$

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

$$\begin{aligned} &\llbracket iss \rrbracket_{iss} \\ &= \\ &\left\{ \begin{array}{l} (\sigma, \text{ph}, (t, f)) \in \text{STATES} \mid \\ \llbracket \text{path} \rrbracket_\sigma = \text{strippath}(p)/a \\ \wedge \text{resolve}(t, \text{strippath}(p), []) \neq \text{undefined} \\ \wedge \text{resolve}(t, \text{strippath}(p)/a, []) = \text{undefined} \end{array} \right\} \end{aligned}$$

Then, by the command interpretation:

$$\begin{aligned} &[\mathbf{r} := \text{mkdir}(\text{path})] (\llbracket iss \rrbracket_{iss}) \\ &= \\ &\left\{ \begin{array}{l} (\sigma, \text{ph}, (t, f)) \in \text{STATES} \mid \\ \sigma(\mathbf{r}) = 0 \wedge \llbracket \text{path} \rrbracket_\sigma = \text{strippath}(p)/a \\ \wedge \text{resolve}(t, \llbracket \text{path} \rrbracket_\sigma, []) = a[\emptyset] \end{array} \right\} \end{aligned}$$

For the postcondition we have:

$$\begin{aligned} &\llbracket \text{var}(\mathbf{r}, 0) * \text{pvar}(\text{path}, p/a_1) * \text{tree}(p/a_1[\emptyset]) \rrbracket \\ &= \\ &\left\{ \begin{array}{l} \sigma, \emptyset, (it, \emptyset) \in \text{ISTATES} \mid \\ \exists \sigma_1, \sigma_2. \sigma_1 = \mathbf{r} \mapsto 0 \wedge \sigma_2 = \text{path} \mapsto \text{strippath}(p/a_1) \\ \wedge \emptyset, \emptyset, (it, \emptyset) \models \text{tree}(p/a_1[\emptyset]) \wedge \sigma = \sigma_1 \uplus \sigma_2 \end{array} \right\} \\ &= \\ &q \end{aligned}$$

with

$$\begin{aligned} &q *_{\text{v}} \llbracket R \rrbracket \\ &= \\ &\left\{ \begin{array}{l} (\sigma, \text{ph}, ifs) \in \text{ISTATES} \mid \\ is \in \llbracket R \rrbracket, \\ \exists \sigma_1, \sigma_2. \sigma_1 = \mathbf{r} \mapsto 0 \wedge \sigma_2 = \text{path} \mapsto \text{strippath}(p/a_1) \\ \wedge \emptyset, \emptyset, (it, \emptyset) \models \text{tree}(p/a_1[\emptyset]) \\ \wedge (\sigma, \text{ph}, ifs) = (\sigma_1 \uplus \sigma_2, \emptyset, (it, \emptyset)) \bullet is \end{array} \right\} \\ &= iss' \end{aligned}$$

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

$$\begin{aligned} &\llbracket iss' \rrbracket_{iss} \\ &= \\ &\left\{ \begin{array}{l} (\sigma, \text{ph}, (t, f)) \in \text{STATES} \mid \\ \sigma(\mathbf{r}) = 0 \wedge \llbracket \text{path} \rrbracket_\sigma = \text{strippath}(p)/a \\ \wedge \text{resolve}(t, \llbracket \text{path} \rrbracket_\sigma, []) = a[\emptyset] \end{array} \right\} \end{aligned}$$

From which we see that: $[r := \text{mkdir}(\text{path})] ([iss]_{iss}) = [iss']_{iss}$ which establishes the atomic soundness property for this axiom.

Now consider the `rmdir` axiom from figure 5. For the precondition we have:

$$\begin{aligned} & \llbracket \text{var}(r, -) * \text{pvar}(\text{path}, p/a_1) * \text{tree}(p/a_1[\emptyset]) \rrbracket_e \\ &= \\ & \left\{ \begin{array}{l} (\sigma, \emptyset, (it, \emptyset)) \in \text{ISTATES} \mid \\ \exists \sigma_1, \sigma_2, v. \sigma_1 = r \mapsto v \wedge \sigma_2 = \text{path} \mapsto \text{strippath}(p/a_1) \\ \wedge \emptyset, \emptyset, (it, \emptyset) \models \text{tree}(p/a_1[\emptyset]) \wedge \sigma = \sigma_1 \uplus \sigma_2 \end{array} \right\} \\ &= p \end{aligned}$$

For every $R \in \text{ASRTS}$

$$\begin{aligned} & p *_v \llbracket R \rrbracket \\ &= \\ & \left\{ \begin{array}{l} (\sigma, \text{ph}, ifs) \in \text{ISTATES} \mid \\ is \in \llbracket R \rrbracket, \\ \exists \sigma_1, \sigma_2, v. \sigma_1 = r \mapsto v \wedge \sigma_2 = \text{path} \mapsto \text{strippath}(p/a_1) \\ \wedge \emptyset, \emptyset, (it, \emptyset) \models \text{tree}(p/a_1[\emptyset]) \\ \wedge (\sigma, \text{ph}, ifs) = (\sigma_1 \uplus \sigma_2, \emptyset, (it, \emptyset)) \bullet is \end{array} \right\} \\ &= iss \end{aligned}$$

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

$$\begin{aligned} & [iss]_{iss} \\ &= \\ & \left\{ \begin{array}{l} (\sigma, \text{ph}, (t, f)) \in \text{STATES} \mid \\ \llbracket \text{path} \rrbracket_\sigma = \text{strippath}(p)/a \\ \wedge \text{resolve}(t, \text{strippath}(p)/a, []) = a[\emptyset] \end{array} \right\} \end{aligned}$$

Then by the command interpretation:

$$\begin{aligned} & [r := \text{rmdir}(\text{path})] ([iss]_{iss}) \\ &= \\ & \left\{ \begin{array}{l} (\sigma, \text{ph}, (t, f)) \in \text{STATES} \mid \\ \sigma(r) = 0 \wedge \llbracket \text{path} \rrbracket_\sigma = \text{strippath}(p)/a \\ \wedge \text{resolve}(t, \text{strippath}(p)/a, []) = \text{undefined} \\ \wedge \text{resolve}(t, \text{lin}(\text{strippath}(p)), []) \neq \text{undefined} \end{array} \right\} \end{aligned}$$

For the postcondition we have:

$$\begin{aligned} & \llbracket \text{var}(r, 0) * \text{pvar}(\text{path}, p/a_1) * \text{tree}(p/\hat{a}_1) \rrbracket_e \\ &= \\ & \left\{ \begin{array}{l} (\sigma, \emptyset, (it, \emptyset)) \in \text{ISTATES} \mid \\ \exists \sigma_1, \sigma_2. \sigma_1 = r \mapsto 0 \wedge \sigma_2 = \text{path} \mapsto \text{strippath}(p/a_1) \\ \wedge \emptyset, \emptyset, (it, \emptyset) \models \text{tree}(p/\hat{a}_1) \wedge \sigma = \sigma_1 \uplus \sigma_2 \end{array} \right\} \\ &= q \end{aligned}$$

with

$$\begin{aligned} & q *_v \llbracket R \rrbracket \\ &= \\ & \left\{ \begin{array}{l} (\sigma, \text{ph}, ifs) \in \text{ISTATES} \mid \\ is \in \text{ISTATES}, \\ \exists \sigma_1, \sigma_2. \sigma_1 = r \mapsto 0 \wedge \sigma_2 = \text{path} \mapsto \text{strippath}(p/a_1) \\ \wedge \emptyset, \emptyset, (it, \emptyset) \models \text{tree}(p/\hat{a}_1) \\ \wedge (\sigma, \text{ph}, ifs) = (\sigma_1 \uplus \sigma_2, \emptyset, (it, \emptyset)) \bullet is \end{array} \right\} \\ &= iss' \end{aligned}$$

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

$$\begin{aligned} & [iss']_{iss} \\ &= \\ & \left\{ \begin{array}{l} (\sigma, \text{ph}, (t, f)) \in \text{STATES} \mid \\ \sigma(r) = 0 \wedge \llbracket \text{path} \rrbracket_\sigma = \text{strippath}(p)/a \\ \wedge \text{resolve}(t, \text{strippath}(p)/a, []) = \text{undefined} \\ \wedge \text{resolve}(t, \text{lin}(\text{strippath}(p)), []) \neq \text{undefined} \end{array} \right\} \end{aligned}$$

From which we see that: $[r := \text{rmdir}(\text{path})] ([iss]_{iss}) = [iss']_{iss}$, which establishes the atomic soundness property for this axiom.

For the rest of the axioms the proof is similar. $\square$

Lemma 4 (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 20. $\square$

The views properties of definition 33 establish soundness of the basic program logic, excluding the effect frame rule. We justify the soundness of the effect frame rule by showing that it is an instance of the generalised frame rule of views.

Definition 34 (Effect Frame Instantiation). *We define the effect frame rule of our logic as an instance of the generalised frame rule of views:*

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

where we instantiate the views transforming function f as:

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

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

For the above instantiation to be sound, we establish the following lemma, as required by views.

Lemma 5 (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.*

$$\begin{aligned} & \forall P, Q, R, R' \in \text{ASRTS}, \mathcal{A} \in \mathbb{C}_{fs}. \\ & [\mathcal{A}] \left(\llbracket \llbracket P \rrbracket *_v \llbracket R \rrbracket \rrbracket_p \right) \subseteq \llbracket \llbracket Q \rrbracket *_v \llbracket R \rrbracket \rrbracket_p \end{aligned}$$

$$\begin{aligned} & \implies \\ & [\mathcal{A}] \left(\llbracket f_{Q,R'}(\llbracket P \rrbracket *_v \llbracket R \rrbracket) \rrbracket_r \right) \subseteq \llbracket f_{Q,R'}(\llbracket Q \rrbracket *_v \llbracket R \rrbracket) \rrbracket_r \end{aligned}$$

where $p = \llbracket P \rrbracket *_v \llbracket R \rrbracket$, and $r = f_{Q,R'}(\llbracket P \rrbracket *_v \llbracket R \rrbracket)$.

Proof. By atomic soundness and lemma 3 the top of the implication is true. By the transformation function, for the precondition we have:

$$\llbracket f_{Q,R'}(\llbracket P \rrbracket *_v \llbracket R \rrbracket) \rrbracket_r = \llbracket \llbracket P \rrbracket *_v \llbracket R \rrbracket *_v \llbracket R' \rrbracket \rrbracket_r$$

and for the postcondition, by lemmas 1 and 2 we have:

$$\llbracket f_{Q,R'}(\llbracket Q \rrbracket *_{\mathbf{v}} \llbracket R \rrbracket) \rrbracket_r = \llbracket (\llbracket Q \rrbracket *_{\mathbf{v}} \llbracket R \rrbracket) \hat{*} \llbracket R' \rrbracket \rrbracket_r$$

Therefore it suffices to show that:

$$\begin{aligned} & \forall P, Q, R \in \text{ASRTS}, \mathcal{A} \in \mathbb{C}_{\text{fs}}. \\ & [\mathcal{A}] \left(\llbracket \llbracket P \rrbracket *_{\mathbf{v}} \llbracket R \rrbracket \rrbracket_p \right) \subseteq \llbracket \llbracket Q \rrbracket \hat{*}_{\mathbf{v}} \llbracket R \rrbracket \rrbracket_p \end{aligned}$$

where $p = \llbracket P \rrbracket *_{\mathbf{v}} \llbracket R \rrbracket$.

Consider the right hand side of $\subseteq$. The $\hat{*}_{\mathbf{v}}$ propagates update effects between Q and R according to $\equiv_{\wedge}$ (definition 12). Now consider $\llbracket \llbracket Q \rrbracket *_{\mathbf{v}} \llbracket R \rrbracket \rrbracket_p$. According to definition 31, because $p \neq \llbracket Q \rrbracket *_{\mathbf{v}} \llbracket R \rrbracket$, the reification is going to force any update effects to be considered via $\equiv_{\wedge}$ before reifying to concrete program states. Both forms have the same effects and therefore, they reify to exactly the same concrete program states. That is:

$$\llbracket \llbracket Q \rrbracket \hat{*}_{\mathbf{v}} \llbracket R \rrbracket \rrbracket_p = \llbracket \llbracket Q \rrbracket *_{\mathbf{v}} \llbracket R \rrbracket \rrbracket_p$$

Therefore, it suffices to show that:

$$\begin{aligned} & \forall P, Q, R \in \text{ASRTS}, \mathcal{A} \in \mathbb{C}_{\text{fs}}. \\ & [\mathcal{A}] \left(\llbracket \llbracket P \rrbracket *_{\mathbf{v}} \llbracket R \rrbracket \rrbracket_p \right) \subseteq \llbracket \llbracket Q \rrbracket *_{\mathbf{v}} \llbracket R \rrbracket \rrbracket_p \end{aligned}$$

where $p = \llbracket P \rrbracket *_{\mathbf{v}} \llbracket R \rrbracket$. But this already holds by lemma 3. $\square$

Having established these properties, the framework provides the following 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 $st \in \llbracket \llbracket P \rrbracket \rrbracket_{\llbracket P \rrbracket}$ and $st' \in \text{STATES}$, if $\mathbb{C}, st \rightsquigarrow^* \text{skip}, st'$ then $st' \in \llbracket \llbracket Q \rrbracket \rrbracket_{\llbracket P \rrbracket}$.*

Proof. By lemmas 1, 3, 4 and 5 the partial correctness result follows from views. The fault avoiding result follows from the fact that $\mathbb{C} \notin \text{STATES}$. $\square$

C. Errors

So far, in figure 5 we have specified when the POSIX commands succeed. In any other case they fail. In POSIX this generally means that an error code is returned to explain the cause of the failure. Programmers should examine this error code, and take remedial action if the command failed. Errors are also sometimes required to perform elementary tasks. For example, to see if the path `path` resolves in the file system, one typically performs `stat(path)` and checks if an error is returned. If an error is not returned then `path` resolves.

In our fragment of POSIX file systems we consider the following error cases:

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

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

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

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

On the other hand, the file system is a shared resource, not owned by any one program. Even in our non-concurrent system, there is nothing to stop someone mutating the file system against our expectations before our program runs. Thus, it is sensible and expected that programmers use errors as a form of “defensive programming”, to cater for unanticipated

Command	Possible errors
<code>r := mkdir(path)</code>	<code>E_NOENT</code> , <code>E_NOTDIR</code> , <code>E_EXIST</code>
<code>r := rmdir(path)</code>	<code>E_NOENT</code> , <code>E_NOTDIR</code> , <code>E_NOTEMPTY</code>
<code>r := link(to, from)</code>	<code>E_NOENT</code> , <code>E_NOTDIR</code> , <code>E_PERM</code> <code>E_EXIST</code> (only for the from parameter)
<code>r := unlink(path)</code>	<code>E_NOENT</code> , <code>E_NOTDIR</code> , <code>E_PERM</code>
<code>r := rename(old, new)</code>	<code>E_NOENT</code> , <code>E_NOTDIR</code> , <code>E_NOTEMPTY</code> , <code>E_ISDIR</code> , <code>E_INVALID</code>
<code>t := stat(path)</code>	<code>E_NOENT</code> , <code>E_NOTDIR</code>
<code>fd := open(path, flags)</code>	<code>E_NOENT</code> , <code>E_NOTDIR</code> , <code>E_ISDIR</code>
<code>o := lseek(fd, off, wh)</code>	No errors
<code>sz := write(fd, str)</code>	No errors
<code>str := read(fd, size)</code>	No errors
<code>close(fd)</code>	No errors
<code>h := opendir(path)</code>	<code>E_NOENT</code> , <code>E_NOTDIR</code>
<code>s := readdir(h)</code>	No errors
<code>closedir(h)</code>	No errors

Table 1: Possible errors that each command can return

external changes to the file system structure. We can use our reasoning to determine if a program detects all possible error cases, and more importantly how it handles them.

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

The properties of each error code are independent of its use by any one command. For example, failing to resolve a path correctly is the same whether the command that failed is `open` or `mkdir`. As such, in figure 17 we define predicates that describe the states on which resolving a given path would result in each error. We name the predicates identically to the error they represent. There are two variants of `E_NOENT`: `E_NOENT*(path)` is satisfied when `path` or any prefix of `path` do not resolve, and `E_NOENT(path)` is satisfied simply when any prefix of `path` does not resolve. Similarly there are two variants of `E_NOTDIR`.

With these error predicates, we can axiomatize commands that may return errors. Consider the `stat` command, which has the following two success case axioms:

$$\begin{aligned}
&\{\text{var}(t, -) * \text{pvar}(\text{path}, p) * \text{tree}(p/\emptyset)\} \\
&\quad t := \text{stat}(\text{path}) \\
&\{\text{var}(t, D) * \text{pvar}(\text{path}, p) * \text{tree}(p/\emptyset)\} \\
&\{\text{var}(t, -) * \text{pvar}(\text{path}, p/a_\pi) * \text{tree}(p/a_\pi : \iota)\} \\
&\quad t := \text{stat}(\text{path}) \\
&\{\text{var}(t, F) * \text{pvar}(\text{path}, p/a_\pi) * \text{tree}(p/a_\pi : \iota)\}
\end{aligned}$$

Using the error detection predicates of figure 17, `stat` has the following two error case axioms:

$$\begin{aligned}
&\{\text{var}(t, -) * \text{var}(\text{errno}, -) * \text{E_NOENT}(\text{path}, v) \wedge t\} \\
&\quad t := \text{stat}(\text{path}) \\
&\{\text{var}(t, -1) * \text{var}(\text{errno}, \text{E_NOENT}) * \text{var}(\text{path}, v) * t\}
\end{aligned}$$

$$\begin{aligned}
&\{\text{var}(t, -) * \text{var}(\text{errno}, -) * \text{E_NOTDIR}(\text{path}) \wedge t\} \\
&\quad t := \text{stat}(\text{path}) \\
&\{\text{var}(t, -1) * \text{var}(\text{errno}, \text{E_NOTDIR}) * \text{var}(\text{path}, v) * t\}
\end{aligned}$$

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

D. File IO and Heap Small Axioms

In this section we give small axioms for regular file IO commands not covered in figure 5, as well as small axioms for heap memory allocation (`malloc`) and deallocation (`free`).

We use the following derived assertions in the specifications:

$$\begin{aligned}
\text{br}(E) &\triangleq E = \emptyset ? 0 : E \\
\text{fh}(\iota, E, E_0, E_1, \dots, E_n) &\triangleq \text{fh}(\iota, E + 0, E_0) * \text{fh}(\iota, E + 1, E_1) \\
&\quad * \dots * \text{fh}(\iota, E + n, E_n) \\
\text{ptr}(E, E_0, E_1, \dots, E_n) &\triangleq \text{ptr}(E + 0, E_0) * \text{ptr}(E + 1, E_1) \\
&\quad * \dots * \text{ptr}(E + n, E_n) \\
\vec{E}_n &\triangleq E_0, E_1, \dots, E_{n-1} \\
E_1 < E_2 &\triangleq (E_1 < E_2) \wedge \text{emp}
\end{aligned}$$

The small axioms for the file IO commands of our fragment are given in figure 18. The small axioms for `malloc` and `free` are given in figure 19.

E. Symbolic Links

Non-linearity of pathnames does not come just from dot and dot-dot, on which we have focused in the rest of the paper. Symbolic links, which are a special type of file containing a pathname, induce non-linearity by affecting pathname resolution. During pathname resolution, if the current pathname component identifies a symbolic link file in the file system structure, its value (the contained pathname) is prepended to the remaining unresolved pathname and the resolution process restarts. Symbolic links do not change the fundamental

$$\begin{aligned}
\text{E_NOENT}^*(\text{path}, v) &\triangleq \text{var}(\text{path}, v) \wedge ((v \doteq 0) \vee (\exists p, a, p', \pi. \text{pvar}(\text{path}, p/a_\pi/p') * \text{tree}(p/\widehat{a}_\pi))) \\
\text{E_NOENT}(\text{path}, v) &\triangleq \text{var}(\text{path}, v) \wedge ((v \doteq 0) \vee (\exists p, a, p', \pi. \text{pvar}(\text{path}, p/a_\pi/p') * \neg(p' \equiv \cdot) \wedge \text{tree}(p/\widehat{a}_\pi))) \\
\text{E_NOTDIR}^*(\text{path}) &\triangleq \exists p, a, p', \pi, \iota. \text{pvar}(\text{path}, p/a_\pi/p') * \text{tree}(p/a_\pi : \iota) \\
\text{E_NOTDIR}(\text{path}) &\triangleq \exists p, a, p, \pi, \iota. \text{pvar}(\text{path}, p/a_\pi/p') * \neg(p' \equiv \cdot) \wedge \text{tree}(p/a_\pi : \iota) \\
\text{E_EXIST}(\text{path}) &\triangleq \exists p, a, \pi. \text{var}(\text{path}, p/a_\pi) \wedge \text{tree}(p/\text{ent}_\pi(\pi))a \\
\text{E_NOTEEMPTY}(\text{path}) &\triangleq \exists p, a, \pi. \text{pvar}(\text{path}, p) \wedge \text{tree}(p/\text{ent}_\pi(\pi))a \\
\text{E_PERM}(\text{path}) &\triangleq \exists p, a, \pi. \text{pvar}(\text{path}, p/a_\pi) \wedge \text{tree}(p/a_\pi[\text{true}]) \\
\text{E_ISDIR}(\text{old}, \text{new}) &\triangleq \exists p, p', a, b, \iota, \pi, \pi'. \text{pvar}(\text{old}, p/a_\pi) * \text{pvar}(\text{new}, p/b_{\pi'}) \\
&\quad * \text{tree}(p/a_\pi : \iota) * \text{tree}(p'/a_{\pi'}[\text{true}]) \\
&\quad \exists p, p', p'', a, b, \pi, \pi', X. \\
\text{E_INVAL}(\text{old}, \text{new}) &\triangleq \text{pvar}(\text{old}, p/a_\pi) * \text{pvar}(\text{new}, p'/p''/b_\pi) \wedge \text{lin}(p) = \text{lin}(p') \\
&\quad \wedge \text{tree}(p/a_\pi[\text{true} * p''/X]) * \text{tree}(p'/p'' / (X \wedge (b_{\pi'}[\emptyset] \vee \widehat{b}_{\pi'})))
\end{aligned}$$

Figure 17: Error detection predicates.

aspects of our reasoning and require only minor modifications to our existing system described in the current section.

As a first step, we need to extend our concrete and instrumented file system states to reflect the addition of the symbolic link file type. We begin by extending directory forests of definition 1.

Definition 35 (Symbolic link extended directory forests). Concrete directory forests, $cd \in \text{DIRS}$, are defined by:

$$cd ::= \emptyset \mid a[cd] \mid a : \iota \mid a @ \text{cap} \mid cd \otimes cd$$

where $\emptyset$ 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 , $a @ \text{cap}$ is a symbolic link with name a and path value cap , and $\otimes$ is the composition of directories. The equivalence relation, $cd \equiv cd'$, states that $\otimes$ is commutative and associative with identity $\emptyset$. Well-formed directory forests have sibling distinct names. We only consider concrete directory forests that are well formed.

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

Similarly we extend the instrumented directory entries of definition 11, but we do not instrument the target pathname that is the value of a symbolic link entry.

Definition 36 (Symbolic link extended instrumented directories). Instrumented directory forests, $d \in \text{IDIRS}$, are defined by:

$$d ::= \emptyset \mid a_\pi[d] \mid a_\pi : \iota \mid a_\pi @ \text{cap} \mid \widehat{\langle S \rangle}_\pi \mid d \otimes d$$

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.

Furthermore, we extend the fusion equivalence axioms of definition 12 with axioms for symbolic link entries.

Definition 37 (Fusion equivalences). The fusion equivalence relation, $\equiv \subseteq \text{IDIRS} \times \text{IDIRS}$, is defined by the following axioms:

$$a_\pi[d] \otimes a_{\pi'}[d'] \equiv a_{\pi+\pi'}[d \otimes d'] \quad \text{if } \pi, \pi' > 0 \quad (9)$$

$$a_\pi[d \otimes d'] \otimes a_0[d'] \equiv a_\pi[d \otimes d'] \quad (10)$$

$$a_\pi : \iota \otimes a_{\pi'} : \iota \equiv a_{\pi+\pi'} : \iota \quad \text{if } \pi, \pi' > 0 \quad (11)$$

$$a_\pi @ \text{cap} \otimes a_{\pi'} @ \text{cap} \equiv a_{\pi+\pi'} @ \text{cap} \quad \text{if } \pi, \pi' > 0 \quad (12)$$

$$\widehat{\langle S \rangle}_\pi \otimes \widehat{\langle S' \rangle}_{\pi'} \equiv \widehat{\langle S \uplus S' \rangle}_{\pi} \quad \text{if } \pi > 0 \quad (13)$$

$$\widehat{\langle S \rangle}_\pi \otimes \widehat{\langle S \rangle}_{\pi'} \equiv \widehat{\langle S \rangle}_{\pi+\pi'} \quad \text{if } \pi, \pi' > 0 \quad (14)$$

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

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

$$\widehat{\langle \{a\} \rangle}_1 \otimes a_0[d] \equiv_\wedge \widehat{\langle \{a\} \rangle}_1 \quad (15)$$

$$a_1 : \iota \otimes a_0[d] \equiv_\wedge a_1 : \iota \quad (16)$$

$$a_1 @ \text{cap} \otimes a_0[d] \equiv_\wedge a_1 @ \text{cap} \quad (17)$$

$$a_1[d] \otimes a_0[d'] \equiv_\wedge a_1[d] \quad \text{if } d \otimes d' \neq d \quad (18)$$

The only additions to definition 12 are equations (12) and (17) which provide fusion and effect fusion for symbolic link entries respectively.

In the previous sections, we have used instrumented pathnames (definition 21) to relate the concrete pathnames (definition 5) used by programs with their local tree footprints. However, allowing symbolic links implies that we cannot infer whether a pathname component refers to a directory or a symbolic link just by examining the pathname structure. For example, in the pathname `/usr/lib/tex` any component could be referring to a symbolic link, meaning that just by examining the pathname we do not know whether the parent

$$\begin{aligned}
& \{ \text{var}(\text{fd}, -) * \text{pvar}(\text{path}, p/a_1) * \text{tree}(p/\widehat{a}_1) \} \\
& \quad \text{fd} := \text{open}(\text{path}, \text{flags}) \\
& \{ \exists \iota. (\text{fd}(\text{fd}, \iota, 0) * \text{pvar}(\text{path}, p/a_1) * \text{tree}(p/a_1 : \iota) * \text{fh}(\iota, 0, \vartheta)) \} \\
& \{ \text{var}(\text{fd}, -) * \text{pvar}(\text{path}, p/a_\pi) * \text{var}(\text{flags}, 0) * \text{tree}(p/a_\pi : \iota) \} \\
& \quad \text{fd} := \text{open}(\text{path}, \text{flags}) \\
& \{ \text{fd}(\text{fd}, \iota, 0) * \text{pvar}(\text{path}, p/a_\pi) * \text{var}(\text{flags}, 0) * \text{tree}(p/a_\pi : \iota) \} \\
& \{ \text{var}(\text{fd}, -) * \text{pvar}(\text{path}, p/a_\pi) * \text{var}(\text{flags}, 0_TRUNC) * \text{tree}(p/a_\pi : \iota) * \text{fh}(\iota, 0, \vec{b}_n) * \text{fh}(\iota, n, \vartheta) \} \\
& \quad \text{fd} := \text{open}(\text{path}, \text{flags}) \\
& \{ \text{fd}(\text{fd}, \iota, 0) * \text{pvar}(\text{path}, p/a_\pi) * \text{var}(\text{flags}, 0_TRUNC) * \text{tree}(p/a_\pi : \iota) * \text{fh}(\iota, 0, \vartheta) \} \\
& \quad \{ \text{fd}(\text{fd}, \iota, \text{off}) \} \\
& \quad \text{close}(\text{fd}) \\
& \quad \{ \text{var}(\text{fd}, -) \} \\
& \{ \text{var}(\text{off}, -) * \text{var}(\text{to}, o') \wedge o' \geq 0 * \text{var}(\text{wh}, \text{SEEK_SET}) * \text{fd}(\text{fd}, \iota, o) \} \\
& \quad \text{off} := \text{lseek}(\text{fd}, \text{to}, \text{wh}) \\
& \quad \{ \text{var}(\text{off}, o') * \text{var}(\text{to}, o') * \text{var}(\text{wh}, \text{SEEK_SET}) * \text{fd}(\text{fd}, \iota, o') \} \\
& \{ \text{var}(\text{off}, -) * \text{var}(\text{to}, o') \wedge (o + o' \geq 0) * \text{var}(\text{wh}, \text{SEEK_CUR}) * \text{fd} \Rightarrow \text{FD} * \text{fd}(\text{fd}, \iota, o) \} \\
& \quad \text{off} := \text{lseek}(\text{fd}, \text{to}, \text{wh}) \\
& \quad \{ \text{var}(\text{off}, o + o') * \text{var}(\text{to}, o') * \text{var}(\text{wh}, \text{SEEK_CUR}) * \text{fd}(\text{fd}, \iota, o + o') \} \\
& \{ \text{var}(\text{off}, -) * \text{var}(\text{to}, o') * \text{var}(\text{wh}, \text{SEEK_END}) \wedge (n + o' \geq 0) * \text{fd}(\text{fd}, \iota, o) * \text{fh}(\iota, n, \vartheta) \} \\
& \quad \text{off} := \text{lseek}(\text{fd}, \text{to}, \text{wh}) \\
& \quad \{ \text{var}(\text{off}, n + o') * \text{var}(\text{to}, o') * \text{var}(\text{wh}, \text{SEEK_END}) * \text{fd}(\text{fd}, \iota, n + o') * \text{fh}(\iota, n, \vartheta) \} \\
& \quad \{ \text{var}(\text{sz}, -) * \text{ptr}(\text{buf}, \vec{b}_n) * \text{fd}(\text{fd}, \iota, o) * \text{fh}(\iota, o, \vec{b}_n) \} \\
& \quad \text{sz} := \text{write}(\text{fd}, \text{buf}) \\
& \quad \{ \text{var}(\text{sz}, n) * \text{ptr}(\text{buf}, \vec{b}_n) * \text{fd}(\text{fd}, \iota, o + n) * \text{fh}(\iota, o, \vec{b}_n) \} \\
& \quad \{ \text{var}(\text{sz}, -) * \text{ptr}(\text{buf}, \vec{b}_n) * \text{fd}(\text{fd}, \iota, o) * \text{fh}(\iota, o, \vec{b}_m, \vartheta) * m \leq n \} \\
& \quad \text{sz} := \text{write}(\text{fd}, \text{buf}) \\
& \quad \{ \text{var}(\text{sz}, n) * \text{ptr}(\text{buf}, \vec{b}_n) * \text{fd}(\text{fd}, \iota, o + n) * \text{fh}(\iota, o, \vec{b}_n, \vartheta) \} \\
& \quad \{ \text{var}(\text{sz}, -) * \text{ptr}(\text{buf}, \vec{b}_n) * \text{fd}(\text{fd}, \iota, o) * \text{fh}(\iota, m, \vartheta) * m \leq o \} \\
& \quad \text{sz} := \text{write}(\text{fd}, \text{buf}) \\
& \quad \{ \text{var}(\text{sz}, n) * \text{ptr}(\text{buf}, \vec{b}_n) * \text{fd} \Rightarrow \text{FD} * \text{fd}(\text{fd}, \iota, o + n) * \text{fh}(\iota, m, \vec{0}_{o-m}, \vec{b}_n, \vartheta) \} \\
& \quad \{ \text{var}(\text{sz}, -) * \text{ptr}(\text{buf}, \vec{b}_n) * \text{var}(\text{len}, n) * \text{fd}(\text{fd}, \iota, o) * \text{fh}(\iota, o, \vec{b}_n) * b_n \neq \vartheta \} \\
& \quad \text{sz} := \text{read}(\text{fd}, \text{buf}, \text{len}) \\
& \quad \{ \text{var}(\text{sz}, n) * \text{ptr}(\text{buf}, \overrightarrow{\text{br}(b_n)}) * \text{var}(\text{len}, n) * \text{fd}(\text{fd}, \iota, o + n) * \text{fh}(\iota, o, \vec{b}_n) * \} \\
& \quad \{ \text{var}(\text{sz}, -) * \text{ptr}(\text{buf}, \vec{b}_n) * \text{var}(\text{len}, n) * \text{fd}(\text{fd}, \iota, o) * \text{fh}(\iota, o, \vec{b}_m, \vartheta) * m \leq n \} \\
& \quad \text{sz} := \text{read}(\text{fd}, \text{buf}, \text{len}) \\
& \quad \{ \text{var}(\text{sz}, m) * \text{ptr}(\text{buf}, \overrightarrow{\text{br}(b_m)}, \vec{b}_{n-m}) * \text{var}(\text{len}, n) * \text{fd}(\text{fd}, \iota, o + m) * \text{fh}(\iota, o, \vec{b}_m, \vartheta) \} \\
& \quad \{ \text{var}(\text{sz}, -) * \text{ptr}(\text{buf}, \vec{b}_n) * \text{var}(\text{len}, n) * \text{fd}(\text{fd}, \iota, o) * \text{fh}(\iota, m, \vartheta) * m \leq o \} \\
& \quad \text{sz} := \text{read}(\text{fd}, \text{buf}, \text{len}) \\
& \quad \{ \text{var}(\text{sz}, 0) * \text{ptr}(\text{buf}, \vec{b}_n) * \text{var}(\text{len}, n) * \text{fd}(\text{fd}, \iota, o) * \text{fh}(\iota, m, \vartheta) \}
\end{aligned}$$

Figure 18: Small axioms for file IO.

$$\begin{aligned}
& \{ \text{var}(\text{ptr}, -) * \text{var}(\text{size}, n) \} \\
& \quad \text{ptr} := \text{malloc}(\text{size}) \\
& \{ \text{ptr}(\text{ptr} - 1, n) * \text{ptr}(\text{ptr}, \vec{n}) \} \\
& \{ \text{ptr}(\text{ptr} - 1, n) * \text{ptr}(\text{ptr}, \vec{b}_n) \} \\
& \quad \text{free}(\text{ptr}) \\
& \{ \text{var}(\text{ptr}, -) \}
\end{aligned}$$

Figure 19: Small axioms for process byte heap.

of `tex` is actually a directory named `lib`. In order to follow the same approach, we extend the instrumentation on pathnames to include richer structural information regarding symbolic links.

Definition 38 (Symbolic link extended instrumented pathnames). *Instrumented relative pathnames with symbolic links, $rp \in \text{IRELPATHS}$, are defined by the following grammar:*

$$rp ::= . \mid .. \mid n_\pi \mid a_\pi @ ap \mid ./rp \mid ../rp \mid n_\pi / rp \mid (a_\pi @ ap) / rp$$

There exists an equivalence relation $\equiv \subset \text{IRELPATHS} \times \text{IRELPATHS}$, such that $/$ is associative with $..$ as unit. Furthermore, there exists an equivalence relation $\equiv.. \subset \text{IRELPATHS} \times \text{IRELPATHS}$ which extends $\equiv$ with:

$$n_\pi / ../rp \equiv.. rp$$

Absolute instrumented pathnames, $ap \in \text{IABSPATHS} \triangleq \{ /rp \mid rp \in \text{IRELPATHS} \}$, are instrumented paths the begin with the root directory. The equivalences on instrumented relative pathnames are naturally extended to absolute pathnames.

$$\begin{aligned}
& /rp \equiv /rp' \iff rp \equiv rp' \\
& ../rp \equiv.. /rp \\
& ap / (a_\pi @ ap') / rp \equiv.. ap' / rp
\end{aligned}$$

The set of all instrumented pathnames (with symbolic links) is $\text{IPATHS} \triangleq \text{IRELPATHS} \cup \text{IABSPATHS}$. Throughout this paper we use the symbol p to denote both instrumented relative and absolute pathnames where the type of the pathname can be easily inferred from the context. The set of instrumented linear pathnames is defined as:

$$\begin{aligned}
& \text{ILPATHS} \\
& \triangleq \\
& \left\{ \begin{array}{l} p \in \text{IPATHS} \mid \\ \exists p', p'', p''', a, \pi. p = p' / ../p'' \vee p = p' / (a_\pi @ p'') / p''' \end{array} \right\}
\end{aligned}$$

and is ranged by lp .

The function $\text{lin} : \text{IPATHS} \rightarrow \text{ILPATHS}$ produces the linear path equivalent to an instrumented path:

$$\text{lin}(p) = lp \xleftrightarrow{\text{def}} p \equiv.. lp \wedge lp \in \text{ILPATHS}$$

Note that we extend the structure of instrumented pathnames with a symbolic link component, $a_\pi @ ap$. This provides enough syntactic information that allows us to determine the footprint required to exist in order for the pathname to be resolved. For example, given the instrumented pathname $/\text{usr}_{\frac{1}{4}} / (\text{lib}_1 @ / \text{usr}_{\frac{1}{4}}) / \text{tex}_1$ we know that the entry `tex` is actually located in the `usr` directory. Note, that by using the pathname instrumentation stripping function `strippath`, which can be trivially extended to drop the permission and pathname target from a symbolic link component, we can relate the instrumented pathname $/\text{usr}_{\frac{1}{4}} / (\text{lib}_1 @ / \text{usr}_{\frac{1}{4}}) / \text{tex}_1$ to the concrete pathname $/\text{usr} / \text{lib} / \text{tex}$.

Finally, we extend the pathname reduction used for the p/d notation with rules such as the following:

$$\begin{aligned}
& (\top[c], (a_\pi @ /rp)) \downarrow (\top[(c \circ \emptyset) \otimes -], /rp) \\
& \quad \text{if } c \equiv c' \circ (a_\pi @ / \text{strippath}(rp')) \otimes - \\
& (\top[c], (a_\pi @ /rp') / rp) \downarrow (\top[(c \circ \emptyset) \otimes -], /rp' / rp) \\
& \quad \text{if } c \equiv c' \circ (a_\pi @ / \text{strippath}(rp')) \otimes -
\end{aligned}$$

POSIX allows the creation of symbolic links that may contain loops. For example, in the directory structure $\top[\text{usr}[\text{1a} @ / \text{usr} / \text{1b} \otimes \text{1b} @ / \text{usr} / \text{1a}]]$ the symbolic links `1a` and `1b` form a loop. Because resolving a pathname with symbolic link loops would never terminate, POSIX requires commands to error with the `E_LOOP` error code whenever they exist. Furthermore, POSIX restricts the total number of symbolic links that can occur within a pathname to the constant `SYMLOOP_MAX`. If a command encounters more than `SYMLOOP_MAX` symbolic links then it must error with the `E_LOOP` error code. Essentially, this means that our success case axioms must not allow symbolic link loops or more than `SYMLOOP_MAX` symbolic links to occur per pathname. Note, that the instrumented pathnames of definition 38, which are used to relate concrete pathnames with their footprints, do not allow the construction of finite instrumented pathnames with symbolic link loops. This guarantees that loops do not occur in our success axioms. To account for the `SYMLOOP_MAX` restriction we count the number of symbolic links that occur in a pathname.

Definition 39 (Symbolic Link Count). *The number of symbolic links within an instrumented pathname is given by the following function:*

$$\begin{aligned}
\text{sl\#} : \text{IPATHS} &\rightarrow \mathbb{N} \\
\text{sl\#}(\cdot) &\triangleq 0 \\
\text{sl\#}(\cdot) &\triangleq 0 \\
\text{sl\#}(a_\pi) &\triangleq 0 \\
\text{sl\#}() &\triangleq 0 \\
\text{sl\#}(n_\pi @ ap) &\triangleq 1 + \text{sl\#}(ap) \\
\text{sl\#}(\cdot / rp) &\triangleq \text{sl\#}(rp) \\
\text{sl\#}(\cdot / rp) &\triangleq \text{sl\#}(rp) \\
\text{sl\#}(a_\pi / rp) &\triangleq \text{sl\#}(rp) \\
\text{sl\#}((a_\pi @ ap) / rp) &\triangleq 1 + \text{sl\#}(ap) + \text{sl\#}(rp)
\end{aligned}$$

The value of SYMLoop_MAX is allowed to vary between implementation. Thus, we make our reasoning parametric with respect to SYMLoop_MAX.

Parameter 1 (SYMLoop_MAX). *Let SYMLoop_MAX be a natural number, with minimum value 8. This is the limit of symbolic links encountered during pathname resolution after which the pathname resolution process fails.*

Next, we make minor extensions to the core assertion language.

Definition 40 (Symbolic link extended assertion language). *We extend the assertion language constructs of figure 3 to include a symbolic link type entry in the instrumented directory assertions.*

Directory Assertions, Δ
Symbolic Link Type Entry $a_\pi @ cap$

Definition 41 (Symbolic link extended satisfaction relation). *We extend the satisfaction relation of figure 4 for the symbolic link entry assertion.*

$$d \models' a_\pi @ cap \iff d = a_\pi @ cap$$

We also introduce new derived assertions to assist us handling symbolic links and restricting assertions to less than SYMLoop_MAX symbolic link occurrences.

Other derived assertions

$$\begin{aligned}
\text{ent}_\pi(a) &\triangleq (\exists \iota. a_\pi : \iota) \vee a_\pi[\emptyset] \vee (\exists cap. a_\pi @ cap) \\
a_\pi \# x &\triangleq a_\pi : x \vee a_\pi @ x \\
\text{tree}(p / : \iota) &\triangleq \bigvee \left(\begin{array}{l} (\exists p', a, \pi. p \equiv p' / a_\pi \wedge \text{tree}(p' / a_\pi : \iota)) \\ \exists p', ap, a, \pi. p \equiv p' / a_\pi \\ \wedge \text{tree}(p' / a_\pi @ \text{strippath}(ap)) * \text{tree}(ap / : \iota) \end{array} \right) \\
\text{tree}(p /_s \Delta) &\triangleq \text{tree}(p / \Delta) \wedge \text{sl\#}(p) \leq \text{SYMLoop_MAX} \\
\text{tree}(p /_s : \iota) &\triangleq \text{tree}(p / : \iota) \wedge \text{sl\#}(p) \leq \text{SYMLoop_MAX}
\end{aligned}$$

We extend our small axioms to include cases for handling symbolic links and to include commands that are used specifically for their manipulation in figure 20.

These cover the cases where commands succeed. We also need to extend the error detection predicates of figure 17 with predicates that detect loops and more than SYMLoop_MAX symbolic link occurrences.

First, we define a predicate that detects the presence of a symbolic link loop.

Definition 42 (No Loop Predicate). *The following predicate is satisfied when a pathname is resolvable, no symbolic link loops exist and no more than SYMLoop_MAX symbolic links are traversed.*

$$\begin{aligned}
&\text{noloop}(cp, T_r, T_c, S, l) \\
&\triangleq \\
&\quad cp \equiv '\cdot' \\
&\quad \vee \left(\begin{array}{l} \exists cp', A. cp \equiv A / cp' \wedge \\ (\exists T'_c, \pi. T_c = (a_\pi[T'_c] * \text{true}) \wedge \text{noloop}(cp', T_r, T'_c, S, l)) \\ \vee \\ (\exists \iota, \pi. T_c = (a_\pi : \iota * \text{true}) \wedge cp' \equiv '\cdot') \\ \vee \\ \left(\begin{array}{l} \exists cp'', \pi. T_c = (a_\pi @ cp'' * \text{true}) \wedge cp'' \notin S \wedge l > 0 \\ \wedge \text{noloop}(cp'' / cp', T_r, T_r, S \cup \{cp''\}, l - 1) \end{array} \right) \end{array} \right)
\end{aligned}$$

Definition 43 (E_LOOP Error Predicate). *The E_LOOP error predicates are defined as:*

$$\begin{aligned}
&\text{E_LOOP}_*(\text{path}, T) \triangleq \\
&\quad \exists cp. \text{var}(\text{path}, cp) * T \wedge \neg \text{noloop}(cp, T, T, \emptyset, \text{SYMLoop_MAX}) \\
&\quad \text{E_LOOP}(\text{path}, T) \\
&\quad \triangleq \\
&\quad \exists cp, cp', A. \text{var}(\text{path}, cp) * cp \equiv cp' / a \wedge T \\
&\quad \wedge \neg \text{noloop}(cp', T, T, \emptyset, \text{SYMLoop_MAX})
\end{aligned}$$

$$\begin{aligned}
& \{ \text{var}(r, -) * \text{var}(\text{target}, cp) * \text{pvar}(\text{link}, p/a_1) * \text{tree}(p/s \hat{a}_1) \} \\
& \quad r := \text{symlink}(\text{target}, \text{link}) \\
& \{ \text{var}(r, 0) * \text{var}(\text{target}, cp) * \text{pvar}(\text{link}, p/a_1) * \text{tree}(p/s a_1 @ cp) \} \\
& \quad \{ \text{var}(sl, -) * \text{pvar}(\text{path}, p/a_\pi) * \text{tree}(p/s a_\pi @ cp) \} \\
& \quad \quad sl := \text{readlink}(\text{path}) \\
& \quad \{ \text{var}(sl, cp) * \text{pvar}(\text{path}, p/a_\pi) * \text{tree}(p/s a_\pi @ cp) n \} \\
& \left\{ \begin{aligned} & \text{var}(r, -) * \text{pvar}(\text{existing}, p/a_\pi) \wedge sl \# (p/a_1) \leq \text{SYMLINK_MAX} * \text{pvar}(\text{new}, p'/b_1) \\ & * \text{tree}(p/s a_\pi @ \text{strippath}(p'')) * \text{tree}(p''/c_\pi : \iota) * \text{tree}(p'/\hat{b}_1) \end{aligned} \right\} \\
& \quad r := \text{link}(\text{existing}, \text{new}) \\
& \left\{ \begin{aligned} & \text{var}(r, 0) * \text{pvar}(\text{existing}, p/a_\pi) * \text{pvar}(\text{new}, p'/b_1) \\ & * \text{tree}(p/s a_\pi @ \text{strippath}(p'')) * \text{tree}(p''/c_\pi : \iota) \\ & * (\text{tree}(p'/b_1 : \iota) \vee (\text{tree}(p'/b_1 @ \text{strippath}(p''))) \end{aligned} \right\} \\
& \quad \{ \text{var}(r, -) * \text{pvar}(\text{path}, p/a_1) * \text{tree}(p/s a_1 @ cp) \} \\
& \quad r := \text{unlink}(\text{path}) \\
& \{ \text{var}(r, 0) * \text{pvar}(\text{path}, p/a_1) * \text{path} \Rightarrow_p P/A_1 * \text{tree}(p/s \hat{a}_1) \} \\
& \quad \left\{ \begin{aligned} & \text{var}(r, -) * \text{pvar}(\text{old}, p/a_1) * \text{pvar}(\text{new}, p'/b_1) \\ & * \text{tree}(p/s a_1 \# X) * \text{tree}(p'/s b_1 \# Y) \end{aligned} \right\} \\
& \quad r := \text{rename}(\text{old}, \text{new}) \\
& \quad \left\{ \begin{aligned} & \text{var}(r, 0) * \text{pvar}(\text{old}, p/a_1) * \text{pvar}(\text{new}, p'/b_1) \\ & * \text{tree}(p/s \hat{a}_1) * \text{tree}(p'/s b_1 \# X) \end{aligned} \right\} \\
& \quad \{ \text{var}(t, -) * \text{pvar}(\text{path}, p) * \text{tree}(p/s : \iota) \} \\
& \quad t := \text{stat}(\text{path}) \\
& \quad \{ \text{var}(t, F) * \text{pvar}(\text{path}, p) * \text{tree}(p/s : \iota) \} \\
& \quad \{ \text{var}(t, -) * \text{pvar}(\text{path}, p/a_\pi) * \text{tree}(p/s a_\pi [\emptyset]) \} \\
& \quad t := \text{lstat}(\text{path}) \\
& \quad \{ \text{var}(t, D) * \text{pvar}(\text{path}, p/a_\pi) * \text{tree}(p/s a_\pi [\emptyset]) \} \\
& \quad \{ \text{var}(t, -) * \text{pvar}(\text{path}, p/a_\pi) * \text{tree}(p/s a_\pi : \iota) \} \\
& \quad t := \text{lstat}(\text{path}) \\
& \quad \{ \text{var}(t, F) * \text{pvar}(\text{path}, p/a_\pi) * \text{tree}(p/s a_\pi : \iota) \} \\
& \quad \{ \text{var}(t, -) * \text{pvar}(\text{path}, p/a_\pi) * \text{tree}(p/s a_\pi @ cp) \} \\
& \quad t := \text{lstat}(\text{path}) \\
& \quad \{ \text{var}(t, S) * \text{pvar}(\text{path}, p/a_\pi) * \text{tree}(p/s a_\pi @ cp) \} \\
& \quad \{ \text{var}(fd, -) * \text{pvar}(\text{path}, p) * \text{var}(\text{flags}, 0) * \text{tree}(p/s : \iota) \} \\
& \quad fd := \text{open}(\text{path}, \text{flags}) \\
& \quad \{ \text{fd}(fd, \iota, 0) * \text{pvar}(\text{path}, p) * \text{var}(\text{flags}, 0) * \text{tree}(p/s : \iota) \} \\
& \quad \left\{ \begin{aligned} & \text{var}(fd, -) * \text{pvar}(\text{path}, p) * \text{var}(\text{flags}, \text{O_TRUNC}) \\ & * \text{tree}(p/s : \iota) * \text{fh}(\iota, 0, \vec{x}_n) * \text{fh}(\iota, n, \vartheta) \end{aligned} \right\} \\
& \quad fd := \text{open}(\text{path}, \text{flags}) \\
& \quad \{ \text{fd}(fd, \iota, 0) * \text{pvar}(\text{path}, p) * \text{var}(\text{flags}, \text{O_TRUNC}) * \text{tree}(p/s : \iota) * \text{fh}(\iota, 0, \vartheta) \}
\end{aligned}$$

Figure 20: Symbolic link extended small axioms.