DOI 10.1007/s00165-010-0171-2

Formal Aspects of Computing (2012) 24: 27–44

Formal Aspects of Computing

Formalizing a hierarchical file system

Wim H. Hesselink[1] and Muhammad Ikram Lali[2]

[1] Department of Computing Science, University of Groningen, P.O. Box 407, 9700 AK Groningen, The Netherlands.
E-mail: w.h.hesselink@rug.nl
[2] Department of Computer Science, COMSATS Institute of Information Technology, Chak Shahzad Campus, Islamabad, Pakistan.
E-mail: ikramlali@gmail.com

Abstract. An abstract file system is defined here as a partial function from (absolute) paths to data. Such a file system determines the set of valid paths. It allows the file system to be read and written at a valid path, and it allows the system to be modified by the Unix operations for creation, removal, and moving of files and directories. We present abstract definitions (axioms) for these operations. This specification is refined towards a pointer implementation. The challenge is to have a natural abstraction function from the implementation to the specification, to define operations on the concrete store that behave exactly in the same way as the corresponding functions on the abstract store, and to prove these facts. To mitigate the problems attached to partial functions, we do this in two steps: first a refinement towards a pointer implementation with total functions, followed by one that allows partial functions. These two refinements are proved correct by means of a number of invariants. Indeed, the insights gained consist, on the one hand, of the invariants of the pointer implementation that are needed for the refinement functions, and on the other hand of the precise enabling conditions of the operations on the different levels of abstraction. Each of the three specification levels is enriched with a permission system for reading, writing, or executing, and the refinement relations between these permission systems are explored. Files and directories are distinguished from the outset, but this rarely affects our part of the specifications. All results have been verified with the proof assistant PVS, in particular, that the invariants are preserved by the operations, and that, where the invariants hold, the operations commute with the refinement functions.

Keywords: File system, Specification, Verification, Refinement, Permission system, Theorem proving

1. Introduction

What is a hierarchical file system? Although most of us seem to know the answer, it is difficult to find a definition, let alone a specification. In [AL96], e.g., we read: "Like most modern operating systems, UNIX organizes its file system as a hierarchy of directories" and "directories, which contain information about a set of files and are used to locate a file by its name." If this answers the question for the impatient, it does not yield a specification. Yet, a specification is needed when we want to verify the correctness of an implementation.

Correspondence and offprint requests to: W. H. Hesselink, E-mail: w.h.hesselink@rug.nl

As file systems are at the core of the operating system kernel, even a simple error can cause a crash of the system, possibly resulting in loss of stored data [YTEM06]. File system errors are among the most dangerous errors because they can cause loss of persistent data stored on the disk. The growing size and complexity of file systems indicates the need of verification of such systems for ensuring reliability. It is very difficult to ensure reliability by testing techniques.

Testing and simulation are traditional techniques to check that the software written is correct with respect to its functionality [HR04]. Many testing techniques are available which help in eliminating coding errors. However, very few defects in end products are due to coding errors. For example, in 197 critical faults, detected during the testing phase of the Voyager and Galileo spacecraft, just three of them were coding errors. About 50% of the faults were traced to requirements, 25% to design, and the rest due to other errors. This is a typical example of a prevalent problem that the majority of faults in software arise in requirements and design and very few occur due to coding. Furthermore, such techniques do not cover all possible behaviors of the system [Lut93].

Formal verification uses mathematical techniques to ensure the design conforms to the functional specification. It can be applied to designs describing many different levels of abstraction [Pec99]. It helps in eliminating errors in the design which can cause problems at later stages.

In this paper, we formalize the most rudimentary aspects of a hierarchical file system: only reading and writing files, deleting them, creating them, and moving them. We do this in a top-down fashion, starting with the point of view of a user who does not want to know anything of the implementation. This is refined into a version with directories that hold subdirectories.

When formalizing this, we encountered the following problem of partial functions. If one implements a data structure with pointers, the model of the implementation contains a partial function that assigns values to some of these pointers, while others are unused. A formal treatment of such a system has to be very careful about the well-definedness of expressions that contain partial functions. The proof assistant PVS is strictly typed and therefore enforces a rigid discipline for dealing with partial functions. Our problem was that this discipline added unwanted complexity to an already complicated problem. This is not a PVS problem: every other prover would give the same problem, though possibly at a different point. It is a logical problem, that corresponds to the implementation problems associated to unallocated pointers.

Our solution was to split the problem by treating the refinement in two steps. In the first refinement step, the partiality is ignored by forcing the functions to be total. In the second refinement step, we recognize the inherent partiality of our functions. From the conceptual point of view, this may seem superfluous. For implementations, however, it is crucial because this partiality corresponds to the potential occurrence of unallocated pointers.

The primary contribution is to formally define a file system at a high level of abstraction with its five operations of reading and writing files, and creating, deleting and moving files and directories, and to refine this specification in two steps to a system with file identifiers as pointers, and to mechanically verify the refinement relations.

This is an extended version of the paper [HL09a] written for the 14th BCS-FACS Refinement Workshop (REFINE 2009). In the present version we distinguish files and directories from the outset, while the workshop version did not distinguish these cases. This makes it possible to approximate the Unix permission system better, e.g., because deletion of a directory requires stronger permissions than deletion of a file.

In the present version, all modifications of the filestore are defined functionally with an additional precondition. If the precondition is not met, the operation fails and the filestore is not modified. The workshop version had the same functionality, but not this explicit separation. The separation makes the definitions more readable. It also enables us to distinguish parts of the precondition that are needed to preserve well-definedness (legitimacy) of the filestore from parts that serve to defend the store against unintended destructive operations.

In data refinement, one often uses the contract approach that allows arbitrary results when the precondition of an operation is not satisfied. This is not what we want for filesystems because the filesystem must be preserved unmodified when the user gives nonsensical commands. Of course, we expect that an actual implementation then also gives error messages.

Mechanical verification

We use the proof assistant PVS [OSRSC01] for our formalization and the verification of the refinement relations. The PVS proof script of our definitions, theorems, and proofs is available at [HL09b]. Our notation in this paper is partially based on the PVS syntax, but we also use concepts from Haskell and standard mathematical notations.

All assertions in this paper have been proved with PVS, usually prepared by a hand-written proof. PVS then uncovers missing details. One can say that we do the proofs and PVS verifies them. Usually, only trivial parts can be left to PVS to prove independently. Indeed we use PVS as a *proof assistant*, to help us learn as much as possible about the model that we investigate. It is not our aim to use it as a *theorem prover* and guide it subtly to find proofs independently.

We needed a proof assistant like PVS for *verification*, because the kind of assertions we wanted to establish are difficult to prove reliably by hand. We come back to this in Sect. 6. A proof assistant is also helpful in *finding* proofs by keeping track of the set of remaining proof obligations, which changes in every proof step, but it has itself no intelligence to suggest promising proof strategies.

1.1. Related work

The 15 year old grand challenge in software verification proposed by Hoare in [Hoa03] was refined by Joshi and Holzmann in [JH07] to a mini-challenge to build a small verifiable file system for flash memory. In [FWB08], Freitas et al. suggested a roadmap towards solving the grand challenge. The current status of the challenge is discussed in [WB07]. Earlier, in [MS84], Morgan and Sufrin proposed abstract specifications of some of the data structures in the UNIX file system.

Several researchers used refinement techniques with automated theorem provers to specify and verify the file systems. The POSIX file store using Z/Eves with refinements based on [MS84] is described in [FWF09, Fu06]. The paper [FWF09] provides a concrete implementation of an abstract specification by means of Java HashMaps, taken from JML annotations given in [BCC+03]. Wenzel [Wen01] analyses aspects of the Unix file system security with the proof assistant Isabelle/HOL. Geambasu et al. [GBM08] formally specify fault tolerant file systems with TLA+. In [KJ08], Kang and Jackson presented an elaborated work on flash memory formalism using Alloy. More recently in 2009, Schierl et al. [SSHR09] contemplated the formalization of flash memory with reference to UBIFS implementation for Linux. They verify pre and postconditions, but do not consider refinements. A formal model of the bottom-level interface to memory is developed in [BFW09].

Model checking techniques to verify the file system implementation have been used in [GLMS09, YTEM06, TP09]. Galloway et al. [GLMS09] verify the existing Linux Virtual File System (VFS) by extracting and validating a model from an available implementation of VFS. Yang et al. [YTEM06] build their own model checker "FiSC" to find serious file system errors. This paper shows that even the most popular file systems contain serious bugs which can cause damage to the stored data. Therefore, it is important to consider correctness proofs even of existing file system implementations. In this regard, a correctness proof of operations like reading and writing in a Unix based file system is presented in [AZKR04] using Athena, an interactive theorem-proving environment. More recently (December 2009), Taverne et al. [TP09], designed a simple robust file store and implemented it in the form of a Promela model. They used model checking to verify the correctness of their implementation and did exhaustive verification of power loss recovery.

In 2008, inspired by Hughes' specification [Hug89] of a visual file system in Z, Damchoom, Butler, and Abrial [DBA08] have modeled a tree structured file system in Event-B and Rodin. This paper gives one of the first specifications of a hierarchical file system in which the tree structure can be modified. It is close to our work. An important difference, however, is that it is more abstract in the sense that it ignores file names and paths, which are central concepts in our specifications. We use top down approach for a deep understanding of system functions.

When presenting the material of this paper, we learned that our Sect. 2 on the user's point of view has some rather striking similarities with the algebraic specification of the Unix file system in [BGM87].

1.2. Overview

A hierarchical file system associates a file or a directory to a path. The user can read files, write into files, create, delete and modify files. From the user's point of view, these are deterministic operations. Under well specified conditions, these operations fail, but then the file store remains unchanged. In Sect. 2, we describe this in an abstract specification based on "the user's point of view".

Hierarchical file systems are usually implemented and described by means of file identifiers that serve as pointers to files and directories. The technical difficulties of unallocated pointers correspond in the theory with the technical difficulties of partial functions. We deal with these difficulties by first constructing in Sect. 3 a file system with total functions and all pointers allocated.

Section 4 presents the second refinement step to a system with pointers modelled as partial functions. In this way, the difficulties are not removed, they only appear one by one. The main result we prove, is that the file system operations of the three levels as specified here correspond exactly. The proof uses the invariants of the levels. It is constructed with the proof assistant PVS.

In Sect. 5, we indicate how file permissions as used in Unix can be specified in our set-up. Here, we again traverse the three levels of abstraction, and we take care that the permissions on the different levels correspond exactly. Conclusions are drawn in Sect. 6.

2. The user's point of view

From the user's point of view, a file store associates a file or a directory to a path. The user can inspect files by reading, modify files by writing, inspect the structure of the store by listing the files in a directory, and modify the structure of the store by the commands *create*, *delete*, and *move*. In this section, we specify these six operations in an abstract way.

2.1. The store as a partial function from paths to data

The user accesses the store usually by means of a relative path from his working directory. For the specification, however, it is easier to eliminate the working directory and exclusively work with the absolute path. Henceforward, we can therefore omit the word *absolute*. A *path* is a finite sequence of names, and the type of paths is defined by

$$Path = \texttt{finite\_sequence}[Name].$$

Name is the unspecified type of names. We use square brackets ([ ]) for type parameters and type constructors, just as in PVS.

For a given store, the question whether there are data stored at path p depends on the path, the sequence of names. If so, the path is called *valid* for the store. Validity of the path should of course imply that all strict prefixes of the path give directories. We come back to this later.

A store thus determines the valid paths, and the associated data for each valid path. We thus define an *abstract store* as a partial function from *Path* to *Data*, according to the following type definition:

$$StoreA = [Path \to \texttt{lift}[Data]],$$

where we use the PVS definition $\texttt{lift}[X] = X \cup \{\bot\}$. The valid paths p for an abstract store x are given by

$$valid(p, x) = (x(p) \neq \bot).$$

We distinguish files and directories by means of a boolean function

$$isdir : [Data \to Bool].$$

We extend this function to $\texttt{lift}[Data]$ by defining $isdir(\bot) = false$. For uniformity of notation, we write $isdir(p, x) = isdir(x(p))$. Note that $isdir(p, x)$ implies $valid(p, x)$. The empty path ε holds the root of the file system and should therefore always hold a directory.

We use the operator $+\!\!+$ for concatenation of paths as finite sequences. This operator is associative, i.e., $(p +\!\!+ q) +\!\!+ r = p +\!\!+ (q +\!\!+ r)$, and has the empty path ε as two-sided unit, i.e., $\varepsilon +\!\!+ p = p = p +\!\!+ \varepsilon$. Path p is called a *prefix* of q, with notation $p \sqsubseteq q$, iff there is a path r with $p +\!\!+ r = q$. Relation $\sqsubseteq$ is an ordering of the set *Path*, i.e., it is reflexive and transitive, and $p \sqsubseteq q \sqsubseteq p$ implies $p = q$. Path p is called a *strict prefix* of q (notation $p \sqsubset q$) iff $p +\!\!+ r = q$ for some $r \neq \varepsilon$.

As announced, the empty path should hold a directory and a strict prefix of a valid path should also hold a directory. We therefore define a store x to be *legitimate* if

$$\mathit{isdir}(\varepsilon, x) \ \wedge \ (\forall\ p, q : p \sqsubset q \ \wedge \ \mathit{valid}(q, x) \ \Rightarrow \ \mathit{isdir}(p, x)).$$

Legitimacy is an invariant of the stores: we prove that every proposed modification of stores preserves legitimacy.

2.2. Reading and writing

Reading the data of a path p in store x is just asking for $x(p)$, which yields $\perp$ iff path p is not valid in x. Reading the data of a directory only gives the metadata associated.

The Unix function *ls* associates to a given store x and a valid path p the set of names n that occur in the directory of p. Name n occurs in the directory of p iff the concatenation $p +\!\!+ n$ (i.e., path p extended with name n) is a valid path. We need to distinguish an empty directory from a nonexistent one. If the path is not valid or not a directory, we define *ls* to yield $\perp$. We thus define:

$$\mathit{ls} : [\mathit{Path} \times \mathit{StoreA} \to \texttt{lift}[\mathbb{P}[\mathit{Name}]]],$$
$$\mathit{ls}(p, x) = (\mathit{isdir}(p, x)\ ?\ \{n \mid \mathit{valid}(p +\!\!+ n, x)\} :\ \perp).$$

Here and henceforth, we use a C-like syntax for conditional expressions (the PVS syntax is more verbose).

Writing a file means modifying the data according to some recipe, e.g., writing from a certain offset. Such a recipe can be regarded as a function from *Data* to *Data*, but it should not change a file into a directory or vice versa. We therefore define the type

$$\mathit{Modifier} = \{m : [\mathit{Data} \to \mathit{Data}] \mid \mathit{isdir} \circ m = \mathit{isdir}\}.$$

Writing with modifier m at path p in store x is only successful when p is valid. Otherwise nothing happens. For simplicity, we do not yet include error messages for failure. We therefore lift every modifier m to $\texttt{lift}[\mathit{Data}]$ by defining $m(\perp) = \perp$ and define writing by:

$$\mathit{write} : [\mathit{Path} \times \mathit{Modifier} \times \mathit{StoreA} \to \mathit{StoreA}],$$
$$\mathit{write}(p, m, x) = (x\ \textbf{with}\ [(p) := m(x(p))]),$$
or equivalently: $\mathit{write}(p, m, x)(q) = (q = p\ ?\ m(x(p)) : x(q))$.

Here we use the **with** notation of PVS for function modification, with a conditional expression as an alternative. If x is legitimate, then $\mathit{write}(p, m, x)$ is also legitimate.

We can also use modifiers when we want to specify that the reader does not read all of the file, but only part of the file. In this case, we just define

$$\mathit{read} : [\mathit{Path} \times \mathit{Modifier} \times \mathit{StoreA} \to \texttt{lift}[\mathit{Data}]],$$
$$\mathit{read}(p, m, x) = m(x(p)).$$

The modifier m is used here to select the part of the file the user is requesting. In specifications, the modifiers used for reading are simply applied after reading the entire file. We can therefore ignore these modifiers.

2.3. Modifying the structure of the store

As announced, we consider modifications of the store by functions *create*, *delete*, *move*. For each of these functions, say F, we define a precondition $\mathit{pre}.F$ and an operation F_0, and the operation is applied when the precondition is satisfied, while the store remains unchanged otherwise. In other words, the function F is defined by the conditional expression

$$F(\mathit{args}, x) = (\mathit{pre}.F(\mathit{args}, x)\ ?\ F_0(\mathit{args}, x) : x),$$

where *args* is the list of additional arguments of the function. In the present section, the preconditions are needed partly to guarantee that the operation preserves legitimacy of the store. Later, they will also be needed to guarantee that the operations are well-defined. The preconditions usually also contain conjuncts to defend the existing file system against unintended destructive operations. These conjuncts are called *defensive*.

In the setting of refinement theory, this use of preconditions is exceptional. In the more usual contract approach, the implementor may assume that the precondition always holds, and may therefore do anything when the precondition fails. We insist, however, that the store remains unmodified when the precondition fails. The reason for this is that the user should be allowed to give arbitrary commands, but should be defended against unintended operations. In practice, we assume that the store's command interpreter (the shell) will give error messages when the precondition is not satisfied. We have omitted these error messages because they would be distracting.

We specify a function *create* that makes a new entry with data d in the store for a given path p. It does so only when path p is not yet valid and has a valid parent directory. Otherwise, *create* has no effect. Here, for a nonempty path p, the parent path $parent(p)$ is defined as the unique maximal strict prefix of p, which satisfies $|\ parent(p)\ | = |\ p\ | - 1$, where $|\ p\ |$ stands for the length of p.

$$
\begin{aligned}
&create : [Path \times Data \times StoreA \to StoreA],\\
&pre.create(p, d, x) :\ isdir(parent(p), x)\ \wedge\ \neg\, valid(p, x),\\
&create_0(p, d, x) = (x\ \mathbf{with}\ [(p) := d]).
\end{aligned}
$$

If store x is legitimate, the store $create(p, d, x)$ is legitimate because $parent(p)$ holds a valid directory. The second conjunct of the precondition is defensive: it precludes the destruction of an existing entry in the store. We define function *parent* to be a total function with $parent(\varepsilon) = \varepsilon$. Therefore, if $p = \varepsilon$, the precondition is false: one cannot create at the root of the store (initialization is treated below).

Deletion of a path p from an abstract store x also deletes all descendant files and directories, because we want a legitimate store to remain legitimate. Deletion is thus specified by

$$
\begin{aligned}
&delete : [Path \times StoreA \to StoreA],\\
&pre.delete(p, x) :\ p \neq \varepsilon,\\
&delete_0(p, x)(r) = (p \sqsubseteq r\ ?\ \bot : x(r)).
\end{aligned}
$$

The root ε of the file system must not be deleted because legitimacy asks for a valid root.

Moving is more complicated. The move from p to q has the effect that the old directory q (if it was valid) is completely overwritten by p, whereas the old directory p disappears. Let store $y = move(p, q, x)$ be the result of the move. For a path r of the form $r = q +\!\!+ s$, we therefore have $y(r) = x(p +\!\!+ s)$. For $q \sqsubseteq r$, this implies $y(r) = x(p +\!\!+ drop(|\ q\ |, r))$ where $drop(k, r)$ is the suffix of r obtained by removing the first k elements. We thus obtain:

$$
\begin{aligned}
&move : [Path \times Path \times StoreA \to StoreA],\\
&pre.move(p, q, x) :\ p \not\sqsubseteq q\ \wedge\ isdir(parent(q), x)\ \wedge\ q \neq \varepsilon\ \wedge\ valid(p, x),\\
&move_0(p, q, x)(r) =\\
&\quad (\ q \sqsubseteq r\ ?\ x(p +\!\!+ drop(|\ q\ |, r))\\
&\quad :\ p \sqsubseteq r\ ?\ \bot\\
&\quad :\ x(r)\ ).
\end{aligned}
$$

The precondition forbids moving a directory p to a proper subdirectory q (in which case $p \sqsubset q$), because this would give an orphaned directory q, making the store illegitimate. The case $p = q$ is excluded because it is useless. The condition that the parent of q must be a directory is also imposed to ensure that the store remains legitimate. The other two conjuncts are defensive.

Because of the case distinctions in the definition of *move*, the proof that *move* preserves legitimacy is rather complicated. A key step in the proof is the observation that, if $q \sqsubseteq s$ and $r \sqsubseteq s$ and $q \not\sqsubseteq r$, then $r \sqsubseteq parent(q)$.

We finally specify an initial store with arbitrary data d and an empty directory:

$$
\begin{aligned}
&initstoreA : [Data \to StoreA],\\
&initstoreA(d)(p) = (p = \varepsilon\ ?\ d : \bot).
\end{aligned}
$$

To summarize the section, we proved with PVS:

Theorem 1 (a) *If $isdir(d)$ holds, then $initstoreA(d)$ is legitimate.*

(b) *If store x is legitimate, then $write(p, m, x)$, $create(p, d, x)$, $delete(p, x)$, and $move(p, q, x)$ are legitimate.*

Note that, when the precondition of the operation is not met in case (b), the assertion is trivial because the result equals x.

3. Refining the store

In this section, we develop a simplified pointer implementation of the file system as specified in Sect. 2. We then define the corresponding operations and prove refinement, i.e., construct a refinement function *abstract* from the implementation to the specification that commutes with the operations.

3.1. The data structure, reading and abstraction

The usual implementation of a file store is by means of the standard pointer implementation of a tree. We use a simple type *Fid* of file identifiers as the pointer type. The root of the tree is given by a constant file identifier $rootId \in Fid$. For now, we define a *directory* to be a total function that maps names to file identifiers. We use a constant $null \in Fid$ as a default file identifier for nonoccurring names. We postulate that $rootId \neq null$.

We thus allow nodes also for invalid paths. They always hold a potential directory *dir*, which may be empty, and they may have data. A *total store* is a total function from file identifiers to nodes.

$$\begin{aligned} &DirT = [Name \to Fid], \\ &NodeT = [\#\quad data : \mathtt{lift}[Data]\ ,\ dir : DirT\quad \#], \\ &StoreT = [Fid \to NodeT]. \end{aligned}$$

Here [# and #] are constructors for record types as used in PVS. The corresponding element constructors are (# and #), used below. For a node v, we write $v.data$ and $v.dir$ for its data and its potential directory.

The first question to consider is whether a node holds a file or a directory. This is determined by the Boolean function *isdir* on `lift`[*Data*] of the previous section. Now, in the above definition, for the sake of uniformity, the type *NodeT* has a field *dir* even when the *data* would prohibit this by $\neg\, isdir(data)$. In such cases, rather than having an undefined directory, we use an empty directory. As *DirT* consists of total functions, we define an element of it to be empty when it maps all names to *null*. We therefore define legitimacy of total stores as follows.

A store $x : StoreT$ is called *legitimate* if it has a directory at *rootId*, and at every node with a nonempty *dir* field:

$$\begin{aligned} &isdir(x(rootid).data) \\ &\wedge\ (\forall\ f \in Fid : isdir(x(f).data)\ \vee\ x(f).dir = (\lambda\ n : null)). \end{aligned}$$

A new node with data d and without children is declared by

$$nodeT(d) = (\#\ data := d\ ,\ dir := (\lambda\ n : null)\ \#).$$

The initial store is defined by

$$initstoreT(d) = (\lambda\ f :\ f = rootId\ ?\ nodeT(d) : nodeT(\bot)).$$

It is legitimate iff $isdir(d)$ holds.

Since a store x is supposed to be a total function, we postulate an invariant to ensure that no data are hidden in or beyond *null*, viz.

$$J0(x) : \qquad x(null) = nodeT(\bot).$$

The file identifier associated to a path in a given store is defined recursively. For this purpose, we define a function $last : [Path \to Name]$ such that, for every nonempty path p, we have

$$p = parent(p) +\!\!+ last(p).$$

The file identifier of a path is given by the recursive *lookup* function L defined by:

$$\begin{aligned} &L : [Path \times StoreT \to Fid], \\ &L(p, x) = (\ p = \varepsilon\ ?\ \ rootId :\ \ x(L(parent(p), x)).dir(last(p))\ ). \end{aligned}$$

PVS accepts this recursive definition because $|\ parent(p)\ |<|\ p\ |$ whenever $p \neq \varepsilon$.

We only want to find $data = \bot$ at the node of *null*. This is expressed in the invariant

$$J1(x) : \qquad \forall\ p : x(L(p, x)).data = \bot\ \Rightarrow\ L(p, x) = null.$$

Reading is defined by

$$read(p, x) = x(L(p, x)).data.$$

We use the abbreviation $isdir(p, x) = isdir(read(p, x))$. The contents of a directory are found by means of function ls defined by

$$ls(p, x) = (isdir(p, x)\ ?\ ls(x(L(p, x)).dir) : \bot), \text{ where}$$
$$ls(di) = \{n \in Name \mid di(n) \neq null\}.$$

We define the abstraction function from total stores to abstract stores by

$$abstract : [StoreT \to StoreA],$$
$$abstract(x)(p) = x(L(p, x)).data.$$

It is straightforward to prove that $abstract(initstoreT(d)) = initstoreA(d)$.

A path p should be valid for a total store x iff the lookup function gives a file identifier; that is $L(p, x) \neq \bot$. This corresponds to validity according to the abstract store because, using $J0(x)$ and $J1(x)$, one can easily prove

$$valid(p, abstract(x)) \quad \equiv \quad L(p, x) \neq null.$$

Using invariant $J0$, we prove that

$$L(p, x) = null \ \wedge\ p \sqsubseteq q \ \Rightarrow\ L(q, x) = null.$$

One can now prove that $abstract(x)$ is legitimate when the total store x itself is legitimate.

Using the invariants $J0$ and $J1$, it is also easy to prove the refinement theorem that $ls(p, abstract(x)) = ls(p, x)$.

The challenge is now to define implementation functions for *write*, *create*, *delete*, and *move* that behave exactly in the same way as the corresponding functions on $StoreA$, and to prove such facts.

3.2. Writing in the store

In this section, we treat the operation *write*.

For writing, we use the PVS conventions for modifying functional structures. We thus define:

$$pre.write(p, m, x) :\ L(p, x) \neq null$$
$$write_0(p, m, x) = (x\ \mathbf{with}\ [(L(p, x)).data := m(x(L(p, x)).data)]).$$

Here, the precondition serves to ensure that $write_0$ is well-defined.

Writing does not change L, because writing affects only field *data*, while L only uses field *dir*. In other words, we have the easy result that

$$L(q, write(p, m, x)) = L(q, x).$$

The specification of Sect. 2 implies that writing at a path p only affects path p. This implies that the total store must be a tree, in the sense that different valid paths have different file identifiers. This is postulated in the invariant:

$$J2(x) : \quad \forall\ p, q :\ L(p, x) = L(q, x) \neq null \ \Rightarrow\ p = q.$$

With PVS, we have proved:

Theorem 2 *Assume that* $x : StoreT$ *satisfies* $J0(x)$, $J1(x)$, *and* $J2(x)$. *Then we have* $abstract(write(p, m, x)) = write(p, m, abstract(x))$.

The proof is not difficult, but also not illuminating. We have therefore left it out.

One may note that the invariant $J2$ forbids hard links as are used in the Unix file system.

3.3. Removals from the store

In this subsection, we treat the operation *delete*.

Given $x : StoreT$, a path p can only be deleted from it if it is not the root. Deletion then amounts to removing its last name from its parent directory:

$$\begin{array}{l} pre.delete(p, x) : \ p \neq \varepsilon, \\ delete_0(p, x) = (x \ \textbf{with} \ [\,(pp).dir(last(p)) := null\,]) \\ \text{where} \ \ pp = L(parent(p), x). \end{array}$$

We postpone garbage collection to Sect. 3.6.

It turns out that the invariants obtained above are enough to prove:

Theorem 3 *Assume that* $x : StoreT$ *satisfies* $J0(x)$ *and* $J2(x)$. *Then we have* $abstract(delete(p, x)) = delete(p, abstract(x))$.

At this point the reader has several options: either to believe that we proved this theorem with PVS, or to verify that our proof script at [HL09b] contains this result, and that PVS has proved it, or to look at the following hand-written proof that largely follows the PVS proof but omits many details.

Proof. We first claim that

$$\begin{array}{l} L(q, delete(p, x)) = \\ \quad (p \neq \varepsilon \ \wedge \ p \sqsubseteq q \,?\, null : L(q, x)). \end{array} \tag{0}$$

This is proved by induction on the length of q, because L is defined recursively. The invariant $J2$ is needed because store x is modified at $pp.dir(last(p))$, and at several points we therefore need to ensure that the arguments we are interested in differ from this.

We verify the final step by observing for every path q:

$$\begin{array}{ll} & abstract(delete(p, x))(q) \\ = & \{ \text{ definition of } abstract\text{; write } y = delete(p, x) \ \} \\ & y(L(q, y)).data \\ = & \{ \ (0) \text{ and } J0 \text{ for } y \ \} \\ & (p \neq \varepsilon \ \wedge \ p \sqsubseteq q \,?\, \bot \ : \ y(L(q, x)).data) \\ = & \{ \ x \text{ and } y \text{ are equal on } data \ \} \\ & (p \neq \varepsilon \ \wedge \ p \sqsubseteq q \,?\, \bot \ : \ x(L(q, x)).data)) \\ = & \{ \text{ definitions of } delete \text{ and } abstract \ \} \\ & delete(p, abstract(x))(q). \end{array}$$

This completes the proof. □

3.4. Creating new entries

In this section, we treat the operation *create*.

In order to preserve *J2* when creating new entries in the store, we need an unbounded heap. We formally ensure this by postulating that the type *Fid* is infinite and that the stores we consider are all finite, according to the invariant

$$\begin{array}{ll} J3(x) : & \#range(x) < \infty, \text{ where} \\ & range(x) = \{null, rootId\} \cup \{f \in Fid \mid \exists \ g, n : f = x(g).dir(n)\}. \end{array}$$

This enables us to define a choice function $new : StoreT \to Fid$ with the property:

$$J3(x) \quad \Rightarrow \quad new(x) \notin range(x). \tag{1}$$

Function *create* at this level of abstraction is defined by

$$\begin{array}{l} pre.create(p, d, x) : \ isdir(parent(p), x) \ \wedge \ L(p, x) = null, \\ create_0(p, d, x) = (x \ \textbf{with} \ [(pp).dir(last(p)) := ln \,, \ (ln) := nodeT(d)]) \\ \text{where} \ \ pp = L(parent(p), x) \text{ and } ln = new(x). \end{array}$$

Function *create* satisfies the refinement theorem:

Theorem 4 *Assume that* x : *StoreT satisfies* $J0(x) \wedge J2(x) \wedge J3(x)$. *Then we have* $abstract(create(p, d, x)) = create(p, d, abstract(x))$.

Proof. One first proves that the failure conditions of both versions of *create* are equivalent, because $abstract(x)(q) = \bot$ if and only if $L(q, x) = null$. Now assume both versions modify the store. We then prove, by induction on the length of q, that

$$\begin{aligned} &L(q, create(p, d, x)) = \\ &\quad (\ q = p \neq \varepsilon \ \wedge\ L(parent(p), x) \neq null = L(p, x)\ ?\ new(x) \\ &\quad :\ L(q, x)\ ). \end{aligned} \tag{2}$$

We verify the final step by observing for every path q:

$$\begin{aligned} & abstract(create(p, d, x))(q) \\ = \ & \{ \text{ definition of } abstract\text{; write } y = create(p, d, x) \ \} \\ & y(L(q, y)).data \\ = \ & \{ \ (2) \ \} \\ & (\ q = p \neq \varepsilon \ \wedge\ L(parent(p), x) \neq null = L(p, x)\ ?\ y(new(x)).data \\ & :\ y(L(q, x)).data) \\ = \ & \{ \text{ definition } y \text{ and } new\text{; } L(q, x) \neq new(x) \ \} \\ & (\ q = p \neq \varepsilon \ \wedge\ L(parent(p), x) \neq null = L(p, x)\ ?\ d \\ & :\ x(L(q, x)).data)) \\ = \ & \{ \text{ write } x' = abstract(x)\text{; definition of } abstract \ \} \\ & (\ q = p \neq \varepsilon \ \wedge\ x'(parent(p)) \neq \bot = x'(p)\ ?\ d \ :\ x'(q)) \\ = \ & \{ \text{ abstract definition of } create \ \} \\ & create(p, d, x')(q). \end{aligned}$$

This completes the proof. □

3.5. Moving files and directories

In this subsection, we treat the operation *move*.

Function *move* at this level is defined by:

$$\begin{aligned} &pre.move(p, q, x) :\ p \not\sqsubseteq q \ \wedge\ isdir(parent(q), x) \ \wedge\ q \neq \varepsilon \ \wedge\ L(p, x) \neq null, \\ &move_0(p, q, x) = (x \textbf{ with } [(qq).dir(last(q)) := L(p, x), \\ &\qquad\qquad\qquad\qquad (pp).dir(last(p)) := null\ ]\ ) \\ &\text{where } qq = L(parent(q), x) \text{ and } pp = L(parent(p), x) \end{aligned}$$

Note that $J2(x)$ implies that the file identifiers pp and qq are equal if and only if p and q have the same parent. If so, then $p \not\sqsubseteq q$ implies that $last(p)$ and $last(q)$ differ. The refinement theorem for *move* is:

Theorem 5 *Assume that* x : *StoreT satisfies* $J0(x) \wedge J1(x) \wedge J2(x)$. *Then we have* $abstract(move(p, q, x)) = move(p, q, abstract(x))$.

We have proved this with PVS in [HL09b]. The structure of the proof is the same as for *delete* and *create*. Due to the many case distinctions, it is cumbersome. We omit it because it is not illuminating.

3.6. Garbage collection

Unreachable nodes in the tree are useless. Garbage collection amounts to the removal of useless nodes. In the present context this is impossible because every store x is a total function: every file identifier in type *Fid* maps to a node. The best we can do is minimize the unreachable nodes. This is done as follows.

The set of reachable file identifiers f is defined by

$$reach(x) = \{f \mid \exists\ p : L(p, x) = f\}.$$

As unreachable file identifiers are never inspected, we define *garbage collection* by

$$gc : [StoreT \to StoreT],$$
$$gc(x)(f) = (f \in reach(x)\,?\; x(f) : nodeT(\bot)).$$

By a straightforward induction on the length of p, one proves that $L(p, gc(x)) = L(p, x)$ for all paths p. Having done this, one can easily prove that $abstract(gc(x)) = abstract(x)$. In words, garbage collection does not influence the meaning of the store.

3.7. Proofs of the invariants

It is straightforward to prove with PVS that the operations *write*, *delete*, *create*, *move*, and *gc* preserve legitimacy of the store. It is also easy to prove that they preserve the invariant $J0$, i.e., $J0(x)$ implies $J0(write(p, m, x))$ for all $x : StoreT$, and similarly for the other functions. The same is done for the invariant $J1$. Preservation of $J3$ under these five operations follows from the fact that they add at most one element (in the case of *create*) to the range of the store.

The invariant $J2$ uses function L, which is defined recursively. We therefore define two simpler invariants, which express that the file tree has no cycles and that all occurring file identifiers $\neq null$ are different:

$$J2a(x): \quad \forall\ f, n : x(f).dir(n) \neq rootId,$$
$$J2b(x): \quad \forall\ f, g, m, n : x(f).dir(m) = x(g).dir(n) \neq null\ \Rightarrow\ f = g\ \wedge\ m = n.$$

Here, f and g range over *Fid* and m and n range over *Name*. By induction on the lengths of the paths, one proves that these two invariants, together with $J0$, imply $J2$. It is fairly easy to prove that *write*, *delete*, *move*, and *gc* preserve the invariants $J2a$ and $J2b$. For *create*, we use $J3$ and formula (1).

Finally, it is straightforward to prove, with PVS or by hand, that $initstoreT(d)$ satisfies the invariants $J0$, $J1$, $J2a$, $J2b$, and $J3$.

4. Implementing the store

We now replace the total functions of the previous section by "finite maps", i.e., partial functions with a finite domain. We thus use the types declared in:

$$DirI = [Name \to \texttt{lift}[Fid]],$$
$$NodeI = [\#\quad data : Data\ ,\ dir : DirI\quad \#],$$
$$StoreI = [Fid \to \texttt{lift}[Node]].$$

Working with partial functions in a theorem prover like PVS gives technical difficulties that, from a conceptual point of view, seem inessential and distracting. In the implementation, however, these difficulties correspond to the usual problems with unallocated pointers. It is therefore important to get it correct at the theoretical level.

In our presentation here, we make one simplification of the PVS code. If X is a type, the PVS type $\texttt{lift}[X]$ represents $X \cup \{\bot\}$, but X is not a subset of $\texttt{lift}[X]$. Instead, there is an injection $\texttt{up} : [X \to \texttt{lift}[X]]$ and an inverse coercion $\texttt{down} : [X' \to X]$ where $X' \subseteq \texttt{lift}[X]$ is the image of up. In the presentation below, we suppress the functions up and down, and regard X and X' as identical.

We construct a refinement function *refine* from the present system to the one of the previous section in:

$$refine : [StoreI \to StoreT],$$
$$refine(x)(f) =$$
$$\quad (x(f) = \bot\,?\; nodeT(\bot)$$
$$\quad : (\#\ \ data := x(f).data\ ,\ dir := \psi \circ (x(f).dir)\ \#)\ )$$
$$\text{where } \psi(g) = (g = \bot\,?\; null : g).$$

An implemented store x is legitimate iff *rootId* holds a directory and every node with a nonempty field *dir* is a directory:

$$x(rootId) \neq \bot\ \wedge\ isdir(x(rootId).data)$$
$$\wedge\ (\forall\ f :\ x(f) = \bot\ \vee\ isdir(x(f).data)\ \vee\ x(f).dir = (\lambda\ n : \bot)).$$

It is easy to verify that, when $x : StoreI$ is legitimate, $refine(x)$ is legitimate in $StoreT$.

4.1. Reading and writing the store

The file identifier *null* is no longer needed in the implementation, but we allow and use it as an alias for $\perp$. We therefore define for $x : \mathit{StoreI}$ the invariant:

$K0(x):\quad x(\mathit{null}) = \perp.$

On the other hand, we want that all other file identifiers used in the store hold genuine nodes, as expressed in the invariant:

$K1(x):\quad \forall\ f \in \mathit{range}(x)\ \Rightarrow\ f = \mathit{null}\ \vee\ x(f) \neq \perp$, where
$\mathit{range}(x) = \{\mathit{null}, \mathit{rootId}\} \cup \{f \in \mathit{Fid} \mid \exists\ g, n : f = x(g).\mathit{dir}(n)\},$

where, by convention, $x(g).\mathit{dir}(n) \notin \mathit{Fid}$ when $x(g) = \perp$ or $x(g).\mathit{dir}(n) = \perp$.

At this refinement level, we need to define the *lookup* function L by

$$\begin{aligned} &L : [\mathit{StoreI} \times \mathit{Path} \to \mathit{Fid}],\\ &L(p, x) = (\ p = \varepsilon\ ?\ \mathit{rootId}\\ &\qquad\quad :\ x(L(\mathit{parent}(p), x)) = \perp\ \vee\ x(L(\mathit{parent}(p), x)).\mathit{dir}(\mathit{last}(p)) = \perp\ ?\ \mathit{null}\\ &\qquad\quad :\ x(L(\mathit{parent}(p), x)).\mathit{dir}(\mathit{last}(p))\ ). \end{aligned}$$

Using a straightforward induction on the length of path p, we now use PVS to prove

$$L(p, x) = L(p, \mathit{refine}(x)). \tag{3}$$

We do not want to present the calculational work needed for the proof. The admittedly ugly definition of the lookup function illustrates the problem of the partial functions mentioned in the introduction, which we have solved by doing a two-step refinement. It is thus the result of our investigation, and it could be helpfull when someone were to verify an actual pointer implementation.

The invariants $K0(x)$ and $K1(x)$ imply the rule:

$K01(x):\quad L(p, x) = \mathit{null}\quad \equiv\quad x(L(p, x)) = \perp.$

At this level, reading store $x : \mathit{StoreI}$ at path p is defined by

$$\mathit{read}(p, x) = (x(L(p, x)) = \perp\ ?\ \perp\ :\ x(L(p, x)).\mathit{data}).$$

A practical implementation would use the test $L(p, x) = \mathit{null}$ rather than the equivalent $x(L(p, x)) = \perp$. Doing this in PVS, however, would raise the objection that $x(L(p, x)).\mathit{data}$ is defined only if $x(L(p, x)) \neq \perp$. In other words, the function *read* would only be defined on the stores where $K01$ holds. Although we shall prove that $K01$ holds for all reachable stores, we prefer to define *read* as a total function in PVS and therefore use the definition above. The same argument applies to several of the definitions below.

Formula (3) enables us to prove that $K01(x)$ implies $\mathit{read}(p, \mathit{refine}(x)) = \mathit{read}(p, x)$. In other words, reading in the implementation corresponds to reading in the intermediate specification, as required.

We define $\mathit{isdir}(p, x) = \mathit{isdir}(\mathit{read}(p, x))$, just as in the previous section. The function *ls* is now defined by

$\mathit{ls}(p, x) = (\mathit{isdir}(p, x)\ ?\ \mathit{ls}(x(L(p, x)).\mathit{dir} : \perp)$, where
$\mathit{ls}(\mathit{di}) = \{n \in \mathit{Name} \mid \mathit{di}(n) \neq \perp\ \wedge\ \mathit{di}(n) \neq \mathit{null}\}.$

Using the invariant $K0$, it is easy to prove the refinement theorem that $\mathit{ls}(p, \mathit{refine}(x)) = \mathit{ls}(p, x)$. Writing of store x is defined by

$\mathit{pre.write}(p, m, x) :\ x(L(p, x)) \neq \perp,$
$\mathit{write}_0(p, m, x) = (x\ \mathbf{with}\ [(L(p, x)).\mathit{data} := m(x(L(p, x)).\mathit{data})]\ ).$

Using $K01(x)$, we proved with PVS that $\mathit{refine}(\mathit{write}(p, m, x)) = \mathit{write}(p, m, \mathit{refine}(x))$.

4.2. Tree modification

Analogously to the definition in Sect. 3.3, here removal is defined by

$$\begin{aligned}&\mathit{pre.delete}(p, x): \ p \neq \varepsilon \ \wedge \ L(p, x) \neq \mathit{null},\\ &\mathit{delete}_0(p, x) = (x \ \textbf{with} \ [\,(\mathit{pp}).\mathit{dir}(\mathit{last}(p)) := \bot\,])\\ &\text{where} \ \ \mathit{pp} = L(\mathit{parent}(p), x).\end{aligned}$$

Note that $L(p, x) \neq \mathit{null}$ implies that $x(L(\mathit{parent}(p), x)) \neq \bot$. Therefore this node indeed has a directory that can be modified. The equality $\mathit{refine}(\mathit{delete}(p, x)) = \mathit{delete}(p, \mathit{refine}(x))$ is proved with the invariant $K01(x)$.

For making a directory, we again need finiteness of the store as expressed in the invariant

$$K2(x): \qquad \#\mathit{range}(x) < \infty.$$

We can therefore define a function $\mathit{new} : [\mathit{Store} \to \mathit{Fid}]$ that satisfies $\mathit{new}(x) \notin \mathit{range}(x)$ for every x with $K2(x)$. We need a different node constructor (compare Sect. 3):

$$\mathit{nodeI}(d) = (\#\ \mathit{data} := d\ ,\ \ \mathit{dir} := (\lambda\ \ n : \bot)\ \#).$$

Analogously to Sect. 3.4, a new node is created by

$$\begin{aligned}&\mathit{pre.create}(p, d, x): \ \mathit{isdir}(\mathit{parent}(p), x) \ \wedge \ L(p, x) = \mathit{null},\\ &\mathit{create}_0(p, d, x) = (x \ \textbf{with} \ [(\mathit{pp}).\mathit{dir}(\mathit{last}(p)) := \mathit{ln}, (\mathit{ln}) := \mathit{node}(d)])\\ &\text{where} \ \ \mathit{pp} = L(\mathit{parent}(p), x) \text{ and } \mathit{ln} = \mathit{new}(x).\end{aligned}$$

It is easy to prove that $\mathit{range}(\mathit{refine}(x)) = \mathit{range}(x)$. We also get $\mathit{new}(\mathit{refine}(x)) = \mathit{new}(x)$, because we can use the same choice function. Using $K01(x)$, one can then prove the equality $\mathit{refine}(\mathit{create}(p, d, x)) = \mathit{create}(p, d, \mathit{refine}(x))$.

Function *move* is defined almost as in Sect. 3.5:

$$\begin{aligned}&\mathit{pre.move}(p, q, x): \ p \not\sqsubseteq q \ \wedge \ \mathit{isdir}(\mathit{parent}(q), x) \ \wedge \ q \neq \varepsilon \ \wedge \ L(p, x) \neq \mathit{null},\\ &\mathit{move}_0(p, q, x) = (x \ \textbf{with} \ [(\mathit{qq}).\mathit{dir}(\mathit{last}(q)) := L(p, x),\\ &\qquad\qquad\qquad\qquad (\mathit{pp}).\mathit{dir}(\mathit{last}(p)) := \bot\,])\\ &\text{where } \mathit{qq} = L(\mathit{parent}(q), x) \text{ and } \mathit{pp} = L(\mathit{parent}(p), x).\end{aligned}$$

At this point, the identification of type *Node* with a subtype of `lift`[*Node*] simplifies the presentation. Working in PVS, we need to make a case distinction whether the file identifiers *pp* and *qq* are equal or differ. Nevertheless, we formally proved the equality $\mathit{refine}(\mathit{move}(p, q, x)) = \mathit{move}(p, q, \mathit{refine}(x))$, using the invariant $K01$.

The verification that the invariants $K0$, $K1$, and $K2$ are preserved by the operations *write*, *delete*, *create*, and *move* are straightforward. These invariants also hold for the initial store defined by

$$\mathit{initstoreI}(d) = (\lambda\ f : \ f = \mathit{rootId}\,?\ \mathit{nodeI}(d) : \bot).$$

Moreover, $\mathit{refine}(\mathit{initstoreI}(d)) = \mathit{initstoreT}(d)$.

It follows that the composition $\mathit{abs} = \mathit{abstract} \circ \mathit{refine}$ is a genuine refinement function $\mathit{StoreI} \to \mathit{StoreA}$.

4.3. Garbage and garbage collection

Garbage collection is more useful at this level than in Sect. 3.6. Again we define:

$$\begin{aligned}&\mathit{reach} : [\mathit{StoreI} \to \mathbb{P}[\mathit{Fid}]],\\ &\mathit{reach}(x) = \{f \mid \exists\ p : L(p, x) = f\}.\end{aligned}$$

Garbage collection now means removal of unreachable nodes:

$$\begin{aligned}&\mathit{gc} : [\mathit{StoreI} \to \mathit{StoreI}],\\ &\mathit{gc}(x)(f) = (f \in \mathit{reach}(x)\,?\ x(f) : \bot).\end{aligned}$$

As before, one first proves that $L(p, \mathit{gc}(x)) = L(p, x)$ for all paths p and $x : \mathit{StoreI}$. Then it is, indeed, straightforward to prove that function *gc* preserves the three invariants $K0$, $K1$, and $K2$.

It is easy to prove that $\mathit{refine}(\mathit{gc}(x)) = \mathit{gc}(\mathit{refine}(x))$. It follows that the composition $\mathit{abs} : [\mathit{StoreI} \to \mathit{StoreA}]$ satisfies $\mathit{abs}(\mathit{gc}(x)) = \mathit{abs}(x)$ for all $x : \mathit{StoreI}$.

5. File permissions at three levels

File system permissions form a core issue in every operating system. Not all users must be able to read and modify all data. We therefore overload the six file system functions by adding a user as a new first argument, where *User* is a new type, uninterpreted for now.

5.1. Permissions in the abstract system

We describe the file system permission model from the user's point of view at the abstract level. For the user, we have typical access types like reading, executing and writing, and the owner can control the permissions to these operations. Furthermore, there is the concept of a super user, who holds all access rights in the file system.

We assume that the permissions attached to a node are encoded in the data of the node by means of predicates:

$$px, pr, pw : \mathbb{P}[User \times Data],$$

where px stands for the permission to execute, pr to read, and pw to write. We do not go into details of how these permissions are represented in the data. Instead, we concentrate on the specification and verification that users can only access and modify according to the permissions granted. As the functions px, pr, pw depend on the user, they can also depend on the classification of the user as creator of the file or directory, as a member of the group, etc. We can therefore here ignore these issues. As we need to apply these predicates in stores at a given path, we overload them to

$$\begin{array}{l} px, pr, pw : \mathbb{P}[User \times Path \times StoreA], \\ px(u, p, x) \;=\; x(p) \neq \bot \;\wedge\; px(u, x(p)), \end{array}$$

and similarly for pr and pw.

In case of files, readable, executable and writable means that the contents of a file can be read, executed (if it is executable) and written. In case of directories, readable corresponds to the listing of the directory entries, and executable means that user is allowed to go into the directory, i.e., "change directory". Writable means the permission to create or remove entries in the given directory. Therefore, for reading and writing in a file or directory at some path, the user needs execution rights along the whole path in the file system [AL96, Sect. 2.8]. This implies that the effective permissions are slightly more complicated functions that depend on the user, the path, and the store. We thus define:

$$\begin{array}{l} pX, pR, pW : \mathbb{P}[User \times Path \times StoreA], \\ pX(u, p, x) \;=\; (\forall\; q :\; q \sqsubseteq p \;\Rightarrow\; px(u, q, x)), \\ pR(u, p, x) \;=\; pr(u, p, x) \;\wedge\; (p = \varepsilon \;\vee\; pX(u, parent(p), x)), \\ pW(u, p, x) \;=\; pw(u, p, x) \;\wedge\; (p = \varepsilon \;\vee\; pX(u, parent(p), x)). \end{array}$$

Here, by convention, $parent(\varepsilon) = \varepsilon$. In some Unix variants, write permission may imply or require read permission. This can be modelled by adapting the relations of pw and pr to the actual permission bits.

The user-adapted abstract versions of *ls* and *read* are simply:

$$\begin{array}{l} ls(u, p, x) = (pR(u, p, x)\;?\; ls(p, x) \;:\; \bot), \\ read(u, p, x) = (pR(u, p, x)\;?\; x(p) \;:\; \bot). \end{array}$$

For the functions *write*, *create*, *delete*, and *move*, the permission system just adds another precondition. In each case, we can take $F_0(u, as, x) = F(as, x)$ and it suffices to specify the additional precondition $pre.F(u, as, x)$.

For *write*, the additional precondition is write permission:

$$pre.write(u, p, m, x) :\; pW(u, p, x).$$

For creation, the user needs permission to execute and write the parent directory. We therefore define

$$\begin{array}{l} pY(u, p, x) \;=\; pX(u, p, x) \;\wedge\; pw(u, p, x), \\ pre.create(u, p, d, x) :\; pY(u, parent(p), x). \end{array}$$

For deletion we need write and execute permission of the parent directory. Deletion of a directory is only allowed when the directory is empty, and we need *ls* to verify this. We therefore define

$$\begin{aligned} &pre.delete(u, p, x) : \\ &\quad pY(u, parent(p), x) \ \wedge \ (\neg\, isdir(p, x) \ \vee \ ls(u, p, x) = \emptyset). \end{aligned}$$

Note that the user u needs read permission to obtain $ls(u, p, x) = \emptyset$. Otherwise function *ls* yields $\perp$, and $\perp \neq \emptyset$.

For *move*, we propose:

$$pre.move(u, p, q, x) : \ pY(u, parent(p), x) \ \wedge \ pW(u, parent(q), x).$$

5.2. Refinement of permissions

We now turn from the abstract stores of Sect. 2 to the total stores of Sect. 3. We extend the permission bit functions px, pr, pw to the type `lift`[*Data*] by defining

$$px(u, \perp) = pr(u, \perp) = pw(u, \perp) = false.$$

The *lookup* function L that gives the file identifier of a path is now modified to verify execution permissions along the path:

$$\begin{aligned} &L : [User \times Path \times StoreT \to Fid], \\ &L(u, p, x) = \\ &\quad (\ p = \varepsilon \ ?\ \ rootId \\ &\quad :\ px(u, xpp.data) \ ?\ xpp.dir(last(p)) \\ &\quad :\ null\ ) \\ &\text{where}\ \ xpp = x(L(u, parent(p), x)). \end{aligned}$$

This expresses that the user can only traverse a path p if he has rights to execute all strict ancestors of p. The perceived permissions now become

$$\begin{aligned} &pX(u, p, x) = px(u, x(L(u, p, x)).data), \\ &pR(u, p, x) = pr(u, x(L(u, p, x)).data), \\ &pW(u, p, x) = pw(u, x(L(u, p, x)).data), \\ &pY(u, p, x) = (pX(u, p, x) \ \wedge \ pW(u, p, x)). \end{aligned}$$

Under assumption of $J0(x)$ and $J1(x)$, we can prove

$$\begin{aligned} &L(u, p, x) = \\ &\quad (p = \varepsilon \ \vee \ pX(u, parent(p), abstract(x)) \ ?\ L(p, x) : null). \end{aligned}$$

The proof of this is complicated. It is used to prove the abstraction result

$$pX(u, p, abstract(x)) = pX(u, p, x),$$

and the analogous formulas for pR, pW, pY.

The user-adapted versions of *read* and *ls* are given by

$$\begin{aligned} &read(u, p, x) = (pR(u, p, x) \ ?\ read(p, x) : \perp), \\ &ls(u, p, x) = (pR(u, p, x) \ ?\ ls(p, x) : \perp). \end{aligned}$$

Now, it is easy to define user-adapted versions of *write*, *create*, *delete*, and *move* in such a way that they correspond exactly to the abstract versions defined in Sect. 5.1.

5.3. Implementation of permissions

We turn to the concrete stores of Sect. 4. For the permission system, we extend the *lookup* function L of Sect. 4 to verify the execution permissions along the path:

$$\begin{array}{l} L : [\mathit{User} \times \mathit{Path} \times \mathit{StoreI} \to \mathit{Fid}], \\ L(u, p, x) = (\ p = \varepsilon\ ?\ \mathit{rootId} \\ \qquad : \ x(L(u, \mathit{parent}(p), x)) = \bot \\ \qquad \vee \ \neg\, px(u, x(L(u, \mathit{parent}(p), x)).\mathit{data}) \\ \qquad \vee \ x(L(u, \mathit{parent}(p), x)).\mathit{dir}(\mathit{last}(p)) = \bot\ ?\ \mathit{null} \\ \qquad : \ x(L(u, \mathit{parent}(p), x)).\mathit{dir}(\mathit{last}(p))\ ). \end{array}$$

The correspondence with the definition of Sect. 5.2 is established by

$$L(u, p, \mathit{refine}(x)) = L(u, p, x) \ \text{ for all } x : \mathit{StoreI}.$$

The perceived user permissions at this level are

$$\begin{array}{l} pX(u, p, x) = (x(L(u, p, x)) \neq \bot \ \wedge \ px(u, x(L(u, p, x)).\mathit{data})), \\ pR(u, p, x) = (x(L(u, p, x)) \neq \bot \ \wedge \ pr(u, x(L(u, p, x)).\mathit{data})), \\ pW(u, p, x) = (x(L(u, p, x)) \neq \bot \ \wedge \ pw(u, x(L(u, p, x)).\mathit{data})), \\ pY(u, p, x) = (pX(u, p, x) \ \wedge \ pW(u, p, x)). \end{array}$$

We can again prove the abstraction result

$$pX(u, p, \mathit{refine}(x)) = pX(u, p, x),$$

and the analogous formulas for pR, pW, pY.

We now define the user-adapted versions of *read* and *ls* just as in Sect. 5.2. The user adapted versions of *write*, *create*, *delete*, *move* are defined with the preconditions specified in 5.1 as translated to the concrete store, with the actions as specified in Sect. 4, but with $L(p, x)$ replaced by $L(u, p, x)$ when access permission is needed.

For example, *create* is given by

$$\begin{array}{l} \mathit{pre.create}(u, p, d, x) : \\ \quad \mathit{isdir}(\mathit{parent}(p), x) \ \wedge \ L(p, x) = \mathit{null} \ \wedge \ pY(u, \mathit{parent}(p), x), \\ \mathit{create}_0(u, p, d, x) = \\ \quad (x \ \textbf{with}\ [(\mathit{pp}).\mathit{dir}(\mathit{last}(p)) := \mathit{ln}, (\mathit{ln}) := \mathit{node}(d)]) \\ \text{where } \ \mathit{pp} = L(u, \mathit{parent}(p), x) \text{ and } \mathit{ln} = \mathit{new}(x). \end{array}$$

Here $L(p, x)$ must not be replaced by $L(u, p, x)$, because this conjunct in the precondition serves to preclude destroying an occupied node.

6. Conclusion

In this work, we developed an abstract specification of the six main operations in the Unix file system. This specification is deterministic: it tells the effect of every command. We refined this to an implementation with file identifiers as pointers, in such a way that the operations on the concrete level correspond exactly with the abstract operations. In order to do this provably correct, we had to use a proof assistant, and we had to do it in two steps.

Initially, we tried to model file systems directly at the implementation level of Sect. 4. In order to evade or at least postpone the details of partial functions, we invented the more abstract level described in Sect. 3. The real breakthrough came when we saw that we had to begin by specifying a hierarchical file system from a user's point of view, as a partial function from absolute paths to data. The requirements for the other two levels then emerged naturally as proof obligations for the refinement functions. Having the three levels was also very helpful in the development of the permission system. It was a relatively minor exercise to rework the PVS proof of the workshop paper towards a proof for the present version, although several of the critical definitions had been changed.

We used the proof assistant PVS. Advantages of PVS are: it has a simple syntax, reasonably close to mathematics, and a simple type system, rich enough for the problems at hand. During the interactive construction of a proof, PVS builds a proof tree with the remaining proof obligations at the leaves. If a proof seems to get stuck, this proof tree greatly helps to see where the proof went wrong and to get back on track. It is also useful for the

analysis of a proof, e.g., to find superfluous conditions. We only used the proof commands provided directly by PVS, although the application of PVS-strategies presumably would have made things easier in the end.

It is likely that the proofs could equally well have been done with any other theorem prover with higher-order functions, like HOL, Isabelle or Coq, although these provers lack proof trees. We have the impression that the level of proof automation of these provers is comparable to PVS. The prover ACL2 has stronger proof automation, but our specifications cannot be elegantly encoded in ACL2 because it lacks function types.

Now that we have the results, the results of the Sects. 2 and 3 can be proved by hand. In principle, the same holds for Sects. 4 and 5, see [Lam93], but in practice, we would not be able to do it. It is not that the proofs are difficult, but that they are complicated and boring. A human prover is therefore not reliable enough. The proof assistant is even more important during the development of the theory, because, when it fails to prove something, it shows which conditions are lacking. An important advantage of using a proof assistant is that, once a proof has been constructed, it can be replayed under modified conditions and then adapted where needed. If proof trees are provided, these are very helpful at this point.

What really happens when one uses PVS (or any other prover) to prove such things, is that one sees details that one missed before. We present such details in the paper at a semantic level. The syntax of PVS is very close to this level, as it is to mathematics in general. The main point of day-to-day user experience is: when you fail to prove something with PVS, you are too close to the prover and too far away from the mathematics; first make a better handwritten proof.

The entire PVS model, available at [HL09b], consists of 221 lemmas. Some of the lemmas were discharged automatically by the PVS theorem prover and rest were proved by human guidance. It took approximately 3 person months to complete this work with moderate level of expertise in using PVS. The main mathematical problems came from the case distinctions, the function types, the partial functions, and the recursive definitions.

As for directions for future research, the model needs an extension with hard links. At the abstract level, the appropriate way to do this may be by means of a modifiable equivalence relation on valid paths, as a second component of the store. Function *write* should then modify all members of the equivalence class of the path. After this, several problem areas call for attention: concurrent access, disk lay-out, distribution, and fault tolerance.

References

[AL96] Abrahams PW, Larson BR (1996) Unix for the impatient. Addison-Wesley, Reading

[AZKR04] Arkoudas K, Zee K, Kuncak V, Rinard M (2004) Verifying a file system implementation. In: Sixth international conference on formal engineering methods (ICFEM04). LNCS, vol 3308, pp 8–12

[BCC+03] Burdy L, Cheon Y, Cok DR, Ernst MD, Kiniry JR, Leino GTKRM, Poll E (2003) An overview of jml tools and applications. Int J Softw Tools Technol Transf 7(3):73–89

[BFW09] Butterfield A, Freitas L, Woodcock J (2009) Mechanising a formal model of flash memory. Sci Comput Programm 74:219–237

[BGM87] Bidoit M, Gaudel M-C, Mauboussin A (1987) How to make algebraic specifications more understandable? In: Wirsing M, Bergstra JA (eds) Algebraic methods: theory, tools and applications. Lect. Notes in Computer Science, vol 394. pp 31–69

[DBA08] Damchoom K, Butler MJ, Abrial J-R (2008) Modelling and proof of a tree-structured file system in Event-B and Rodin. In: ICFEM, pp 25–44

[Fu06] Fu Z (2006) A refinement of the UNIX filing system using Z/Eves. Master's thesis, University of York, October 2006

[FWB08] Freitas L, Woodcock J, Butterfield A (2008) POSIX and the verification grand challenge: A roadmap. In: ICECCS '08: proceedings of the 13th IEEE international conference on engineering of complex computer systems. IEEE Computer Society, Washington, DC, pp 153–162

[FWF09] Freitas L, Woodcock J, Fu Z (2009) POSIX file store in Z/Eves: an experiment in the verified software repository. Sci Comput Program 74(4):238–257

[GBM08] Geambasu R, Birrell A, MacCormick J (2008) Experiences with formal specification of fault-tolerant file systems. In: IEEE international conference on dependable systems and networks with FTCS and DCC, pp 96–101, June 2008

[GLMS09] Galloway A, Luttgen G, Muhlberg JT, Siminiceanu R (2009) Model-checking the Linux virtual file system. In: VMCAI, pp 74–88

[HL09a] Hesselink WH, Lali MI (2009) Formalizing a hierarchical file system. Electron Notes Theor Comput Sci 59:67–85

[HL09b] Hesselink WH, Lali MI (2009) PVS proof script of "file system formalization". http://www.cs.rug.nl/~wim/mechver/fs/index.html

[Hoa03] Hoare CAR (2003) The verifying compiler: A grand challenge for computing research. J ACM 50:63–69

[HR04] Huth M, Ryan M (2004) Logic in Computer Science: Modelling and reasoning about systems, 2nd ed. Cambridge University Press, London

[Hug89] Hughes J (1989) Specifying a visual file system in Z. Technical report. Department of Computing Science, University of Glasgow, 3 p
[JH07] Joshi R, Holzmann GJ (2007) A Mini Challenge: build a verifiable filesystem. Formal Aspects Comput 19:4
[KJ08] Kang E, Jackson D (2008) Formal modeling and analysis of a flash filesystem in alloy. In: ABZ '08: proceedings of the 1st international conference on abstract state machines, B and Z. Springer, Berlin, pp 294–308
[Lam93] Lamport L (1993) How to write a proof. Am Math Mon 102:600–608
[Lut93] Lutz R (1993) Analyzing software requirements errors in safety-critical embedded systems. In: IEEE international symposium on requirements engineering. CA, pp 126–133, January 1993
[MS84] Morgan C, Sufrin B (1984) Specification of the UNIX filing system. IEEE Trans Softw Eng SE-10:128–142
[OSRSC01] Owre S, Shankar N, Rushby JM, Stringer-Calvert DWJ (2001) PVS version 2.4. System Guide, Prover Guide, PVS Language Reference. http://pvs.csl.sri.com
[Pec99] Pecheur C (1999) Advanced modelling and verification techniques applied to a cluster file system. In: Proceedings of the 14th IEEE international conference on automated software engineering. IEEE Computer Society, Washington, DC, USA, pp 119–126
[SSHR09] Schierl A, Schellhorn G, Haneberg D, Reif W (2009) Abstract specification of the UBIFS file system for flash memory. In: Cavalcanti A, Dams D (eds) FM. Lecture notes in computer science, vol 5850. Springer, Berlin, pp 190–206
[TP09] Taverne P, Pronk C (2009) RAFFS: Model checking a robust abstract flash file store. In: Breitman K, Cavalcanti A (eds) Formal methods and software engineering. 11th international conference on formal engineering methods. LNCS, vol 5885. ICFEM 2009, Springer, Berlin, pp 226–245, December 2009
[WB07] Woodcock J, Banach R (2007) The verification grand challenge. Comput Soc India Commun 661–668
[Wen01] Wenzel M (2001) Some aspects of Unix file-system security. Isabelle/Isar proof document. T.U. Munchen
[YTEM06] Yang J, Twohey P, Engler D, Musuvathi M (2006) Using model checking to find serious file system errors. ACM Trans Comput Syst 24(4):393–423

Received 8 February 2010
Revised 1 October 2010
Accepted 23 November 2010 by Eerke Boiten, John Derrick, Dong Jin Song and Steve Reeves
Published online 17 December 2010