A Formal Proof of a Unix Path Resolution Algorithm

Ran Chen, Martin Clochard, Claude Marché

A Formal Proof of a Unix Path Resolution Algorithm

Ran Chen, Martin Clochard, Claude Marché

RESEARCH REPORT
N° 8987
December 2016
Project-Team Toccata

ISSN 0249-6399 ISRN INRIA/RR--8987--FR+ENG

A Formal Proof of a Unix Path Resolution Algorithm*

Ran Chen[†‡], Martin Clochard[§†], Claude Marché[†§]

Project-Team Toccata

Research Report n° 8987 — December 2016 — 27 pages

Abstract: In the context of file systems like those of Unix, *path resolution* is the operation that given a character string denoting an access path, determines the target object (file, directory, etc.) designated by this path. This operation is not trivial because of the presence of *symbolic links*. Indeed, the presence of such links may induce infinite loops.
In this report we consider a path resolution algorithm that always terminate. We propose a formal specification of path resolution and we formally prove that our algorithm terminates on any input, and is correct and complete.

Key-words: Formal Specification, Deductive Verification, Program Verifier Why3, Unix file system, Path resolution

* Work partly supported by the Joint Laboratory ProofInUse (ANR-13-LAB3-0007, `http://www.spark-2014.org/proofinuse`) and by the CoLiS project (ANR-15-CE25-0001, `https://www.irif.fr/~treinen/colis/`) of the French national research organization

† Inria, Université Paris-Saclay, F-91893 Palaiseau
‡ Institute of Software, Chinese Academy of Science, Beijing, China
§ LRI (CNRS & Univ. Paris-Sud), Université Paris-Saclay, F-91405 Orsay

RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Preuve formelle d'un algorithme de résolution de chemins dans Unix

Résumé : Dans le contexte des systèmes de fichiers comme celui d'UNIX, la *résolution d'un chemin* est l'opération qui étant donnée une chaîne de caractère dénotant un chemin d'accès, détermine l'objet cible (fichier, répertoire, etc.) désigné par ce chemin. Cette opération n'est pas triviale à cause de la présence de *liens symboliques*. En effet, la présence de tels liens peut induire la présence de boucles infinies.

Dans ce rapport nous considérons un algorithme de résolution de chemin qui termine toujours. Nous proposons une spécification formelle de la résolution de chemins et nous prouvons formellement la terminaison de notre algorithme, sa correction et sa complétude.

Mots-clés : Spécification formelle, preuve de programmes, environnement de preuve Why3, Système de fichier Unix, résolution de chemin

Contents

The Portable Operating System Interface (POSIX) is an IEEE family of standards [5] that defines standard operating system interface and environment. Its goal is to provide software compatibility between variants of Unix and other operating systems.

In the part of the POSIX standard concerning file systems, path resolution, or more precisely *pathname resolution*[1] is the operation that given a pathname determines the target object (file, directory, etc.) it denotes, if any. A pathname is a character string that is made of a sequence of *filenames* separated by the special character `"/"`. A pathname is *absolute* if it starts with `"/"` and *relative* otherwise. A filename, also called *pathname component* in POSIX, is a non-empty sequence of characters, containing neither `"/"` nor the NUL character (of ASCII code 0). The filenames `"."` and `".."` have special meanings, respectively the current and the parent directory. When the given pathname is absolute, pathname resolution starts from the root directory, otherwise it starts from the *current* directory of the process that attempts resolution.

The process of pathname resolution is non trivial because of the presence of *symbolic links*. A symbolic link is some kind of entry allowed in directories whose contents is a pathname. Resolving a pathname containing symbolic links somehow amounts to "recursively" resolve the pathnames associated to the links. Figure 1 presents an excerpt of the real file system tree that appear in a computer with a typical Debian installation. Notice the relative symbolic link `/usr/bin/latex` that points to `pdftex`, that is `/usr/bin/pdftex` as an absolute path ; the absolute symbolic link `/usr/bin/emacs` that points to `/etc/alternatives/emacs` which is itself a symbolic link to `/usr/bin/emacs24-x`.

Notice that depending on the context, a pathname under consideration does not need to be resolved completely, e.g. when performing a `mkdir` command, only the prefix without the last component of the pathname must be resolved. In this report, we only consider the problem of complete resolution of a

[1] `http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_13`

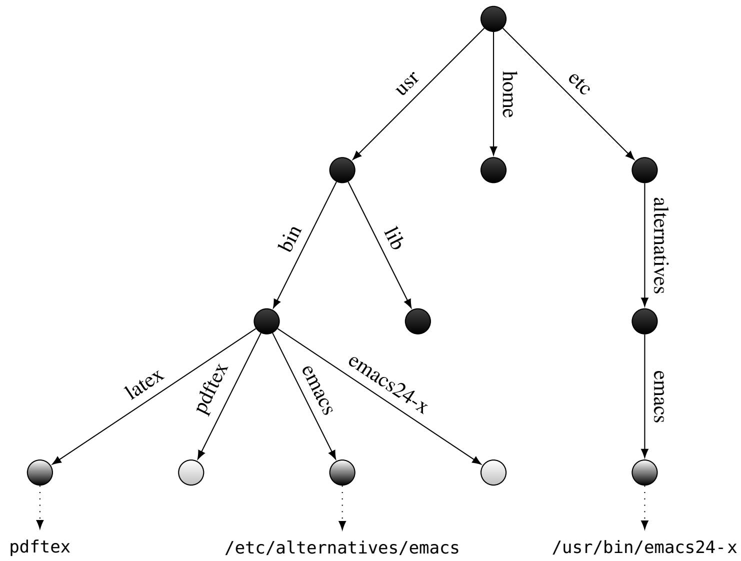

Figure 1: Excerpt of the file system tree typically found in a Debian installation. Black nodes denote directories, white nodes denote regular files, and gray nodes denote symbolic links.

pathname, thus resolution fails if the given path contains a non-existent component (e.g. when resolving `/usr/bin/absent` on the tree of figure 1). In practice, path resolution also fails if it attempts to access to a directory with insufficient permissions. Permissions do not pose any particular problem for a path resolution algorithm and we ignore them in this report.

The main difficulty comes from the presence of *symbolic links*: because of such links, the file system tree becomes some kind of a graph in which links may define cycles. A simple example would be a symbolic link that points to itself, or, on Figure 1, if the link `/etc/alternatives/emacs` was pointing to `/usr/bin/emacs` instead of `/usr/bin/emacs24-x`. In the presence of such cycles, the pathname resolution algorithm must be careful not to go into an infinite loop. This is the issue we address in this report. This issue is solved in practice by setting an arbitrary bound on the number of symbolic links that can be traversed during a given path resolution, in POSIX this number is required to be at least 32[2] . It means that the typical algorithm for path resolution implemented in a real OS is an incomplete one. The question of the existence of a terminating and complete algorithm is not open, such algorithms are known[3] but are not used in practice, probably because they require extra memory, and the limit of 32 is enough in practice.

The CoLiS project[4] aims at applying techniques from deductive program verification and analysis of tree transformations to the problem of analyzing shell scripts, in particular those that are used in software installation. During this project, we face the need for a formal specification of pathname resolution. We report here on the design of this formal specification and how it is used to formally prove a pathname resolution algorithm. The formalization is done using the Why3 program verifier [3]. In Section 1 we first present how we model file systems. We then present our resolution algorithm in Section 2. We describe our formal specification in Section 3 and show how the algorithm is proved in Section 4. Section 5 presents some related work and perspectives.

The annotated code for this work is available at URL `http://toccata.lri.fr/gallery/path_`

[2] `http://pubs.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap04.html#tag_04_11`

[3] see e.g. `http://unix.stackexchange.com/questions/99159/is-there-an-algorithm-to-decide-if-a-symlink-loops`

[4] `https://www.irif.fr/~treinen/colis/`

resolution.en.html.

1 Simplified Model of the file system

For our development, we formalize the file system in some abstract way. Regular files play no role in the path resolution algorithm so we just ignore them. The file system is a directed graph where the vertices are directories, called *dirnodes*. Edges of this graph are labeled by file names. An edge from a dirnode d_1 to a dirnode d_2 labeled by f means that f is a name that belongs to directory d_1 and that points to the sub-directory d_2.

1.1 Pathnames

A *pathname* (as defined by POSIX) is a sequence of file names, separated by slash characters, used to identify a file or a directory in the file system. In a pathname, "." and ".." have a special meaning, respectively to denote the current directory and the parent directory. We formalize them abstractly as follows.

```
type filename
type pathcomponent = Down filename | Up | Here
type path = list pathcomponent
```

The type for filenames is abstract, since for our purpose we don't need to know anything about it. A path is a list of path components, which can be either `Up` to denote `".."`, `Here` to denote `"."`, or (`Down` f) to denote a normal filename f.

1.2 The file system

The graph formed by the file system is formalized using these declarations:

```
type dirnode
constant root : dirnode
type child =
  | Absent
  | Dir dirnode
  | AbsLink path
  | RelLink path
function lookup dirnode filename : child
function parent dirnode : dirnode
  axiom parent_root: parent root = root
  axiom parent_non_root: forall d1 f d2. lookup d1 f = Dir d2 → parent d2 = d1
```

The constant `root` denotes the root directory of the file system. A path resolution may start from root or from current working directory. The function `lookup` is a total function which looks up a filename of a directory and returns the corresponding *child*. A *child* could be four cases, namely `Absent` denoting 'There is no such directory', (`Dir` d) denoting that successfully found a sub-directory d, (`AbsLink` p) denoting it is a symbolic link which stores an absolute path p, and (`RelLink` p) denoting it is a relative symbolic link which stores an relative path. A useful simple function we need is `parent` to get the parent directory of some dirnode. We axiomatize the function with two axioms.The first axiom indicates that the parent directory of root is root. The second axiom indicates that if we can lookup a filename f from directory d_1 to directory d_2, then d_1 is the parent directory of d_2. Thus, The parent function implies the graph is almost a tree in which there is no two different father directories of one directory node. Notice that for the moment we do not require the graph is finite.

Example 1 *Considering the structure of Figure 1, we have*

```
lookup root "usr" = Dir d1
lookup d1 "bin" = Dir d2
lookup d2 "emacs" = AbsLink "/etc/alternatives/emacs"
lookup d2 "latex" = RelLink "pdftex"
lookup d2 "foo" = Absent
```

2 Resolution Algorithms

We can give a naive path resolution algorithm now. We start to resolve a path p from some directory d. We match the path with several cases.

- If it's an empty path, then we stay in directory d.
- If the path starts with "`..`" , then we go to the parent directory of d and resolve the rest of the path.
- If it starts with "`.`", then we stay in the current directory d and resolve the rest of the path.
- If it starts with a normal file name, then we lookup the filename in directory d:
  - If it is absent, then path p resolve to nowhere. We raise an error then.
  - If it is a directory d'. then we resolve the remaining path from d'.
  - If it denotes an absolute link ps, then we recursively resolve the path ps from root to some directory d', and then resolve the remaining path of p from d'. The case of a relative link is similar, except we resolve ps from current directory d.

Here is the corresponding Why3 code for this naive algorithm.

```
exception Error

let rec aux_resolve (d:dirnode) (p:path) : dirnode =
  match p with
  | Nil → d
  | Cons Up pr → aux_resolve (parent d) pr
  | Cons Here pr → aux_resolve d pr
  | Cons (Down f) pr →
    match lookup d f with
    | Absent → raise Error
    | Dir d' → aux_resolve d' pr active
    | AbsLink ps →
      let d' = aux_resolve root ps in
      aux_resolve d' pr
    | RelLink ps →
      let d' = aux_resolve d ps in
      aux_resolve d' pr
    end
  end
```

We should notice that the naive algorithm above doesn't test a loop in the path. Because of the existing of the symbolic links, we may have a loop in the path which results the path cannot be resolved to anywhere and keep looping forever. Fig 2 presents some examples of the loops. $/a/e$ is a path with a loop in it since there is a symbolic link points to itself. $/c/f$ is also a path with a loop in it because there are two symbolic links in the path and they point to each other. From these two examples, we may suggest that we can detect a loop in the path by record the symbolic links we meet in it. But here we present another example $/d/d/d/d/c$ in which we meet the same symbolic link (root, d) several times. But this path can be resolved successfully.

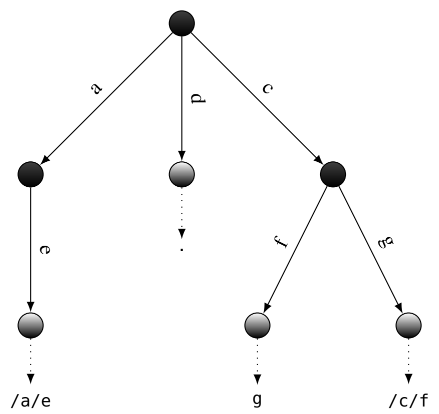

Figure 2: Toy examples of partial path resolution.

From the third example above, we can now present a good algorithm to resolve a path with an 'active' parameter to detect a loop in the path. 'active' is a set of pairs made of a directory node and a file name that resolves to a symbolic link at that directory. Every time we meet a symbolic link (d, f) in the path, we will detect the repetition of symbolic links when we resolve the link itself. Here is the Why3 code of the algorithm.

```
type lnk = (dirnode,filename)

exception Error

let rec aux_resolve (d: dirnode) (p:path) (active:set lnk) : dirnode
= match p with
  | Nil → d
  | Cons Up pr → let d' = parent d in aux_resolve d' pr active
  | Cons Here pr → aux_resolve d pr active
  | Cons (Down f) pr →
    match lookup d f with
    | Absent → raise Error
    | Dir d' → aux_resolve d' pr active
    | AbsLink ps →
      if mem (d,f) active
      then raise Error
      else begin
        let actadd = add (d,f) active in
        let d' = aux_resolve root ps actadd in
        aux_resolve d' pr active
      end
    | RelLink ps →
      if mem (d, f) active
      then raise Error
      else begin
        let actadd = add (d, f) active in
        let d' = aux_resolve d ps actadd in
        aux_resolve d' pr active
      end
    end
  end
```

The resolving function `aux_resolve` keeps looking up the path component recursively. Every time we meet a symbolic link in the path, we start to record the link. Then we will resolve the link itself, and keep record the links we meet during the resolving. Once we meet a same link again, we know that the link is looping and the path could not be resolved.

3 Formal Specification of Path Resolution

Our goal is now to express the informal property "from some directory d_1 we can resolve a path p to some other directory d_2". Because resolution does not always succeed, we cannot formalize this property as a logical function that, from d_1 and p, returns d_2. This is because in the logic of Why3, all logical functions are total. Instead, we formalize this property as a ternary predicate, that we denote as $d_1, p \rightsquigarrow d_2$.

We define this predicate inductively, that is we define it as the smallest predicate satisfying the rules below.

$$\frac{}{d, \varepsilon \rightsquigarrow d} \quad (\textsf{ResolveNil})$$

$$\frac{d_1(f) = \texttt{Dir}\, d_2 \quad d_2, p \rightsquigarrow d_3}{d_1, f/p \rightsquigarrow d_3} \quad (\textsf{ResolveDir})$$

$$\frac{d_1(f) = \texttt{AbsLink}\, ps \quad root, ps \rightsquigarrow d_2 \quad d_2, p \rightsquigarrow d_3}{d_1, f/p \rightsquigarrow d_3} \quad (\textsf{ResolveAbsLink})$$

$$\frac{d_1(f) = \texttt{RelLink}\, ps \quad d_1, ps \rightsquigarrow d_2 \quad d_2, p \rightsquigarrow d_3}{d_1, f/p \rightsquigarrow d_3} \quad (\textsf{ResolveRelLink})$$

$$\frac{\texttt{parent}\, d_1 = d_2 \quad d_2, p \rightsquigarrow d_3}{d_1, \texttt{../}p \rightsquigarrow d_3} \quad (\textsf{ResolveUp})$$

$$\frac{d_1, p \rightsquigarrow d_2}{d_1, \texttt{./}p \rightsquigarrow d_2} \quad (\textsf{ResolveHere})$$

The first rule means that resolving the empty path from some node d results to d itself. The second rule means that if from node d_1 the filename f denotes a directory node d_2, and if from d_2 the path p resolves to some node d_3, then we know from node d_1 we can resolve the path f/p to node d_3. The third rule indicates that if from some node d_1 we look up filename f then meet an absolute link which stores a path ps, we resolve this link from root to some node d_2 and from d_2 we can resolve path p to d_3, then from node d_1 we can resolve path f/p to node d_3. The fourth rule is similar to the third one, except when we look up the first filename we meet a relative link and we resolve this link from current directory which is d_1 itself. The fifth rule means if d_2 is the parent directory of some node d_1, and from d_2 we can resolve path p to some node d_3, then the path $../p$ can be resolved from d_1 to d_3. The last rule denotes if from some node d_1 we can resolve path p to node d_2 then we can resolve the path $./p$ from d_1 to d_2.

The predicate $d_1, p \rightsquigarrow d_2$ can be formalised in the Why3 logic using an inductive definition, as follows, that corresponds closely to the rules above.

```
inductive resolve_to dirnode path dirnode =
  | ResolveNil : forall d. resolve_to d Nil d
  | ResolveDir : forall d1 f d2 p d3.
    lookup d1 f = Dir d2 → resolve_to d2 p d3 → resolve_to d1 (Cons (Down f) p) d3
  | ResolveAbsLink : forall d1 f ps pr d2 d3.
    lookup d1 f = AbsLink ps → resolve_to root ps d2 → resolve_to d2 pr d3 →
    resolve_to d1 (Cons (Down f) pr) d3
  | ResolveRelLink : forall d1 f ps pr d2 d3.
    lookup d1 f = RelLink ps → resolve_to d1 ps d2 → resolve_to d2 pr d3 →
    resolve_to d1 (Cons (Down f) pr) d3
  | ResolveUp: forall d1 d2 d3 p.
    parent d1 = d2 → resolve_to d2 p d3 → resolve_to d1 (Cons Up p) d3
  | ResolveHere: forall d1 p d2.
```

```
resolve_to d1 p d2 → resolve_to d1 (Cons Here p) d2
```

Example 2 *Here are some examples of valid resolution. First we pretend that resolving the path* `/usr/bin` *from root results in* d_2*, that is* $root, \texttt{usr/bin} \rightsquigarrow d_2$*. The proof of this fact is*

$$\dfrac{root(\texttt{usr}) = d_1 \qquad \dfrac{d_1(\texttt{bin}) = d_2 \qquad \dfrac{}{d_2, \varepsilon \rightsquigarrow d_2}}{d_1, \texttt{bin} \rightsquigarrow d_2}}{root, \texttt{usr/bin} \rightsquigarrow d_2}$$

Now from d_2 *we pretend that resolving the path* `./latex` *to* d_3 *which is* $d_2, \texttt{./latex} \rightsquigarrow d_3$*. The proof is*

$$\dfrac{\dfrac{d_2(\texttt{latex}) = \texttt{RelLink}(\texttt{pdftex}) \qquad \dfrac{d_2(\texttt{pdftex}) = d_3 \qquad d_3, \varepsilon \rightsquigarrow d_3}{d_2, \texttt{pdftex} \rightsquigarrow d_3} \qquad \dfrac{}{d_3, \varepsilon \rightsquigarrow d_3}}{d_2, \texttt{latex} \rightsquigarrow d_3}}{d_2, \texttt{./latex} \rightsquigarrow d_3}$$

Suppose the parent directory of some directory node d_1 *is* `root`*, and we pretend that resolving the path* `../etc/alternatives/emacs` *from* d_1 *results in* d_4*. The proof of* $d_1, \texttt{../etc/alternatives/emacs} \rightsquigarrow d_4$ *is*

$$\dfrac{\texttt{parent}\ d_1 = \texttt{root} \qquad \dfrac{\texttt{root}(\texttt{etc}) = d_2 \qquad \dfrac{d_2(\texttt{alternatives}) = d_3 \qquad \Pi_1}{d_2, \texttt{alternatives/emacs} \rightsquigarrow d_4}}{\texttt{root}, \texttt{etc/alternatives/emacs} \rightsquigarrow d_4}}{d_1, \texttt{../etc/alternatives/emacs} \rightsquigarrow d_4}$$

where Π_1 *is the proof*

$$\dfrac{d_3(\texttt{emacs}) = \texttt{AbsLink}(\texttt{/usr/bin/emacs24-x}) \qquad \Pi_2 \qquad \dfrac{}{d_4, \varepsilon \rightsquigarrow d_4}}{d_3, \texttt{emacs} \rightsquigarrow d_4}$$

where Π_2 *is the proof*

$$\dfrac{\texttt{root}(\texttt{usr}) = d_5 \qquad \dfrac{d_5(\texttt{bin}) = d_6 \qquad \dfrac{d_6(\texttt{emacs24-x}) = d_4 \qquad \dfrac{}{d_4, \varepsilon \rightsquigarrow d_4}}{d_6, \texttt{emacs24-x} \rightsquigarrow d_4}}{d_5, \texttt{bin/emacs24-x} \rightsquigarrow d_4}}{\texttt{root}, \texttt{usr/bin/emacs24-x} \rightsquigarrow d_4}$$

3.1 Comparison with POSIX specification of resolution

If we compare our formal specification of path resolution to the informal one of POSIX[5], we can notice a slight divergence, lying on the way symbolic links must be handled: "If a symbolic link is encountered during pathname resolution, [...] the system shall prefix the remaining pathname, if any, with the contents of the symbolic link". On other words, in our rules ResolveAbsLink and ResolveRelLink, we should not have two premises but only one to resolve the concatenation of the link and the remaining pathname.

Our definition is indeed simpler because it does not use concatenation, and in particular it will make the proofs easier. To show that there is no difference with POSIX informal specification, we now define

[5] `http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_13`

another predicate closer to POSIX specification and prove the equivalence between this specification and the one above. The predicate $d_1, p \underset{POSIX}{\leadsto} d_2$ can be presented as an inductive predicate in Why3 code below.

```
inductive resolve_to_POSIX dirnode path dirnode =
  | ResolveNilPOSIX : forall d. resolve_to_POSIX d Nil d
  | ResolveDirPOSIX : forall d1 f d2 p d3.
    lookup d1 f = Dir d2 → resolve_to_POSIX d2 p d3 → resolve_to_POSIX d1 (Cons (Down f) p) d3
  | ResolveAbsLinkPOSIX : forall d1 f ps pr d2.
    lookup d1 f = AbsLink ps → resolve_to_POSIX root (ps ++ pr) d2 →
    resolve_to_POSIX d1 (Cons (Down f) pr) d2
  | ResolveRelLinkPOSIX : forall d1 f ps pr d2.
    lookup d1 f = RelLink ps → resolve_to_POSIX d1 (ps ++ pr) d2 →
    resolve_to_POSIX d1 (Cons (Down f) pr) d2
  | ResolveUpPOSIX: forall d1 d2 d3 p.
    parent d1 = d2 → resolve_to_POSIX d2 p d3 → resolve_to_POSIX d1 (Cons Up p) d3
  | ResolveHerePOSIX: forall d1 p d2.
    resolve_to_POSIX d1 p d2 → resolve_to_POSIX d1 (Cons Here p) d2
```

We want to prove that the predicate above is equivalent to the first one, as stated by the following theorem.

Theorem 3 *For any directory node d_1, d_2, and any path p,*

$$d_1, p \underset{POSIX}{\leadsto} d_2 \quad \text{if and only if} \quad d_1, p \leadsto d_2$$

To prove this theorem we have to state a few auxiliary lemmas. The first lemmas below are related to the first resolution predicate. We use the operator $++$ to denote the concatenation of paths.

Lemma 4 *for all dirnodes d_1, d_2, d_3 and paths p, q, if $d_1, p \leadsto d_2$ and $d_2, q \leadsto d_3$ then $d_1, p{+}{+}q \leadsto d_3$.*

The proof is done by induction on the hypothesis $d_1, p_1 \leadsto d_2$. Within Why3 such a lemma can be stated directly as follows.

```
lemma resolve_to_append : forall d1 d2 d3 p q.
   resolve_to d1 p d2 → resolve_to d2 q d3 →
   resolve_to d1 (p ++ q) d3
```

and the proof is done using the Why3 transformation `induction_pr` and the 6 resulting goals are proved by automatic provers. From now on we do not mention the Why3 code of our lemmas. See appendix for the detailed code and the detailed proofs.

We state the converse property using two lemmas. We denote by operator :: the list cons.

Lemma 5 *for all dirnodes d_1, d_3, any path component c and any path p, if $d1, c :: p \leadsto d_3$ then there exists d_2 such that $d_11, c :: Nil \leadsto d_2$ and $d_2, p_2 \leadsto d_3$.*

This lemma is also proved by induction on hypothesis $d1, c :: p \leadsto d_3$.

Lemma 6 *for all paths p_1, p_2 and all dirnodes d_1, d_3, if $d_1, p_1 {+}{+} p_2 \leadsto d_3$ then there exists d_2 such that $d_1, p_1 \leadsto d_2$ and $d_2, p_2 \leadsto d_3$.*

This lemma is proved by structural induction on p_1, using the previous lemma.

The next lemma concerns the POSIX variant of resolution predicate. It is similar to lemma 4.

Lemma 7 *For any directory node d_1, d_2 and d_3, and any path p_1, p_2, if $d_1, p_1 \underset{POSIX}{\leadsto} d_2$ and $d_2, p_2 \underset{POSIX}{\leadsto} d_3$ then $d_1, p_1 {+}{+} p_2 \underset{POSIX}{\leadsto} d_3$*

The proof is done by induction on hypothesis $d_1, p_1 \underset{POSIX}{\leadsto} d_2$.

Finally, the proof of Theorem 3 is done by considering each direction of the equivalence separately, and reasoning by induction on the predicate in hypothesis in both cases.

3.2 Resolution Indexed with Explicit Height

For the purpose the proving our resolution algorithm, we will need to explicitly refer to the height of the proof of some judgment $d_1, p \rightsquigarrow d_2$. For that purpose we introduce another predicate with an extra argument corresponding to that height. Moreover, we want to express that some path p can *not* be resolved, we express that by saying that it can be resolved with an *infinite* height. We denote $d_1, p \overset{\frown}{\rightsquigarrow} d_2, h$ to mean "resolving path p from node d_1 results in node d_2 with a proof of height h", and $d_1, p \overset{\frown}{\rightsquigarrow} \infty$ means we cannot resolve path p from node d_1. That predicate is also defined inductively with the rules below.

$$\frac{\forall d_1.\ \neg(d, p \rightsquigarrow d_1)}{d, p \overset{\frown}{\rightsquigarrow} \infty} \quad (\mathsf{ResolveHeightAbsent})$$

$$\frac{}{d, \varepsilon \overset{\frown}{\rightsquigarrow} d, 0} \quad (\mathsf{ResolveHeightNil})$$

$$\frac{d_1(f) = \mathtt{Dir}\ d_2 \quad d_2, p \overset{\frown}{\rightsquigarrow} d_3, h}{d_1, \mathtt{f/p} \overset{\frown}{\rightsquigarrow} d_3, h+1} \quad (\mathsf{ResolveHeightDir})$$

$$\frac{d_1(f) = \mathtt{AbsLink}\ ps \quad \mathtt{root}, ps \overset{\frown}{\rightsquigarrow} d_2, h_1 \quad d_2, p \overset{\frown}{\rightsquigarrow} d_3, h_2}{d_1, \mathtt{f/p} \overset{\frown}{\rightsquigarrow} d_3, \max(h_1, h_2) + 1} \quad (\mathsf{ResolveHeightAbsLink})$$

$$\frac{d_1(f) = \mathtt{RelLink}\ ps \quad d_1, ps \overset{\frown}{\rightsquigarrow} d_2, h_1 \quad d_2, p \overset{\frown}{\rightsquigarrow} d_3, h_2}{d_1, \mathtt{f/p} \overset{\frown}{\rightsquigarrow} d_3, \max(h_1, h_2) + 1} \quad (\mathsf{ResolveHeightRelLink})$$

$$\frac{\mathtt{parent}\ d_1 = d_2 \quad d_2, p \overset{\frown}{\rightsquigarrow} d_3, h}{d_1, \mathtt{../p} \overset{\frown}{\rightsquigarrow} d_3, (h+1)} \quad (\mathsf{ResolveHeightUp})$$

$$\frac{d_1, p \overset{\frown}{\rightsquigarrow} d_2, h}{d_1, \mathtt{./p} \overset{\frown}{\rightsquigarrow} d_2, (h+1)} \quad (\mathsf{ResolveHeightHere})$$

These rules can be written in Why3 as an inductive definition as follows. To separate the case of infinite height from the others, we use the option type, where `None` means infinity and `Some(d,h)` denotes a case of resolution in finite height.

```
inductive resolve_with_height dirnode path (option (dirnode,int)) =
  | ResolveHeightAbsent: forall d p. (forall d1. not resolve_to d p d1) →
                    resolve_with_height d p None
  | ResolveHeightNil : forall d. resolve_with_height d Nil (Some (d,0))
  | ResolveHeightDir : forall d1 f d2 p d3 h.
    lookup d1 f = Dir d2 → resolve_with_height d2 p (Some(d3,h)) →
    resolve_with_height d1 (Cons (Down f) p) (Some (d3,h + 1))
  | ResolveHeightAbsLink : forall d1 f ps pr d2 d3 h1 h2.
    lookup d1 f = AbsLink ps → resolve_with_height root ps (Some (d2,h1)) →
    resolve_with_height d2 pr (Some (d3,h2)) →
    resolve_with_height d1 (Cons (Down f) pr) (Some (d3, max h1 h2 + 1))
  | ResolveHeightRelLink : forall d1 f ps pr d2 d3 h1 h2.
    lookup d1 f = RelLink ps → resolve_with_height d1 ps (Some (d2,h1)) →
    resolve_with_height d2 pr (Some (d3,h2)) →
    resolve_with_height d1 (Cons (Down f) pr) (Some (d3,max h1 h2 + 1))
  | ResolveHeightUp : forall d1 d2 d3 p h.
    parent d1 = d2 → resolve_with_height d2 p (Some(d3, h)) →
    resolve_with_height d1 (Cons Up p) (Some (d3,h + 1))
  | ResolveHeightHere : forall d1 p d2 h.
    resolve_with_height d1 p (Some (d2,h)) →
    resolve_with_height d1 (Cons Here p) (Some (d2,h + 1))
```

3.3 Technical Lemmas

We need some technical lemmas to prove the algorithm. First, we need to state that proof height of a resolved path is greater or equal than 0.

Lemma 8 *For any directory node d_1 and d_2, and any path p, and any height h if $d_1, p \overline{\rightsquigarrow} d_2, h$ then $h \geq 0$*

The proof can be done by induction on the hypothesis $d_1, p \rightsquigarrow d_2$, and looking at all the 7 cases of rules applied to establish $d_1, p \rightsquigarrow d_2$.

We also need lemmas that relate the predicate `resolves_to` to the predicate with height `resolve_with_height`. if there is a resolution with explicit height $d_1, p \overline{\rightsquigarrow} d_2, h$ with any finite height h, then there is a resolution $d_1, p \rightsquigarrow d_2$.

Lemma 9 *For any directory nodes d_1 and d_2, any path p, and any height h if $d_1, p \overline{\rightsquigarrow} d_2, h$ then $d_1, p \rightsquigarrow d_2$.*

The proof can be done by induction on the hypothesis $d_1, p \rightsquigarrow d_2$, and looking at all the 7 cases of rules applied to establish $d_1, p \rightsquigarrow d_2$. That lemma can be turned into Why3 and as expected, the proof is done using the transformation `induction_pr`. The 7 resulting goals are proved automatically by Alt-Ergo.

A second lemma works the other way around: every time we resolve a path to some directory we can always resolve with some height.

Lemma 10 *For any directory node d_1 and d_2, and any path p, if $d_1, p \rightsquigarrow d_2$ then there exists h such that $d_1, p \overline{\rightsquigarrow} d_2, h$.*

The proof proceeds by induction also. The proof in Why3 has 6 cases and all of them are proved automatically by Alt-Ergo and CVC4.

3.4 Determinism of Resolution

An important property we need is the uniqueness of the result of resolution, if it exists. The following lemma states this property.

Lemma 11 *For any directory node d_1, d_2 and d_3, and any path p, if $d_1, p \rightsquigarrow d_2$ and $d_1, p \rightsquigarrow d_3$ then $d_2 = d_3$*

A similar lemma goes for the same as former one that same resolving with height will result to a same proof height.

Lemma 12 *For any directory node d, and any path p, and any $\overline{h_1}$ and $\overline{h_2}$, if $d, p \overline{\rightsquigarrow} \overline{h_1}$ and $d, p \overline{\rightsquigarrow} \overline{h_2}$ then $\overline{h_1} = \overline{h_2}$*

The last lemma is saying that we can always find a proof height (possibly infinite) for any path resolving from any directory.

Lemma 13 *For any directory node d, and any path p, exists $\overline{h}$ that $d, p \overline{\rightsquigarrow} \overline{h}$*

4 Proof of the Path Resolution Algorithm

4.1 Termination

Up to now, we did not specify that the filesystem has finitely many nodes. Potentially, resolution could not terminate even if there is no loop: imagine a link l_1 pointing to another link l_2 itself pointing to l_3 etc.

To prove termination, we thus need to add more constraints in our model of the filesystem, as follows.

```
constant alllinks : set lnk     (* finite set *)
axiom alllinks_in : forall d f ps.
  lookup d f = AbsLink ps ∨ lookup d f = RelLink ps → mem (d,f) alllinks
```

In other words, there exists some finite set $alllinks$ of pairs (dirnode,filename) such that all links in the file system belong to $alllinks$.

To achieve the proof of termination, we just need to state a *variant*, that is a quantity that decreases at each recursive call. A proper variant in this case is as follows: either the active set increases, or it remains unchanged and the length of the path p decreases. Instead of saying that the `active` set increases, we say that the complement set `alllinks` − `active` decreases. This requires to add a precondition stating that the `active` set is always a subset of `alllinks`. Such a precondition acts as an invariant maintained for all the recursive calls.

```
let rec aux_resolve (d: dirnode) (p:path) (active:set lnk) : dirnode
  requires { subset active alllinks }
  variant { cardinal alllinks - cardinal active, p }
= ...
```

From the variant above, given as a pair of an integer and a list, Why3 implicitly considers that the associated well-founded ordering is the lexicographic composition of the natural ordering on non-negative integers and the sub-list ordering on lists. The proof of termination is then obtained by automatic provers.

4.2 Correctness

The correctness of the algorithm is stated using the following post-condition.

```
let rec aux_resolve (d: dirnode) (p:path) (active:set lnk) : dirnode
  ensures { resolve_to d p result }
```

The proof works very easily, because the recursive calls of the algorithm recursively construct the needed premises to build the inductive proof of $d, p \rightsquigarrow$ `result`. Notice that for this proof it is important to use our first variant of the resolution predicate, and not the POSIX one.

4.3 Completeness

The completeness is stated using the following post-condition stated when the function raise the exception `Error`.

```
let rec aux_resolve (d: dirnode) (p:path) (active:set lnk) : dirnode
  raises { Error → forall d'. not resolve_to d p d' }
```

The hard part of the proof is to prove this completeness property. We need to add more invariants on the active set, again under the form of preconditions to be satisfied by the recursive calls.

The invariants on the `active` set are as follows: for all $(d_1, f) \in$ `active` and for any ps, if $d_1(f) =$ `AbsLink` ps then

- $\forall d_2.\ \texttt{root}, ps \rightsquigarrow d_2 \ \rightarrow\ \exists d'.\ d, p \rightsquigarrow d'$
- $\forall d_2, h_1, d', h.\ \texttt{root}, ps \stackrel{\rightsquigarrow}{\scriptstyle\rightsquigarrow} d_2, h_1 \ \rightarrow\ d, p \stackrel{\rightsquigarrow}{\scriptstyle\rightsquigarrow} d', h \ \rightarrow\ h \leq h_1$

Both assertions say something about an arbitrary pair (d_1, f) in the active set. If the filename f in dirnode d_1 denotes an absolute link to some path ps, then:

- The first assertion states that if ps is resolvable from root (to some d_2) then p is resolvable from d (to some d').
- The second assertion states that if ps is resolvable from root with some finite height h_1 and if p is resolvable from d with some finite height h then h is smaller than h_1.

In other words, these two assertions together mean that if you resolve any pair in the active set, then the input path p is resolvable also, with a proof that as a smaller height. This is the key property that allows us to prevent cycles in the proofs of path resolution, as we will see below. Still another way to express this is to say that the resolution of the current path p is a part of the resolution of all the paths that appear in the active set.

We also need two similar invariants about relative links: for all $(d_1, f) \in$ `active` and for any ps, if $d_1(f) =$ `RelLink` ps then

- $\forall d_2.\ d_1, ps \rightsquigarrow d_2 \ \rightarrow \ \exists d'.d, p \rightsquigarrow d'$
- $\forall d_2, h_1, d', h.\ d_1, ps \overset{\rightsquigarrow}{=} d_2, h_1 \ \rightarrow \ d, p \overset{\rightsquigarrow}{=} d', h \ \rightarrow \ h \leq h_1$

The code is thus annotated as follows.

```
let rec aux_resolve (d: dirnode) (p:path) (active:set lnk) : dirnode
  requires { subset active alllinks }
  requires { forall d1 f ps d2.
               mem (d1, f) active → lookup d1 f = AbsLink ps →
               resolve_to root ps d2 → exists r. resolve_to d p r }
  requires { forall d1 f ps d2.
               mem (d1, f) active → lookup d1 f = RelLink ps →
               resolve_to d1 ps d2 → exists r. resolve_to d p r }
  requires { forall d1 f ps d2 h1 d' h.
               mem (d1,f) active → lookup d1 f = AbsLink ps →
               resolve_with_height root ps (Some(d2, h1)) →
               resolve_with_height d p (Some(d', h)) → h ≤ h1 }
  requires { forall d1 f ps d2 h1 d' h.
               mem (d1,f) active → lookup d1 f = RelLink ps →
               resolve_with_height d1 ps (Some(d2, h1)) →
               resolve_with_height d p (Some (d',h)) → h ≤ h1 }
  ensures { resolve_to d p result }
  raises { Error → forall d'. not resolve_to d p d' }
  variant { cardinal alllinks - cardinal active, p }
= assert { exists h. resolve_with_height d p h }; (* to help provers *)
  match p with
  | Nil → d
  | Cons Up pr → let d' = parent d in aux_resolve d' pr active
  | Cons Here pr → aux_resolve d pr active
  | Cons (Down f) pr →
    match lookup d f with
    | Absent → raise Error
    | Dir d' → aux_resolve d' pr active
    | AbsLink ps →
      if mem (d,f) active
      then raise Error
      else begin
        let actadd = add (d,f) active in
        let d' = aux_resolve root ps actadd in
        aux_resolve d' pr active
      end
    | RelLink ps →
```

Prover	number of VCs solved	min time	max time	average time	number of VCs solved only by this prover
Coq (8.5pl3)	1	0.52	0.52	0.52	1
CVC3 (2.4.1)	115	0.01	2.89	0.20	9
CVC4 (1.4)	136	0.01	2.37	0.14	8
Alt-Ergo (1.01)	119	0.00	8.17	0.41	7
Eprover (1.8-001)	125	0.01	7.87	0.26	6
Z3 (4.4.1)	108	0.00	6.25	0.26	0

Figure 3: Summary of proof results

```
          if mem (d, f) active
          then raise Error
          else begin
            let actadd = add (d, f) active in
            let d' = aux_resolve d ps actadd in
            aux_resolve d' pr active
          end
        end
      end

    let resolve (d: dirnode) (p:path) : dirnode
      ensures { resolve_to d p result }
      raises { Error → forall d2. not resolve_to d p d2 }
    = aux_resolve d p empty
```

The assertion in the first line of the body of `aux_resolve` is added to help the provers to instantiate the lemma 13.

The proof of the exceptional post-condition must be done in the case of each occurrence of `raise Error` in the code. The first case, when the considered filename does not exists, is easy: no rules for construct a proof of resolution can apply. The two other cases concern the symbolic links. Let's consider the first case, for an absolute link (the other case is similar). By contradiction, if we assume that it is possible to resolve d, p to some d', then this proof has some finite height h. But then since (d, f) is in the active set and points to `AbsLink` ps, the first part of the invariant says that `root`, ps is resolvable. Moreover, the second part of the invariant says that the height of that resolution is some $h_1 \geq h$. By applying the rule ResolveAbsLink we can then build a proof of resolution of d, p of height $h' = 1 + \min(h_1, h'')$ where h'' the height of the proof of resolution of the remaining path `pr`. Hence $h' \geq h_1 + 1$, but by uniqueness of resolution h' must be equal to h, contradicting $h_1 \geq h$.

4.4 Proof results

The table of Figure 3 summarizes the provers' results on all the verification conditions of our development. The total number of VCs is 198. We run all provers on all VCs with a time limit of 10 seconds. The first column gives the number of VCs successfully proved by the given prover. The other columns give respectively the minimum, average and maximum time the prover took to solve the VCs it proved. The last column gives the number of VCs that are proved only by the given prover. This number is 0 for Z3, meaning that Z3 is not really needed, but all the other provers are needed to make a complete proof of our development.

Notice that we needed one Coq proof to solve one VC in the part where we prove the equivalence between our definition of resolution and the one closer to POSIX informal definition. This Coq proof is not complex at all (see appendix), but surprisingly is no solved by any of our provers.

In the appendix we give the details of each verification conditions and which transformations and provers we used to discharge them.

5 Conclusions

We designed a formal specification of the intended meaning of pathname resolution in a Unix file system. We considered an algorithm that is not limited in the number of traversed symbolic links, and we formally proved that this algorithm is terminating, correct and complete. The main difficulty of this work is to design an adequate definition of the meaning of path resolution under the form of a ternary predicate $d_1, p \rightsquigarrow d_2$, and also, in order to achieve the formal verification of the algorithm, to design an adequate variant of this predicate, indexed with an explicit height of the derivation.

This idea of using the height of the derivation is a new lesson we learned during this work. In particular, such a concept have not be used so far in the formal verification of other algorithms for graph traversal. It seems that similar approaches exist in formal reasoning about semantics of programmation languages, for example the technique so-called *step-indexing* [1].

The path resolution algorithm is indeed some kind of graph traversal, and its formal proof could be compared with those of standard graph algorithms. There is a collection of such graph algorithms proved using Why3 due to Chen and Lévy[6] [2]. It seems that the presence of symbolic links in the Unix filesystem adds a significant difficulty to reason about graph traversal, which required the use our technique of indexing with height.

There exists an increasing amount of work on formal reasoning about Unix, file systems and shell scripts. Gardner, Ntzik and Wright proposed a framework based on an ad-hoc separation logic to reason about Unix commands that modify the file system [4]. Our own work is conducted within the CoLiS project which aims at reasoning about shell scripts for package installation. In CoLiS, a full formalization of the file system, including owners, groups, permissions and such is in progress. In this context, a subset of the POSIX shell is already formalized using Why3 [6].

References

[1] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, September 2001.

[2] Ran Chen and Jean-Jacques Lévy. Une preuve formelle de l'algorithme de Tarjan-1972 pour trouver les composantes fortement connexes dans un graphe. In *Vingt-huitièmes Journées Francophones des Langages Applicatifs*, Gourette, France, January 2017.

[3] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.

[4] Philippa Gardner, Gian Ntzik, and Adam Wright. Local reasoning for the POSIX file system. In *ESOP*, volume 8410 of *Lecture Notes in Computer Science*, pages 169–188. Springer, 2014.

[5] IEEE and The Open Group. *POSIX.1-2008/Cor 1-2013*. `http://pubs.opengroup.org/onlinepubs/9699919799/`.

[6] Nicolas Jeannerod. Le coquillage dans le CoLiS-mateur: formalisation d'un langage de programmation de type Shell. In *Vingt-huitièmes Journées Francophones des Langages Applicatifs*, Gourette, France, January 2017.

[6] `pauillac.inria.fr/~levy/why3/`

A Complete Annotated Code

The full annotated code is given below.

```
(** {1 A Formal Proof of an Unix Path Resolution Algorithm}


*)

(** {2 Formalization of File Systems and Path Resolution} *)

module FileSystem

  use import int.Int
  use import int.MinMax

  use export list.List
  use export option.Option

(** {3 Pathnames} *)

  type filename
  type pathcomponent = Down filename | Up | Here
  (** Up denotes ".." and Here denotes "." *)
  type path = list pathcomponent

(** {3 File System} *)

  type dirnode
  type child =
    | Absent
    | Dir dirnode
    | AbsLink path
    | RelLink path
  constant root : dirnode
  function lookup dirnode filename : child
  function parent dirnode : dirnode
    axiom parent_root: parent root = root
    axiom parent_non_root: forall d1 f d2. lookup d1 f = Dir d2 -> parent d2 = d1


(** {3 Resolution Predicates} *)

  inductive resolve_to dirnode path dirnode =
    | ResolveNil : forall d. resolve_to d Nil d
    | ResolveDir : forall d1 fn d2 p d3.
      lookup d1 fn = Dir d2 -> resolve_to d2 p d3 -> resolve_to d1 (Cons (Down fn) p) d3
    | ResolveAbsLink : forall d1 fn ps pr d2 d3.
      lookup d1 fn = AbsLink ps -> resolve_to root ps d2 -> resolve_to d2 pr d3 ->
      resolve_to d1 (Cons (Down fn) pr) d3
    | ResolveRelLink : forall d1 fn ps pr d2 d3.
      lookup d1 fn = RelLink ps -> resolve_to d1 ps d2 -> resolve_to d2 pr d3 ->
      resolve_to d1 (Cons (Down fn) pr) d3
    | ResolveUp: forall d1 d2 d3 p.
      parent d1 = d2 -> resolve_to d2 p d3 -> resolve_to d1 (Cons Up p) d3
    | ResolveHere: forall d1 p d2.
      resolve_to d1 p d2 -> resolve_to d1 (Cons Here p) d2

  inductive resolve_with_height dirnode path (option (dirnode,int)) =
    | ResolveHeightAbsent: forall d p. (forall d1. not resolve_to d p d1) ->
                        resolve_with_height d p None
```

```
    | ResolveHeightNil : forall d. resolve_with_height d Nil (Some (d,0))
    | ResolveHeightDir : forall d1 fn d2 p d3 h.
      lookup d1 fn = Dir d2 -> resolve_with_height d2 p (Some(d3,h)) ->
      resolve_with_height d1 (Cons (Down fn) p) (Some (d3,h + 1))
    | ResolveHeightAbsLink : forall d1 fn ps pr d2 d3 h1 h2.
      lookup d1 fn = AbsLink ps -> resolve_with_height root ps (Some (d2,h1)) ->
      resolve_with_height d2 pr (Some (d3,h2)) ->
      resolve_with_height d1 (Cons (Down fn) pr) (Some (d3, max h1 h2 + 1))
    | ResolveHeightRelLink : forall d1 fn ps pr d2 d3 h1 h2.
      lookup d1 fn = RelLink ps -> resolve_with_height d1 ps (Some (d2,h1)) ->
      resolve_with_height d2 pr (Some (d3,h2)) ->
      resolve_with_height d1 (Cons (Down fn) pr) (Some (d3,max h1 h2 + 1))
    | ResolveHeightUp : forall d1 d2 d3 p h.
      parent d1 = d2 -> resolve_with_height d2 p (Some(d3, h)) ->
      resolve_with_height d1 (Cons Up p) (Some (d3,h + 1))
    | ResolveHeightHere : forall d1 p d2 h.
      resolve_with_height d1 p (Some (d2,h)) ->
      resolve_with_height d1 (Cons Here p) (Some (d2,h + 1))


  lemma resolve_height_resolve : forall d1 p d2 h.
    resolve_with_height d1 p (Some(d2, h)) -> resolve_to d1 p d2

  lemma resolve_height_pos : forall d1 p d2 h.
    resolve_with_height d1 p (Some (d2,h)) -> h >= 0

  lemma resolve_make_height : forall d1 p d2. resolve_to d1 p d2 ->
    exists h. resolve_with_height d1 p (Some(d2, h))

end



(** {2 Conformance with POSIX informal definition} *)

module POSIX_resolution

  use import FileSystem
  use import list.Append

  lemma resolve_to_append : forall d1 d2 d3 p q.
     resolve_to d1 p d2 -> resolve_to d2 q d3 ->
     resolve_to d1 (p ++ q) d3

  lemma resolve_to_decomp : forall d1 d3:dirnode, c:pathcomponent, p:path.
    resolve_to d1 (Cons c p) d3 ->
    exists d2. resolve_to d1 (Cons c Nil) d2 /\ resolve_to d2 p d3

  lemma resolve_to_decomposition : forall p1 p2:path, d1 d3:dirnode.
    resolve_to d1 (p1 ++ p2) d3 ->
    exists d2. resolve_to d1 p1 d2 /\ resolve_to d2 p2 d3

  inductive resolve_to_POSIX dirnode path dirnode =
    | ResolveNilPOSIX : forall d. resolve_to_POSIX d Nil d
    | ResolveDirPOSIX : forall d1 fn d2 p d3.
      lookup d1 fn = Dir d2 -> resolve_to_POSIX d2 p d3 ->
      resolve_to_POSIX d1 (Cons (Down fn) p) d3
    | ResolveAbsLinkPOSIX : forall d1 fn ps pr d2.
      lookup d1 fn = AbsLink ps -> resolve_to_POSIX root (ps ++ pr) d2 ->
      resolve_to_POSIX d1 (Cons (Down fn) pr) d2
    | ResolveRelLinkPOSIX : forall d1 fn ps pr d2.
      lookup d1 fn = RelLink ps -> resolve_to_POSIX d1 (ps ++ pr) d2 ->
```

```
      resolve_to_POSIX d1 (Cons (Down fn) pr) d2
    | ResolveUpPOSIX: forall d1 d2 d3 p.
      parent d1 = d2 -> resolve_to_POSIX d2 p d3 ->
      resolve_to_POSIX d1 (Cons Up p) d3
    | ResolveHerePOSIX: forall d1 p d2.
      resolve_to_POSIX d1 p d2 -> resolve_to_POSIX d1 (Cons Here p) d2

  lemma resolve_to_POSIX_append:
    forall d1 d2 p1. resolve_to_POSIX d1 p1 d2 ->
      forall p2 d3. resolve_to_POSIX d2 p2 d3 ->
      resolve_to_POSIX d1 (p1 ++ p2) d3

  lemma resolve_to_equivalence:
    forall d1 d2 p. resolve_to_POSIX d1 p d2 <-> resolve_to d1 p d2


end

(** {2 Determinism of Path Resolution} *)

theory Determinism

  use import int.Int
  use import FileSystem

  lemma resolve_unique : forall d1 p d2.
    resolve_to d1 p d2 -> forall d3. resolve_to d1 p d3 -> d2 = d3

  lemma resolve_with_height_unique:
      forall d p h1. resolve_with_height d p h1 ->
      forall h2. resolve_with_height d p h2 -> h1 = h2
      by match h1, h2 with
      | None, None -> true
      | _, None | None, _ -> false
      | Some(_,u), Some(_,v) -> u = v end

  lemma resolve_with_height_exists :
    forall d p. exists h. resolve_with_height d p h

end


(** {2 Path Resolution Algorithm} *)

module Resolution

  use import int.Int
  use import FileSystem
  use Determinism

  (** obvious technical lemmas, but needed to help provers *)

  lemma resolve_height_absLink:
    forall d x p r h ps d' h'. resolve_with_height d (Cons (Down x) p) (Some(r, h)) ->
      lookup d x = AbsLink ps -> resolve_with_height root ps (Some(d', h')) -> h' <= h

  lemma resolve_height_relLink:
    forall d x p r h ps d' h'. resolve_with_height d (Cons (Down x) p) (Some(r, h)) ->
      lookup d x = RelLink ps -> resolve_with_height d ps (Some(d', h')) -> h' <= h

  use import set.Fset
  use import map.Map
```

```
  type lnk = (dirnode,filename)
  constant alllinks : set lnk
  axiom alllinks_in : forall d fn ps.
    lookup d fn = AbsLink ps \/ lookup d fn = RelLink ps -> mem (d,fn) alllinks

  exception Error

  let rec aux_resolve (d: dirnode) (p:path) (active:set lnk) : dirnode
    requires { subset active alllinks }
    requires { forall d1 fn ps d2.
               mem (d1, fn) active -> lookup d1 fn = AbsLink ps ->
               resolve_to root ps d2 -> exists r. resolve_to d p r}
    requires { forall d1 fn ps d2.
               mem (d1, fn) active -> lookup d1 fn = RelLink ps ->
               resolve_to d1 ps d2 -> exists r. resolve_to d p r}
    requires { forall d1 fn ps d2 h1 d' h.
               mem (d1,fn) active -> lookup d1 fn = AbsLink ps ->
               resolve_with_height root ps (Some(d2, h1)) ->
               resolve_with_height d p (Some(d', h)) -> h <= h1 }
    requires { forall d1 fn ps d2 h1 d' h.
               mem (d1,fn) active -> lookup d1 fn = RelLink ps ->
               resolve_with_height d1 ps (Some(d2, h1)) ->
               resolve_with_height d p (Some (d',h)) -> h <= h1 }
    ensures { resolve_to d p result }
    raises { Error -> forall d'. not resolve_to d p d' }
    variant { cardinal alllinks - cardinal active, p }
  = assert {exists h. resolve_with_height d p h }; (* to help provers *)
    match p with
    | Nil -> d
    | Cons Up pr -> let d' = parent d in aux_resolve d' pr active
    | Cons Here pr -> aux_resolve d pr active
    | Cons (Down fn) pr ->
      match lookup d fn with
      | Absent -> raise Error
      | Dir d' -> aux_resolve d' pr active
      | AbsLink ps ->
        if mem (d,fn) active
        then raise Error
        else begin
          let actadd = add (d,fn) active in
          let d' = aux_resolve root ps actadd in
          aux_resolve d' pr active
        end
      | RelLink ps ->
        if mem (d, fn) active
        then raise Error
        else begin
          let actadd = add (d, fn) active in
          let d' = aux_resolve d ps actadd in
          aux_resolve d' pr active
        end
      end
    end

  let resolve (d: dirnode) (p:path) : dirnode
    ensures { resolve_to d p result }
    raises { Error -> forall d2. not resolve_to d p d2 }
  = aux_resolve d p empty

end
```

B Detailed Proof Results

The tables from Figure 4, 5, 6, 7, 8 and 9 give the details of each verification conditions and which transformations and provers we used to discharge them. The script for the only proof for which need Coq (in Figure 4) is the following. There is no great difficulty, indeed, it is unclear why no automatic prover is able to discharge it.

```
intros p1 x x1 h1 h2 p2 d1 d3 h3.
subst p1.
simpl in h3.
destruct (resolve_to_decomp _ _ _ _ h3) as (d2 & h4 & h5).
destruct (h2 _ _ _ h5) as (d4 & h6 & h7).
exists d4.
split; auto.
replace (x :: x1)%list with ((x :: nil) ++ x1)%list by auto.
apply resolve_to_append with d2; auto.
```

Proof obligations	Alt-Ergo (1.01)	CVC3 (2.4.1)	CVC4 (1.4)	Coq (8.5pl3)	Eprover (1.8-001)	Z3 (4.4.1)
resolve_to_append						
transformation `induction_pr`						
1.	0.01	0.03	0.05		0.02	0.01
2.	(10s)	(10s)	0.04		0.21	(10s)
3.	(10s)	(10s)	0.05		0.52	(10s)
4.	(10s)	(10s)	0.07		0.23	(10s)
5.	(10s)	(10s)	0.04		0.20	(10s)
6.	0.13	(10s)	0.05		0.20	(10s)
resolve_to_decomp						
transformation `induction_pr`						
1.	0.01	0.02	0.03		0.02	0.01
2.	(10s)	(10s)	(10s)		0.14	(10s)
3.	(10s)	(10s)	(10s)		0.45	0.03
4.	(10s)	(10s)	(10s)		0.50	0.02
5.	(10s)	(10s)	0.77		0.13	(10s)
6.	0.70	0.08	0.54		0.13	0.03
resolve_to_decomposition						
transformation `induction_ty_lex`						
1.						
transformation `split_goal_wp`						
1.	0.09	0.03	0.05		0.13	0.02
2.	(10s)	(10s)	(10s)	0.52	(10s)	(10s)
resolve_to_POSIX_append						
transformation `induction_pr`						
1.	0.02	0.03	0.05		0.02	0.03
2.	(10s)	(10s)	0.04		0.26	(10s)
3.	(10s)	(10s)	(10s)		0.42	(10s)
4.	(10s)	(10s)	(10s)		0.22	(10s)
5.	(10s)	(10s)	0.04		0.28	(10s)
6.	(10s)	(10s)	0.04		0.28	0.91
resolve_to_equivalence						
transformation `split_goal_wp`						
1.						
transformation `induction_pr`						
1.	0.02	0.03	0.04		0.02	0.02
2.	(10s)	(10s)	0.04		0.27	(10s)
3.	(10s)	(10s)	0.06		(10s)	(10s)
4.	(10s)	(10s)	0.08		(10s)	(10s)
5.	(10s)	(10s)	0.07		0.27	(10s)
6.	0.02	0.04	0.04		0.26	0.03
2.						
transformation `induction_pr`						
1.	0.02	0.03	0.04		0.03	0.02
2.	(10s)	(10s)	0.04		0.28	(10s)
3.	(10s)	(10s)	0.05		0.29	(10s)
4.	(10s)	(10s)	0.05		0.31	(10s)
5.	(10s)	(10s)	0.05		0.27	(10s)
6.	0.02	0.04	0.03		0.26	0.03

Figure 4: Detailed proof results for equivalence lemmas between POSIX definition of resolution and ours

Proof obligations	Alt-Ergo (1.01)	CVC3 (2.4.1)	CVC4 (1.4)	Eprover (1.8-001)	Z3 (4.4.1)
resolve_height_resolve					
transformation `induction_pr`					
1.	0.01	0.01	0.02	0.02	0.01
2.	0.01	0.01	0.03	0.10	0.01
3.	(10s)	(10s)	0.04	4.29	(10s)
4.	(10s)	(10s)	0.05	(10s)	(10s)
5.	(10s)	(10s)	0.04	(10s)	(10s)
6.	(10s)	1.71	0.04	0.14	(10s)
7.	0.05	0.11	0.03	0.13	0.02
resolve_height_pos					
transformation `induction_pr`					
1.	0.00	0.01	0.02	0.01	0.00
2.	0.01	0.02	0.03	0.10	0.01
3.	(10s)	(10s)	0.01	0.13	(10s)
4.	(10s)	(10s)	0.03	0.22	(10s)
5.	(10s)	(10s)	0.02	0.12	(10s)
6.	(10s)	(10s)	0.02	0.20	(10s)
7.	(10s)	(10s)	0.02	0.13	(10s)
resolve_make_height					
transformation `induction_pr`					
1.	0.01	8.42	(10s)	0.04	(10s)
2.	(10s)	(10s)	0.04	1.51	(10s)
3.	(10s)	(10s)	0.05	(10s)	(10s)
4.	(10s)	(10s)	0.05	(10s)	(10s)
5.	(10s)	(10s)	0.04	0.86	1.48
6.	(10s)	(10s)	0.03	0.40	0.63

Figure 5: Detailed proof results for equivalence lemmas between resolve and resolve with height

Proof obligations	Alt-Ergo (1.01)	CVC3 (2.4.1)	CVC4 (1.4)	Eprover (1.8-001)	Z3 (4.4.1)
resolve_with_height_exists	0.02	(10s)	0.03	0.17	(10s)

Proof obligations	Alt-Ergo (1.01)	CVC3 (2.4.1)	CVC4 (1.4)	Eprover (1.8-001)	Z3 (4.4.1)
resolve_unique					
transformation `induction_pr`					
1.					
transformation `simplify_trivial_quantification_in_goal`					
1.	0.03	(10s)	7.00	0.17	(10s)
2.					
transformation `simplify_trivial_quantification_in_goal`					
1.	(10s)	0.85	(10s)	(10s)	(10s)
3.					
transformation `simplify_trivial_quantification_in_goal`					
1.					
transformation `inversion_pr`					
1.	0.01	0.02	0.04	0.02	0.01
2.	0.01	0.02	0.04	0.17	0.02
3.	(10s)	0.78	1.63	0.17	0.07
4.	0.01	0.02	0.03	0.17	0.01
5.	0.01	0.03	0.04	0.06	0.02
6.	0.02	0.06	0.04	0.06	0.02
4.					
transformation `simplify_trivial_quantification_in_goal`					
1.					
transformation `inversion_pr`					
1.	0.01	0.02	0.04	0.02	0.02
2.	0.01	0.03	0.04	0.18	0.02
3.	0.01	0.02	0.04	0.18	0.02
4.	(10s)	0.06	1.31	0.18	0.06
5.	0.01	0.02	0.04	0.06	0.02
6.	0.01	0.07	0.04	0.06	0.02
5.					
transformation `simplify_trivial_quantification_in_goal`					
1.	4.08	0.91	6.28	(10s)	(10s)
6.					
transformation `simplify_trivial_quantification_in_goal`					
1.	3.52	0.31	4.09	(10s)	(10s)

Figure 6: Detailed proof results for determinism lemmas (part 1)

Proof obligations	Alt-Ergo (1.01)	CVC3 (2.4.1)	CVC4 (1.4)	Eprover (1.8-001)	Z3 (4.4.1)
resolve_with_height_unique					
transformation `split_goal_wp`					
1.	0.03	(10s)	(10s)	(10s)	(10s)
2.	0.05	(10s)	(10s)	7.87	(10s)
3.					
transformation `induction_pr`					
1.					
transformation `inversion_pr`					
1.	0.02	0.02	0.04	0.02	0.02
2.	0.02	0.02	0.03	0.02	0.02
3.	0.02	0.03	0.04	0.02	0.02
4.	0.02	0.03	0.04	0.02	0.02
5.	0.02	0.03	0.04	0.01	0.02
6.	0.02	0.03	0.04	0.02	0.02
7.	0.02	0.03	0.14	0.02	0.02
2.	0.01	(10s)	(10s)	0.17	(10s)
3.					
transformation `inversion_pr`					
1.	0.02	0.03	0.05	0.02	0.02
2.	0.02	0.03	0.04	0.02	0.02
3.	(10s)	(10s)	0.05	0.24	(10s)
4.	0.02	0.03	0.05	0.24	0.02
5.	0.02	0.03	0.05	0.19	0.02
6.	0.02	0.02	0.04	0.18	0.02
7.	0.02	0.08	0.04	0.18	0.02
4.					
transformation `inversion_pr`					
1.	0.02	0.03	0.04	0.02	0.02
2.	0.02	0.03	0.05	0.02	0.03
3.	0.02	0.03	0.06	0.19	0.02
4.	(10s)	(10s)	0.12	(10s)	(10s)
5.	0.02	0.04	0.05	0.18	0.02
6.	0.02	0.03	0.05	0.19	0.02
7.	0.02	0.15	0.05	0.19	0.02
5.					
transformation `inversion_pr`					
1.	0.02	0.03	0.05	0.02	0.02
2.	0.02	0.03	0.04	0.02	0.02
3.	0.02	0.03	0.05	0.18	0.02
4.	0.02	0.03	0.06	0.18	0.03
5.	(10s)	(10s)	0.13	(10s)	(10s)
6.	0.02	0.03	0.05	0.11	0.03
7.	0.02	0.20	0.05	0.19	0.02
6.					
transformation `inversion_pr`					
1.	0.02	0.02	0.04	0.02	0.02
2.	0.02	0.02	0.04	0.02	0.02
3.	0.02	0.02	0.05	0.17	0.03
4.	0.02	0.03	0.05	0.18	0.02
5.	0.02	0.03	0.05	0.18	0.02
6.	(10s)	(10s)	0.14	0.20	(10s)
7.	0.02	0.08	0.04	0.18	0.02
7.					
transformation `inversion_pr`					
1.	0.02	0.03	0.04	0.02	0.02
2.	0.02	0.03	0.04	0.01	0.02
3.	0.02	0.08	0.03	0.17	0.02
4.	0.02	0.14	0.05	0.18	0.02
5.	0.02	0.14	0.05	0.18	0.02
6.	0.02	0.09	0.04	0.18	0.03
7.	(10s)	(10s)	0.04	0.21	(10s)
4.	0.18	(10s)	0.05	0.18	0.03

Figure 7: Detailed proof results for determinism lemmas (part 2)

Proof obligations	Alt-Ergo (1.01)	CVC3 (2.4.1)	CVC4 (1.4)	Eprover (1.8-001)	Z3 (4.4.1)
VC for aux_resolve					
transformation `split_goal_wp`					
1. assertion	0.05	(10s)	4.74	0.12	(10s)
2. postcondition	0.01	0.04	0.06	0.03	0.03
3. variant decrease	0.01	0.04	0.07	0.23	0.03
4. precondition	0.01	0.04	0.06	0.07	0.03
5. precondition	0.23	0.24	(10s)	(10s)	(10s)
6. precondition	0.21	0.24	(10s)	(10s)	(10s)
7. precondition	(10s)	(10s)	0.14	(10s)	0.03
8. precondition	(10s)	(10s)	0.14	(10s)	0.04
9. postcondition	0.02	0.04	0.07	0.08	0.03
10. exceptional postcondition	0.11	0.11	4.76	(10s)	2.33
11. variant decrease	0.02	0.04	0.06	0.22	0.04
12. precondition	0.02	0.04	0.05	0.07	0.03
13. precondition	0.19	0.24	(10s)	(10s)	(10s)
14. precondition	0.19	0.23	(10s)	(10s)	(10s)
15. precondition	2.45	(10s)	0.15	(10s)	0.03
16. precondition	2.16	(10s)	0.14	(10s)	0.03
17. postcondition	0.02	0.04	0.07	0.25	0.02
18. exceptional postcondition	0.10	0.11	4.66	(10s)	2.29
19. exceptional postcondition	0.04	0.08	(10s)	(10s)	1.71
20. variant decrease	0.02	0.04	0.06	0.23	0.04
21. precondition	0.02	0.03	0.05	0.07	0.02
22. precondition	0.15	0.28	(10s)	(10s)	(10s)
23. precondition	0.13	0.27	(10s)	(10s)	(10s)
24. precondition	(10s)	(10s)	0.12	(10s)	0.03
25. precondition	(10s)	(10s)	0.13	(10s)	0.04
26. postcondition	0.02	0.04	0.06	0.26	0.03
27. exceptional postcondition	0.05	0.12	7.92	(10s)	(10s)
28. exceptional postcondition	4.50	(10s)	(10s)	(10s)	(10s)
29. variant decrease	0.03	0.05	0.07	(10s)	0.03
30. precondition	(10s)	0.13	0.07	(10s)	0.02
31. precondition	0.55	0.34	(10s)	(10s)	(10s)
32. precondition	0.45	0.37	(10s)	(10s)	(10s)
33. precondition	8.17	(10s)	(10s)	(10s)	(10s)
34. precondition	3.90	(10s)	(10s)	(10s)	(10s)
35. variant decrease	0.03	0.05	0.07	0.24	(10s)
36. precondition	0.02	0.04	0.05	0.08	0.02
37. precondition	(10s)	1.65	(10s)	(10s)	(10s)
38. precondition	(10s)	1.81	(10s)	(10s)	(10s)
39. precondition	(10s)	(10s)	2.37	(10s)	4.80
40. precondition	(10s)	(10s)	2.35	(10s)	1.46
41. postcondition	0.03	(10s)	0.07	(10s)	0.03
42. exceptional postcondition	(10s)	0.96	(10s)	(10s)	(10s)
43. exceptional postcondition	0.05	0.24	(10s)	(10s)	(10s)
44. exceptional postcondition	1.62	(10s)	(10s)	(10s)	6.25
45. variant decrease	0.03	0.04	0.07	(10s)	0.03
46. precondition	0.03	(10s)	4.79	(10s)	(10s)
47. precondition	0.48	0.31	(10s)	(10s)	(10s)
48. precondition	0.47	0.31	6.07	(10s)	(10s)
49. precondition	4.27	(10s)	(10s)	(10s)	(10s)
50. precondition	7.45	(10s)	(10s)	(10s)	(10s)
51. variant decrease	0.03	0.04	0.08	0.25	0.30
52. precondition	0.02	0.04	0.06	0.07	0.03
53. precondition	(10s)	1.92	(10s)	(10s)	(10s)
54. precondition	(10s)	2.89	(10s)	(10s)	(10s)
55. precondition	(10s)	(10s)	2.04	(10s)	3.09
56. precondition	(10s)	(10s)	1.73	(10s)	0.47
57. postcondition	0.03	(10s)	0.08	0.29	0.02
58. exceptional postcondition	(10s)	0.94	(10s)	(10s)	(10s)
59. exceptional postcondition	0.16	0.23	(10s)	(10s)	(10s)

Figure 8: Detailed proof results for resolution programs (part 1)

Proof obligations	Alt-Ergo (1.01)	CVC3 (2.4.1)	CVC4 (1.4)	Eprover (1.8-001)	Z3 (4.4.1)
VC for resolve					
transformation `split_goal_wp`					
1. precondition	0.03	0.03	0.06	0.21	0.03
2. precondition	0.02	0.04	0.05	0.02	0.02
3. precondition	0.01	0.04	0.06	0.02	0.02
4. precondition	0.03	0.04	0.05	0.02	0.03
5. precondition	0.03	0.03	0.06	0.03	0.03
6. postcondition	0.03	0.03	0.05	0.03	0.01
7. exceptional postcondition	0.03	0.04	0.07	0.03	0.00

Figure 9: Detailed proof results for resolution programs (part 2)

RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399