On Verifying a File System Implementation

Konstantine Arkoudas, Karen Zee, Viktor Kuncak, Martin Rinard

MIT Computer Science and AI Lab
{arkoudas,kkz,vkuncak,rinard}@lcs.mit.edu
MIT CSAIL Technical Report 946 VK0118

Abstract. We present a correctness proof for a basic file system implementation. This implementation contains key elements of standard Unix file systems such as inodes and fixed-size disk blocks. We prove the implementation correct by establishing a simulation relation between the specification of the file system (which models the file system as an abstract map from file names to sequences of bytes) and its implementation (which uses fixed-size disk blocks to store the contents of the files).
We used the Athena proof checker to represent and validate our proof. Our experience indicates that Athena's use of block-structured natural deduction, support for structural induction and proof abstraction, and seamless connection with high-performance automated theorem provers were essential to our ability to successfully manage a proof of this size.

Table of Contents

1 Introduction

In this paper we explore the challenges of verifying the core operations of a standard Unix file system [20, 16]. We formalize the specification of the file system as a map from file names to sequences of bytes, then formalize an implementation that uses such standard file system data structures as inodes and fixed-sized disk blocks. We verify the correctness of the implementation by proving the existence of a simulation relation between the specification and the implementation.

The proof is expressed and checked in Athena, an interactive theorem-proving environment based on denotational proof languages (DPLs [3]) for first-order logic with sorts and polymorphism. Athena uses a Fitch-style natural deduction calculus, formalized via the abstraction of *assumption bases*. High-level idioms that are frequently encountered in common mathematical reasoning (such as "pick any x and y $\cdots$" or "assume P in $\cdots$") are directly available to the user. Athena also includes a higher-order functional language in the style of Scheme and ML and offers flexible mechanisms for expressing proof-search algorithms in a trusted manner (akin to the "tactics" and "tacticals" of LCF-like systems such as HOL [11]).

The proof comprises 283 lemmas and theorems, and took 1.5 person-months of full-time work to complete. It consists of roughly 5,000 lines of Athena code, for an average of about 18 lines per lemma. It takes about 9 minutes to check on a high-end Pentium, for an average of 1.9 seconds per lemma. Athena seamlessly integrates cutting-edge automated theorem provers (ATPs) such as Vampire [21] and Spass [22] to mechanically prove tedious steps, leaving the user to focus on the interesting parts of the proof. Athena invokes Vampire and Spass over 2,000 times during the course of the proof. That the proof is still several thousand lines long reflects the sheer size of the problem. For instance, we needed to prove 12 invariants and there are 10 state-transforming operations, which translates to 120 lemmas for each invariant/operation pair (I, f), each guaranteeing that f preserves I. Most of these lemmas are non-trivial; many require induction, and several require a number of other auxiliary lemmas. Further complicating matters is the fact that we can show that some of these invariants are preserved only if we assume that certain other invariants hold. In these cases we must consider simultaneously the conjunction of several invariants. The resulting formulas are several pages long and have dozens of quantified variables. We believe that Athena's combination of natural deduction, versatile mechanisms for proof abstraction, and seamless incorporation of very efficient ATPs were crucial to our ability to successfully complete a proof effort of this scale.

To place our results in a broader context, consider that organizations rely on storage systems in general and file systems in particular to store critical persistent data. Because errors can cause the file system to lose this data, it is important for the implementation to be correct. The standard wisdom is that core system components such as file systems will always remain beyond the reach of full correctness proofs, leaving extensive testing—and the possibility of undetected residual errors—as the only option. Our results, however, suggest that correctness proofs for crucial system components (especially for the key algorithms and data structures at the heart of such components) may very well be within reach.

The remainder of the paper is structured as follows. Section 2 informally describes a simplified file system. Section 3 presents an abstract specification of the file system. This specification hides the complexity of implementation-specific data structures such as inodes and data blocks by representing files simply as indexable sequences of bytes. Section 4 presents our model of the implementation of the file system. This implementation contains many more details, e.g., the mapping from file names to inodes, as well as the representation of file contents using sequences of non-contiguous data blocks that are dynamically allocated on the disk. Section 5 presents the statement of the correctness criterion. This criterion uses an abstraction function [15] that maps the state of the implementation to the state of the specification. Section 5 also sketches out the overall strategy of the proof. Section 6 and Section 7 address the key role that invariants and proof tactics played in this project. Section 8 gives a flavor of our correctness proof by presenting a proof of a frame-condition lemma. Section 9 presents related work, and Section 10 concludes. The Appendix contains a description of the relevant parts of certain Athena libraries that were used in this project.

2 A Simple File System

In this section we describe the high-level structure of a simple file system. In Section 4 we present a formal model of such a file system.

In our file system the physical media is divided into blocks containing a fixed number of bytes. The contents of a file are divided into block-sized segments, and stored in a series of blocks that are not necessarily consecutive.

The file system associates each file with an *inode*, which is a data structure that contains information about the file, including the file size and which blocks contain the file data. Unlike actual UNIX file systems, the inodes in our system do not contain other information such as access privileges and time stamps.

There is only one directory, the root directory, which maps file names to inode numbers. No two file names can refer to the same file, so no two file identifiers can map to the same inode number. We also assume that the disk is unbounded—the file system has access to an infinite number of inodes and blocks.

To read a byte from a given file, the file system first looks up the file name in the root directory, and obtains the number of the corresponding inode. Assuming the file exists, the file system then looks up the inode. From the information in the inode, the file system determines if it is reading a byte that is within the bounds of the file size, and if so, which block contains the relevant byte. Finally, the file system reads the byte from that block and returns the value read.

A similar look-up process occurs when writing a byte in a file. In this case, if the file system is writing a byte that is within the bounds of the existing file size, it simply stores the new value to the appropriate byte. Otherwise, the file system extends the file up to the index of the byte it is writing. It then stores the appropriate value to the byte it is writing, and a default pad value to the bytes in between.

Our formalization consists of a set of axioms in first-order logic with sorts, polymorphism, and structural induction. We use generic Athena libraries that contain axiomatizations of natural numbers, value options, finite maps, and resizable arrays; see the Appendix for a brief description of those libraries.

3 Abstract specification of the file system

Our specification is an abstract model of the file system that hides the complexity of data structures such as inodes and data blocks by representing files as indexable sequences of bytes.

The specification uses the following sorts (the first two are introduced as new primitive domains, while the latter two are defined as sort abbreviations):

$$\begin{array}{l} \textbf{sorts } \mathit{Byte}, \mathit{FileID} \\ \textbf{define } \mathit{File} = \mathit{RSArrayOf}(\mathit{Byte}) \\ \textbf{define } \mathit{AbState} = \mathit{FMap}(\mathit{FileID}, \mathit{File}) \end{array}$$

The sort *Byte* is an abstract type whose values represent the units of file content. *FileID* is also an abstract type; its values represent file identifiers. We define *File* as a resizable array of *Byte*. The abstract state of the file system, *AbState*, is represented as a finite map from file identifiers (*FileID*) to file contents (*File*). We also introduce a distinguished element of *Byte*, called *fillByte*, which is used to pad a file in the case of an attempt to write at a position exceeding the file size: **declare** *fillByte* : *Byte*.

3.1 Specification of the abstract read operation

We begin by giving the signature of the abstract read operation, *absRead*:

$$\textbf{declare } \mathit{absRead} : \mathit{FileID} \times \mathit{Nat} \times \mathit{AbState} \rightarrow \mathit{ReadResult}$$

Thus *absRead* takes a file identifier *fid*, an index i in the file, and an abstract file system state s; and returns an element of *ReadResult*. The latter is defined as the following datatype:

$$\begin{array}{rl} \textbf{datatype } \mathit{ReadResult} = & \mathit{EOF} \\ | & \mathit{Ok}(\mathit{Byte}) \\ | & \mathit{FileNotFound} \end{array}$$

Therefore, the result of any *absRead* operation is one of three things: *EOF*, if the index is out of bounds; *FileNotFound*, if the file does not exist; or, if all goes well, a value of the form $\mathit{Ok}(v)$ for some byte v, representing the content of file *fid* at position i. More precisely, the semantics of *absRead* are given by the following three axioms:

$$[\mathbf{AR}_1] \quad \forall\, \mathit{fid}\ i\ s\,.\, \mathit{lookUp}\,(\mathit{fid}, s) = \mathit{NONE} \Rightarrow \boldsymbol{read}(\mathit{fid}, i, s) = \mathit{FileNotFound}$$

$$[\mathbf{AR}_2] \quad \forall\, \mathit{fid}\ i\ s\ \mathit{file}\,.\, [\mathit{lookUp}\,(\mathit{fid}, s) = \mathit{SOME}(\mathit{file}) \wedge \mathit{arrayLen}(\mathit{file}) \leq i] \Rightarrow \boldsymbol{read}(\mathit{fid}, i, s) = \mathit{EOF}$$

$$[\mathbf{AR}_3]\ \forall\, \mathit{fid}\ i\ s\ v\ \mathit{file}\,.\, [\mathit{lookUp}\,(\mathit{fid}, s) = \mathit{SOME}(\mathit{file}) \wedge \mathit{arrayRead}(\mathit{file}, i) = \mathit{SOME}(v)] \Rightarrow \boldsymbol{read}(\mathit{fid}, i, s) = \mathit{Ok}(v)$$

Using the equality conditions for finite maps and resizable arrays, we are able to prove the following extensionality theorem for abstract states:

$$\forall\, \boldsymbol{s}_1\ \boldsymbol{s}_2\,.\, \boldsymbol{s}_1 = \boldsymbol{s}_2 \Leftrightarrow [\forall\, \mathit{fid}\ i\,.\, \boldsymbol{read}(\mathit{fid}, i, \boldsymbol{s}_1) = \boldsymbol{read}(\mathit{fid}, i, \boldsymbol{s}_2)]. \tag{1}$$

3.2 Specification of the abstract write operation

The abstract ***write*** operation has the following signature:

$$\textbf{declare}\ \boldsymbol{write} : \mathit{FileID} \times \mathit{Nat} \times \mathit{Byte} \times \mathit{AbState} \rightarrow \mathit{AbState}$$

This is the operation that defines state transitions in our file system. It takes as arguments a file identifier *fid*, an index i indicating a file position, a byte v representing the value to be written, and a file system state $\boldsymbol{s}$. The result is a new state where the contents of the file associated with *fid* have been updated by storing v at position i. Note that if i exceeds the length of the file in state $\boldsymbol{s}$, then in the resulting state the file will be extended to size $i+1$ and all newly allocated positions below i will be padded with the *fillByte* value. Finally, if *fid* does not correspond to a file in $\boldsymbol{s}$, then an empty file of size $i+1$ is first created and then the value v is written. More precisely, we introduce the following axioms:

$$[\mathbf{AW}_1]\ \ \forall\, \mathit{fid}\ i\ v\ \boldsymbol{s}\,.\, \mathit{lookUp}\,(\mathit{fid}, \boldsymbol{s}) = \mathit{NONE} \Rightarrow \boldsymbol{write}(\mathit{fid}, i, v, \boldsymbol{s}) = \mathit{update}(\boldsymbol{s}, \mathit{fid}, \mathit{arrayWrite}(\mathit{makeArray}(\mathit{fillByte}, i+1), i, v, \mathit{fillByte}))$$

$$[\mathbf{AW}_2]\ \ \forall\, \mathit{fid}\ i\ v\ \boldsymbol{s}\ \mathit{file}\,.\, \mathit{lookUp}\,(\mathit{fid}, \boldsymbol{s}) = \mathit{SOME}(\mathit{file}) \Rightarrow \boldsymbol{write}(\mathit{fid}, i, v, \boldsymbol{s}) = \mathit{update}(\boldsymbol{s}, \mathit{fid}, \mathit{arrayWrite}(\mathit{file}, i, v, \mathit{fillByte}))$$

4 File system implementation

Standard Unix file systems store the contents of each file in separate disk blocks, and maintain a table of structures called *inodes* that index those blocks and store various types of information about the file. Our implementation operates directly on the inodes and disk blocks and therefore models the operations that the file system performs on the disk. We omit details such as file permissions, dates, links, multi-layered directories, and optimizations such as caching. Some of these (e.g., permissions and date stamps) are orthogonal to the verification obligation and could be included with minimal changes to our proof, while others (e.g., caching) would likely introduce additional complexity.

File data is organized in *Block* units. A *Block* is an array of *blockSize* bytes, where *blockSize* is a positive constant. Specifically, we model a *Block* as a finite map from natural numbers to *Byte*:

$$\textbf{define}\ \mathit{Block} = \mathit{FMap}(\mathit{Nat}, \mathit{Byte})$$

We also define a distinguished element of *Block*, called *initialBlock*, such that:

$$\forall\, i \,.\, i < \mathit{blockSize} \Rightarrow \mathit{lookUp}\,(i, \mathit{initialBlock}) = \mathit{SOME}(\mathit{fillByte})$$
$$\forall\, i \,.\, \mathit{blockSize} \leq i \Rightarrow \mathit{lookUp}\,(i, \mathit{initialBlock}) = \mathit{NONE}$$

In other words, an *initialBlock* consists of *blockSize* copies of *fillByte*.

File meta-data is stored in inodes:

$$\textbf{datatype}\ \mathit{INode} = \mathit{inode}(\mathit{fileSize} : \mathit{Nat}, \mathit{blockCount} : \mathit{Nat}, \mathit{blockList} : \mathit{FMap}(\mathit{Nat}, \mathit{Nat}))$$

An *INode* is a datatype consisting of the file size in bytes and in blocks, and a list of block numbers. The list of block numbers is an array of the block numbers that contain the file data. We model this array as a finite map from natural numbers (array indices) to natural numbers (block numbers).

The data type *State* represents the file system state:

$$\textbf{datatype}\ \mathit{State} = \mathit{state}(\mathit{inodeCount} : \mathit{Nat}, \mathit{stateBlockCount} : \mathit{Nat},$$
$$\mathit{inodes} : \mathit{FMap}(\mathit{Nat}, \mathit{INode}), \mathit{blocks} : \mathit{FMap}(\mathit{Nat}, \mathit{Block}), \mathit{root} : \mathit{FMap}(\mathit{FileID}, \mathit{Nat}))$$

A *State* consists of a count of the inodes in use; a count of the blocks in use; an array of inodes; an array of blocks; and the root directory. We model the array of inodes as a finite map from natural numbers (array indices) to *INode* (inodes). Likewise, we model the array of blocks as a finite map from natural numbers (array indices) to *Block* (blocks). We model the root directory as a finite map from *FileID* (file identifiers) to natural numbers (inode numbers).

We also define *initialState*, a distinguished element of *State*, which describes the initial state of the file system. In the initial state, no inodes or blocks are in use, and the root directory is empty:

$$\textbf{declare}\ \mathit{initialState} : \mathit{State}$$
$$\mathit{initialState} = \mathit{state}(0, 0, \mathit{empty\text{-}map}, \mathit{empty\text{-}map}, \mathit{empty\text{-}map})$$

4.1 Definition of the concrete read operation

The concrete read operation, *read*, has the following signature:

$$\textbf{declare}\ \mathit{read} : \mathit{FileID} \times \mathit{Nat} \times \mathit{State} \rightarrow \mathit{ReadResult}$$

The *read*[1] operation takes a file identifier *fid*, an index i in the file, and a concrete file system state s, and returns an element of *ReadResult*. It first determines if *fid* is present in the root directory of s. If not, *read* returns *FileNotFound*. Otherwise, it looks up the corresponding inode. If i is not less than the file size, *read* returns *EOF*. Otherwise, *read* looks up the block containing the data and returns the relevant byte. The following axioms capture these semantics (for ease of presentation, we omit universal quantifiers from now on; all variables can be assumed to be universally quantified):

[1] As a convention, we use bold italic font to indicate the abstract-state version of something: e.g., abstract ***read*** vs. concrete *read*, an abstract state ***s*** vs. a concrete state s, etc.

$$[CR_1] \quad \mathit{lookUp}(\mathit{fid}, \mathit{root}(s)) = \mathit{NONE} \Rightarrow \mathit{read}(\mathit{fid}, i, s) = \mathit{FileNotFound}$$
$$[CR_2] \quad [\mathit{lookUp}(\mathit{fid}, \mathit{root}(s)) = \mathit{SOME}(n) \wedge$$
$$\mathit{lookUp}(n, \mathit{inodes}(s)) = \mathit{SOME}(\mathit{inode}(fs, bc, bl)) \wedge (fs \leq i)] \Rightarrow \mathit{read}(\mathit{fid}, i, s) = \mathit{EOF}$$
$$[CR_3] \quad [\mathit{lookUp}(\mathit{fid}, \mathit{root}(s)) = \mathit{SOME}(n) \wedge$$
$$\mathit{lookUp}(n, \mathit{inodes}(s)) = \mathit{SOME}(\mathit{inode}(fs, bc, bl)) \wedge (i \ < \ fs) \wedge$$
$$\mathit{lookUp}(i \ \mathit{div} \ \mathit{blockSize}, bl) = \mathit{SOME}(bn) \wedge \mathit{lookUp}(bn, \mathit{blocks}(s)) = \mathit{SOME}(\mathit{block}) \wedge$$
$$\mathit{lookUp}(i \ \mathit{mod} \ \mathit{blockSize}, \mathit{block}) = \mathit{SOME}(v)] \Rightarrow \mathit{read}(\mathit{fid}, i, s) = \mathit{Ok}(v)$$

4.2 Definition of the concrete write operation

The concrete write operation, *write*, takes a file identifier *fid*, a byte index i, the byte value v to write, and a state s, and returns the updated state:

$$\textbf{declare} \ \mathit{write} : \mathit{FileID} \times \mathit{Nat} \times \mathit{Byte} \times \mathit{State} \to \mathit{State}$$
$$[CW_1] \quad \mathit{lookUp}(\mathit{fid}, \mathit{root}(s)) = \mathit{SOME}(n) \Rightarrow \mathit{write}(\mathit{fid}, i, v, s) = \mathit{writeExisting}(n, i, v, s)$$
$$[CW_2] \quad \textbf{let} \ s' = \mathit{allocINode}(\mathit{fid}, s) \ \textbf{in}$$
$$[\mathit{lookUp}(\mathit{fid}, \mathit{root}(s)) = \mathit{NONE} \wedge \mathit{lookUp}(\mathit{fid}, \mathit{root}(s')) = \mathit{SOME}(n)] \Rightarrow$$
$$\mathit{write}(\mathit{fid}, i, v, s) = \mathit{writeExisting}(n, i, v, s')$$

If the file associated with *fid* already exists, *write* delegates the write to the helper function *writeExisting*. If the file does not exist, *write* first invokes *allocINode*, which creates a new, empty file, then calls *writeExisting* with the inode number of the new file.

allocINode takes a file identifier *fid* and a state s, and returns an updated state:

$$\textbf{declare} \ \mathit{allocINode} : \mathit{FileID} \times \mathit{State} \to \mathit{State}$$
$$\mathit{getNextINode}(s) = \mathit{state}(\mathit{inc}+1, \mathit{bc}, \mathit{inm}, \mathit{bm}, \mathit{root}) \Rightarrow$$
$$\mathit{allocINode}(\mathit{fid}, s) = \mathit{state}(\mathit{inc}+1, \mathit{bc}, \mathit{inm}, \mathit{bm}, \mathit{update}(\mathit{root}, \mathit{fid}, \mathit{inc}))$$

allocINode creates a new inode by invoking *getNextINode*, then associates *fid* with the new inode.

getNextINode takes a state and returns an updated state. It allocates and initializes a new inode:

$$\textbf{declare} \ \mathit{getNextINode} : \mathit{State} \to \mathit{State}$$
$$\mathit{getNextINode}(\mathit{state}(\mathit{inc}, \mathit{bc}, \mathit{inm}, \mathit{bm}, \mathit{root})) =$$
$$\mathit{state}(\mathit{inc}+1, \mathit{bc}, \mathit{update}(\mathit{inm}, \mathit{inc}, \mathit{inode}(0, 0, \mathit{empty\text{-}map})), \mathit{bm}, \mathit{root})$$

writeExisting takes an inode number n, a byte index i, the byte value v to write, and a state s, and returns the updated state:

$$\textbf{declare} \ \mathit{writeExisting} : \mathit{Nat} \times \mathit{Nat} \times \mathit{Byte} \times \mathit{State} \to \mathit{State}$$
$$[WE_1] \quad [\mathit{lookUp}(n, \mathit{inodes}(s)) = \mathit{SOME}(\mathit{inode}) \wedge$$
$$(i \ \mathit{div} \ \mathit{blockSize}) < \mathit{blockCount}(\mathit{inode}) \wedge i < \mathit{fileSize}(\mathit{inode})] \Rightarrow$$
$$\mathit{writeExisting}(n, i, v, s) = \mathit{writeNoExtend}(n, i, v, s)$$
$$[WE_2] \quad [\mathit{lookUp}(n, \mathit{inodes}(s)) = \mathit{SOME}(\mathit{inode}) \wedge$$
$$(i \ \mathit{div} \ \mathit{blockSize}) < \mathit{blockCount}(\mathit{inode}) \wedge \mathit{fileSize}(\mathit{inode}) \leq i] \Rightarrow$$
$$\mathit{writeExisting}(n, i, v, s) = \mathit{writeSmallExtend}(n, i, v, s)$$
$$[WE_3] \quad [\mathit{lookUp}(n, \mathit{inodes}(s)) = \mathit{SOME}(\mathit{inode}) \wedge$$
$$\mathit{blockCount}(\mathit{inode}) \leq (i \ \mathit{div} \ \mathit{blockSize})] \Rightarrow$$
$$\mathit{writeExisting}(n, i, v, s) = \mathit{writeNoExtend}(n, i, v, \mathit{extendFile}(n, i, s))$$

If i is less than the file size, *writeExisting* delegates the writing to *writeNoExtend*, which stores the value v in the appropriate location. If i is not less than the file size but is located in the last block of the file, *writeExisting* delegates to *writeSmallExtend*, which stores the value v in the appropriate position and updates the file size. Otherwise, *writeExisting* first invokes *extendFile*, which extends the file by the appropriate number of blocks, and then calls *writeNoExtend* on the updated state.

writeNoExtend takes an inode number n, a byte index i, the byte value v to write, and a state s, and returns the updated state after writing v at index i:

$$\begin{gathered}\textbf{declare } \mathit{writeNoExtend} : \mathit{Nat} \times \mathit{Nat} \times \mathit{Byte} \times \mathit{State} \rightarrow \mathit{State} \\ [\mathit{lookUp}\,(n, \mathit{inodes}(s)) = \mathit{SOME}(\mathit{inode}) \wedge \\ \mathit{lookUp}\,(i\ \mathit{div}\ \mathit{blockSize}, \mathit{blockList}(\mathit{inode})) = \mathit{SOME}(\mathit{bn}) \wedge \\ \mathit{lookUp}\,(\mathit{bn}, \mathit{blocks}(s)) = \mathit{SOME}(\mathit{block})] \Rightarrow \\ \mathit{writeNoExtend}(n, i, v, s) = \mathit{updateStateBM}\,(s, \mathit{bn}, \mathit{update}(\mathit{block}, i\ \mathit{mod}\ \mathit{blockSize}, v))\end{gathered}$$

writeNoExtend uses the helper function *updateStateBM*. The function *updateStateBM* takes the state, the block number *bn*, and the block *block*, and returns an updated state where *bn* maps to *block*:

$$\begin{gathered}\textbf{declare } \mathit{updateStateBM} : \mathit{State} \times \mathit{Nat} \times \mathit{Block} \rightarrow \mathit{State} \\ \mathit{updateStateBM}\,(\mathit{state}(\mathit{inc}, \mathit{bc}, \mathit{inm}, \mathit{bm}, \mathit{root}), \mathit{bn}, \mathit{block}) = \\ \mathit{state}(\mathit{inc}, \mathit{bc}, \mathit{inm}, \mathit{update}(\mathit{bm}, \mathit{bn}, \mathit{block}), \mathit{root})\end{gathered}$$

writeSmallExtend takes an inode number n, a byte index i, the byte value v to write, and a state. It updates the file size and writes the byte value v at byte index i for the file associated with the inode number n, and returns the updated state:

$$\begin{gathered}\textbf{declare } \mathit{writeSmallExtend} : \mathit{Nat} \times \mathit{Nat} \times \mathit{Byte} \times \mathit{State} \rightarrow \mathit{State} \\ [\mathit{lookUp}\,(n, \mathit{inm}) = \mathit{SOME}(\mathit{inode}(fs, \mathit{bc}, \mathit{bl})) \wedge \\ \mathit{lookUp}\,(i\ \mathit{div}\ \mathit{blockSize}, \mathit{bl}) = \mathit{SOME}(\mathit{bn}) \wedge \\ \mathit{lookUp}\,(\mathit{bn}, \mathit{bm}) = \mathit{SOME}(\mathit{block}) \wedge fs \leq i] \Rightarrow \\ \mathit{writeSmallExtend}(n, i, v, \mathit{state}(\mathit{snc}, \mathit{sbc}, \mathit{inm}, \mathit{bm}, \mathit{root})) = \\ \mathit{state}(\mathit{snc}, \mathit{sbc}, \mathit{update}(\mathit{inm}, n, \mathit{inode}(i+1, \mathit{bc}, \mathit{bl})), \\ \mathit{update}(\mathit{bm}, \mathit{bn}, \mathit{update}(\mathit{block}, i\ \mathit{mod}\ \mathit{blockSize}, v)), \mathit{root})\end{gathered}$$

extendFile takes an inode number n, the byte index of the write, and the state s. It delegates the task of allocating the necessary blocks to *allocBlocks*:

$$\begin{gathered}\textbf{declare } \mathit{extendFile} : \mathit{Nat} \times \mathit{Nat} \times \mathit{State} \rightarrow \mathit{State} \\ [\mathit{lookUp}\,(n, \mathit{inodes}(s)) = \mathit{SOME}(\mathit{inode}) \wedge \mathit{blockCount}(\mathit{inode}) \leq (j\ \mathit{div}\ \mathit{blockSize})] \Rightarrow \\ \mathit{extendFile}(n, j, s) = \mathit{allocBlocks}(n, (j\ \mathit{div}\ \mathit{blockSize}) - \mathit{blockCount}(\mathit{inode}) + 1, j, s)\end{gathered}$$

allocBlocks takes an inode number n, the number of blocks to allocate, the byte index j, and the state s. We define it by primitive recursion:

$$\begin{gathered}\textbf{declare } \mathit{allocBlocks} : \mathit{Nat} \times \mathit{Nat} \times \mathit{Nat} \times \mathit{State} \rightarrow \mathit{State} \\ [AB_1] \quad \mathit{allocBlocks}(n, 0, j, s) = s \\ [AB_2] \quad [\mathit{getNextBlock}\,(s) = \mathit{state}(\mathit{inc}, \mathit{bc}+1, \mathit{inm}, \mathit{bm}, \mathit{root}) \wedge \\ \mathit{lookUp}\,(n, \mathit{inm}) = \mathit{SOME}(\mathit{inode}(fs, \mathit{inbc}, \mathit{inbl}))] \Rightarrow \\ \mathit{allocBlocks}(n, k+1, j, s) = \mathit{allocBlocks}(n, k, j, \mathit{state}(\mathit{inc}, \mathit{bc}+1, \\ \mathit{update}(\mathit{inm}, n, \mathit{inode}(j+1, \mathit{inbc}+1, \mathit{update}(\mathit{inbl}, \mathit{inbc}, \mathit{bc}))), \mathit{bm}, \mathit{root}))\end{gathered}$$

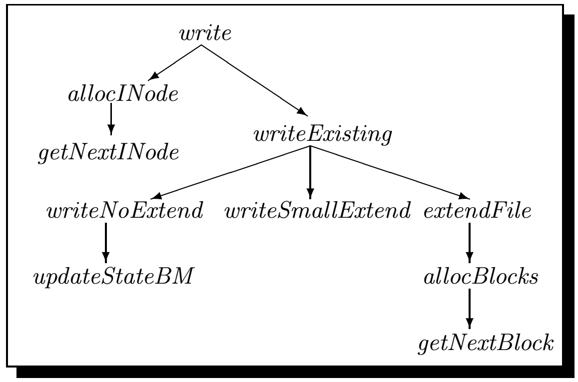


Fig. 1. The call graph of *write*.

allocBlocks uses the helper function *getNextBlock*, which takes the state s, allocates and initializes the next free block, and returns the updated state:

$$\textbf{declare } \mathit{getNextBlock} : \mathit{State} \to \mathit{State}$$
$$\mathit{getNextBlock}(\mathit{state}(\mathit{inc}, \mathit{bc}, \mathit{inm}, \mathit{bm}, \mathit{root})) =$$
$$\mathit{state}(\mathit{inc}, \mathit{bc}+1, \mathit{inm}, \mathit{update}(\mathit{bm}, \mathit{bc}, \mathit{initialBlock}), \mathit{root})$$

The call graph summarizing the *write* operation is shown in Figure 1. This call graph largely determines the auxiliary lemmas that need to be established every time we wish to prove a result about *write*. That is, whenever we need to prove a result L about *write*, we prove appropriate lemmas L_1 and L_2 about *allocINode* and *writeExisting*. In turn, L_1 will rely on a lemma L_{11} about *getNextINode* and L_2 will reference lemmas L_{21}, L_{22}, and L_{23} about *writeNoExtend*, *writeSmallExtend*, and *extendFile*, respectively; and so on. In this way we obtain a lemma dependency graph for L whose structure mirrors that of the call graph for *write*.

In what follows we will restrict our attention to *reachable* states, those that can be obtained from the initial state by some finite sequence of *write* operations. Specifically, we define a predicate *reachableN* ("reachable in n steps") via two axioms: $\mathit{reachableN}(s, 0) \Leftrightarrow s = \mathit{initialState}$, and

$$\mathit{reachableN}(s, n+1) \Leftrightarrow \exists\, s'\ \mathit{fid}\ i\ v\ .\ \mathit{reachableN}(s', n) \wedge s = \mathit{write}(\mathit{fid}, i, v, s')$$

We then set $\mathit{reachable}(s) \Leftrightarrow \exists\, n\ .\ \mathit{reachableN}(s, n)$. We will write $\widehat{\mathit{State}}$ for the set of all reachable states, and we will use the symbol $\widehat{s}$ to denote a reachable state. Propositions of the form $\forall \cdots \widehat{s} \cdots .\ P(\cdots \widehat{s} \cdots)$ and $\exists \cdots \widehat{s} \cdots .\ P(\cdots \widehat{s} \cdots)$ should be taken as abbreviations for $\forall \cdots s \cdots .\ \mathit{reachable}(s) \Rightarrow P(\cdots s \cdots)$ and $\exists \cdots s \cdots .\ \mathit{reachable}(s) \wedge P(\cdots s \cdots)$, respectively.

5 The correctness proof

5.1 State abstraction and homomorphic simulation

This section presents a correctness criterion for the implementation. The correctness criterion is specified using an abstraction function [15] that maps the state of the implementation to the state of the specification.

Consider the following binary relation A from concrete to abstract states:

$$\forall\, s\ \boldsymbol{s}\,.\, A(s,\boldsymbol{s}) \Leftrightarrow [\forall\, \mathit{fid}\ i\,.\, \mathit{read}(\mathit{fid},i,s) = \boldsymbol{read}(\mathit{fid},i,\boldsymbol{s})]$$

It follows directly from the extensionality principle on abstract states (1) that A is functional:

$$\forall\, s\ \boldsymbol{s}_1\ \boldsymbol{s}_2\,.\, A(s,\boldsymbol{s}_1) \wedge A(s,\boldsymbol{s}_2) \Rightarrow \boldsymbol{s}_1 = \boldsymbol{s}_2.$$

Accordingly, we postulate the existence of an abstraction function $\alpha : \mathit{State} \rightarrow \mathit{AbState}$ such that:

$$\forall\, s\ \boldsymbol{s}\,.\, \alpha(s) = \boldsymbol{s} \Leftrightarrow A(s,\boldsymbol{s}).$$

That is, an abstracted state $\alpha(s)$ has the exact same contents as s: reading any position of a file in one state yields the same result as reading that position of the file in the other state.

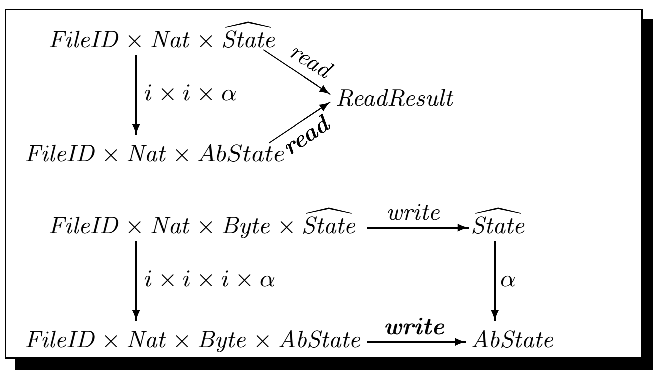


Fig. 2. Commuting diagrams for the *read* and *write* operations.

A standard way of formalizing the requirement that an implementation $\mathcal{I}$ is faithful to a specification $\mathcal{S}$ is to express $\mathcal{I}$ and $\mathcal{S}$ as many-sorted algebras and establish a homomorphism from one to the other. In our case the two algebras are $\mathcal{I} = (\mathit{FileID}, \mathit{Nat}, \mathit{Byte}, \widehat{\mathit{State}}; \mathit{read}, \mathit{write})$ and

$$\mathcal{S} = (\mathit{FileID}, \mathit{Nat}, \mathit{Byte}, \mathit{AbState}; \boldsymbol{read}, \boldsymbol{write})$$

The embeddings from $\mathcal{I}$ to $\mathcal{S}$ for the carriers *FileID*, *Nat*, and *Byte* are simply the identity functions on these domains; while the embedding from $\widehat{\mathit{State}}$ to

AbState is the abstraction mapping α. In order to prove that this translation yields a homomorphism we need to show that the two diagrams shown in Figure 2 commute. Symbolically, we need to prove the following:

$$\forall\, \mathit{fid}\ i\ \widehat{s}\,.\, \mathit{read}(\mathit{fid}, i, \widehat{s}) = \boldsymbol{read}(\mathit{fid}, i, \alpha(\widehat{s})) \tag{2}$$

and

$$\forall\, \mathit{fid}\ i\ v\ \widehat{s}\,.\, \alpha(\mathit{write}(\mathit{fid}, i, v, \widehat{s})) = \boldsymbol{write}(\mathit{fid}, i, v, \alpha(\widehat{s})) \tag{3}$$

5.2 Proof outline

Goal (2) follows immediately from the definition of the abstraction function α. For (3), since the consequent is equality between two abstract states and we have already proven that two abstract states $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$ are equal iff any abstract read operation yields identical results on $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$, we transform (3) into the following:

$$\forall\, \mathit{fid}\ i\ v\ \widehat{s}\ \mathit{fid}'\ j\,.\, \boldsymbol{read}(\mathit{fid}', j, \alpha(\mathit{write}(\mathit{fid}, i, v, \widehat{s}))) = \boldsymbol{read}(\mathit{fid}', j, \boldsymbol{write}(\mathit{fid}, i, v, \alpha(\widehat{s})))$$

Finally, using (2) on the above gives:

$$\forall\, \mathit{fid}\ \mathit{fid}'\ i\ j\ v\ \widehat{s}\,.\, \mathit{read}(\mathit{fid}', j, \mathit{write}(\mathit{fid}, i, v, \widehat{s})) = \boldsymbol{read}(\mathit{fid}', j, \boldsymbol{write}(\mathit{fid}, i, v, \alpha(\widehat{s})))$$

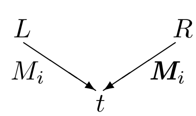
Therefore, choosing arbitrary $\mathit{fid}, \mathit{fid}', j, v, i$, and $\widehat{s}$, we need to show $L = R$, where $L = \mathit{read}(\mathit{fid}', i, \mathit{write}(\mathit{fid}, j, v, \widehat{s}))$ and

$$R = \boldsymbol{read}(\mathit{fid}', i, \boldsymbol{write}(\mathit{fid}, j, v, \alpha(\widehat{s})))$$

Showing $L = R$ is the main goal of the proof. We proceed by a case analysis as shown in Fig. 3. The decision tree of Fig. 3 has the following property: if the conditions that appear on a path from the root of the tree to an internal node u are all true, then the conditions at the children of u are mutually exclusive and jointly exhaustive (given that certain invariants hold, as discussed in Section 6). There are ultimately eight distinct cases to be considered, C_1 through C_8, appearing at the leaves of the tree. Exactly one of those eight cases must be true for any given $\mathit{fid}, \mathit{fid}', j, v, \widehat{s}$ and i. We prove that $L = R$ in all eight cases.

For each case C_i, $i = 1, \ldots, 8$, we formulate and prove a pair of lemmas M_i and $\boldsymbol{M}_i$ that facilitate the proof of the goal $L = R$. Specifically, for each case C_i there are two possibilities:

1. $L = R$ follows because both L and R reduce to a common term t, with $L = t$ following by virtue of lemma M_i and $R = t$ following by virtue of lemma $\boldsymbol{M}_i$:



2. The desired identity follows because L and R respectively reduce to $\mathit{read}(\mathit{fid}', i, \widehat{s})$ and $\boldsymbol{read}(\mathit{fid}', i, \alpha(\widehat{s}))$, which are equal owing to (2). In this case, M_i is used to show $L = \mathit{read}(\mathit{fid}', i, \widehat{s})$ and $\boldsymbol{M}_i$ is used to show $R = \boldsymbol{read}(\mathit{fid}', i, \alpha(\widehat{s}))$:

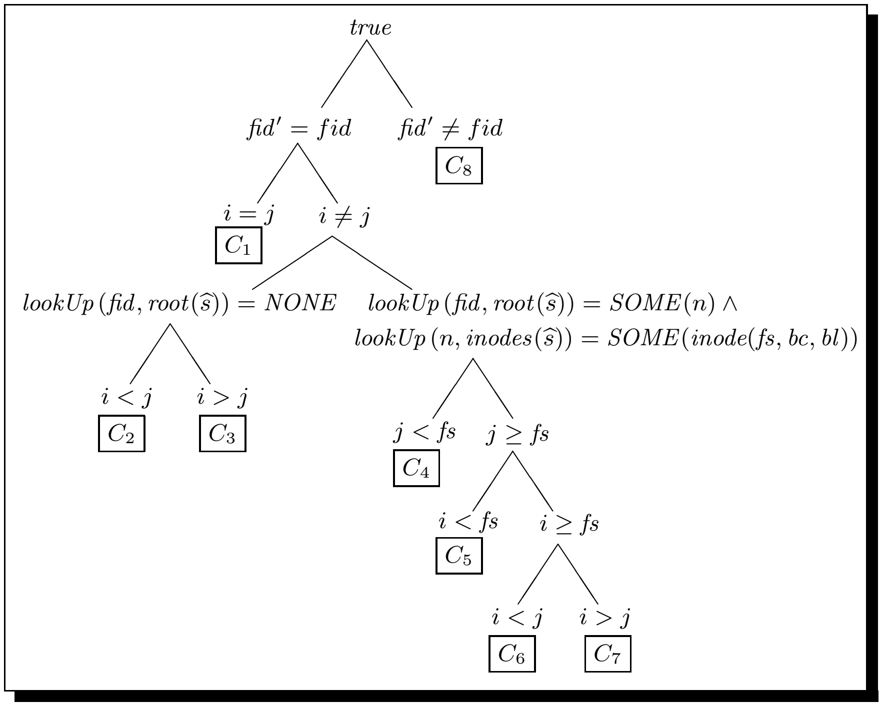
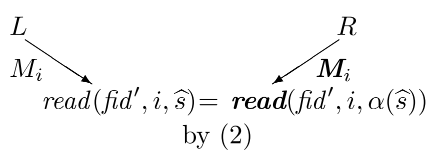


Fig. 3. Case analysis for proving the correctness of *write*.



The eight pairs of lemmas are shown in Figure 4. The "abstract-state" versions of the lemmas ($[\boldsymbol{M}_i], i = 1, \ldots, 8$) are readily proved with the aid of Vampire from the axiomatizations of maps, resizable arrays, options, natural numbers, etc., and the specification axioms. The concrete lemmas M_i are much more challenging.

6 Reachability invariants

Reachable states have a number of properties that make them "well behaved." For instance, if a file identifier is bound in the root of a state s to some inode number n, then we expect n to be bound in the mapping $inodes(s)$. While this is not true for arbitrary states s, it is true for reachable states. In what follows, by a state *invariant* we will mean a unary predicate on states $I(s)$ that is true for all reachable states, i.e., such that $\forall\, \widehat{s}\,.\, I(\widehat{s})$.

$$[M_1]\ \ read(fid, i, write(fid, i, v, \widehat{s})) = Ok(v)$$
$$[\boldsymbol{M}_1]\ \ \boldsymbol{read}(fid, i, \boldsymbol{write}(fid, i, v, \boldsymbol{s})) = Ok(v)$$

$$[M_2]\ \ [lookUp\,(fid, root(\widehat{s})) = NONE \wedge i < j] \Rightarrow read(fid, i, write(fid, j, v, \widehat{s})) = Ok(v)$$
$$[\boldsymbol{M}_2]\ \ [lookUp\,(fid, \boldsymbol{s}) = NONE \wedge i < j] \Rightarrow \boldsymbol{read}(fid, i, \boldsymbol{write}(fid, j, v, \boldsymbol{s})) = Ok(v)$$

$$[M_3]\ \ [lookUp\,(fid, root(\widehat{s})) = NONE \wedge j < i] \Rightarrow read(fid, i, write(fid, j, v, \widehat{s})) = EOF$$
$$[\boldsymbol{M}_3]\ \ [lookUp\,(fid, \boldsymbol{s}) = NONE \wedge j < i] \Rightarrow \boldsymbol{read}(fid, i, \boldsymbol{write}(fid, j, v, \boldsymbol{s})) = EOF$$

$$[M_4]\ \ \begin{array}{c}[lookUp\,(fid, root(\widehat{s})) = SOME(n) \wedge \\ lookUp\,(n, inodes(\widehat{s})) = SOME(inode(fs, bc, bl)) \wedge i \neq j \wedge j < fs] \Rightarrow \\ read(fid, i, write(fid, j, v, \widehat{s})) = read(fid, i, \widehat{s})\end{array}$$
$$[\boldsymbol{M}_4]\ \ \begin{array}{c}[lookUp\,(fid, \boldsymbol{s}) = SOME(A) \wedge i \neq j \wedge j < arrayLen(A)] \Rightarrow \\ \boldsymbol{read}(fid, i, \boldsymbol{write}(fid, j, v, \boldsymbol{s})) = \boldsymbol{read}(fid, i, \boldsymbol{s})\end{array}$$

$$[M_5]\ \ \begin{array}{c}[lookUp\,(fid, root(\widehat{s})) = SOME(n) \wedge \\ lookUp\,(n, inodes(\widehat{s})) = SOME(inode(fs, bc, bl)) \wedge fs \leq j \wedge i < fs] \Rightarrow \\ read(fid, i, write(fid, j, v, \widehat{s})) = read(fid, i, \widehat{s})\end{array}$$
$$[\boldsymbol{M}_5]\ \ \begin{array}{c}[lookUp\,(fid, \boldsymbol{s}) = SOME(A) \wedge arrayLen(A) \leq j \wedge i < arrayLen(A)] \Rightarrow \\ \boldsymbol{read}(fid, i, \boldsymbol{write}(fid, j, v, \boldsymbol{s})) = \boldsymbol{read}(fid, i, \boldsymbol{s})\end{array}$$

$$[M_6]\ \ \begin{array}{c}[lookUp\,(fid, root(\widehat{s})) = SOME(n) \wedge \\ lookUp\,(n, inodes(\widehat{s})) = SOME(inode(fs, bc, bl)) \wedge fs \leq i \wedge i < j] \Rightarrow \\ read(fid, i, write(fid, j, v, \widehat{s})) = Ok(fillByte)\end{array}$$
$$[\boldsymbol{M}_6]\ \ \begin{array}{c}[lookUp\,(fid, \boldsymbol{s}) = SOME(A) \wedge arrayLen(A) \leq j \wedge arrayLen(A) \leq i \wedge i < j] \Rightarrow \\ \boldsymbol{read}(fid, i, \boldsymbol{write}(fid, j, v, \boldsymbol{s})) = Ok(fillByte)\end{array}$$

$$[M_7]\ \ \begin{array}{c}[lookUp\,(fid, root(\widehat{s})) = SOME(n) \wedge \\ lookUp\,(n, inodes(\widehat{s})) = SOME(inode(fs, bc, bl)) \wedge fs \leq j \wedge j < i] \Rightarrow \\ read(fid, i, write(fid, j, v, \widehat{s})) = EOF\end{array}$$
$$[\boldsymbol{M}_7]\ \ \begin{array}{c}[lookUp\,(fid, \boldsymbol{s}) = SOME(A) \wedge arrayLen(A) \leq j \wedge arrayLen(A) \leq i \wedge j < i] \Rightarrow \\ \boldsymbol{read}(fid, i, \boldsymbol{write}(fid, j, v, \boldsymbol{s})) = EOF\end{array}$$

$$[M_8]\ \ fid_1 \neq fid_2 \Rightarrow read(fid_2, i, write(fid_1, j, v, \widehat{s})) = read(fid_2, i, \widehat{s})$$
$$[\boldsymbol{M}_8]\ \ fid_1 \neq fid_2 \Rightarrow \boldsymbol{read}(fid_2, i, \boldsymbol{write}(fid_1, j, v, \boldsymbol{s})) = \boldsymbol{read}(fid_2, i, \boldsymbol{s})$$

Fig. 4. Main lemmas

There are 12 invariants $inv_0, \ldots, inv_{11}$, that are of particular interest. The proof relies on them explicitly, i.e., at various points in the course of the argument we assume that all reachable states have these properties. Therefore, for the proof to be complete, we need to discharge these assumptions by proving that the properties in question are indeed invariants.

The process of guessing useful invariants—and then, more importantly, trying to prove them—was very helpful in strengthening our understanding of the implementation. More than once we conjectured false invariants, properties that appeared reasonable at first glance but later, when we tried to prove them, turned out to be false. For instance, a seemingly sensible "size invariant" is that for every inode of size *fs* and block count *bc* we have

$$fs = [(bc - 1) \cdot blockSize] + (fs \ mod \ blockSize)$$

But this equality does not hold when the file size is a multiple of the block count. The proper invariant is [2]

$$[fs \ mod \ blockSize = 0 \Rightarrow fs = bc \cdot blockSize] \wedge$$
$$[fs \ mod \ blockSize \neq 0 \Rightarrow fs = ((bc - 1) \cdot blockSize) + (fs \ mod \ blockSize)]$$

where *div* denotes integer division. For any inode of file size *fs* and block count *bc*, we will write $szInv(fs, bc)$ to indicate that *fs* and *bc* are related as shown by the above formula.

Figure 5 presents the twelve reachability invariants for our file system implementation. In the sequel we focus on the first four invariants, inv_0,inv_1,inv_2,inv_3. These four invariants are fundamental and must be established before anything non-trivial can be proven about the system. They are also co-dependent, meaning that in order to prove that an operation preserves one of them, say inv_j, we often need to assume that the incoming state not only has inv_j but also one or more of the other three invariants. For instance, we cannot prove that *write* preserves inv_3, i.e., that

$$\forall \ i \ v \ s \, . \, inv_3(s) \Rightarrow inv_3(write(fid, i, v, s))$$

unless we also assume that s has inv_0. Or suppose we want to prove that *writeExisting* preserves any of the four invariants, say inv_0, so that our goal is to show $inv_0(writeExisting(n, i, v, s))$ on the assumptions

$$lookUp\,(n, inodes(s)) = SOME(inode(fs, bc, bl)) \tag{4}$$

and

$$inv_0(s) \tag{5}$$

Consider the case

$$bc \leq i \ div \ blockSize,$$

whereby $writeExisting(n, i, v, s)$ returns

$$writeNoExtend(n, i, v, extendFile(n, i, s)).$$

Since *writeNoExtend* is conditionally defined, we need to show that its three preconditions are satisfied in the intermediate state $s_1 = extendFile(n, i, s)$. It is easy enough to show that the first precondition holds, i.e., that

$$lookUp\,(n, inodes(s_1)) = SOME(inode(fs_1, bc_1, bl_1))$$

for some fs_1, bc_1, and bl_1; this follows from (4) and an auxiliary lemma stating that *extendFile* preserves the invariant $I(s) \equiv inDom(n, inodes(s))$ (for

[2] This invariant is equivalent to $bc = (fs + blockSize - 1) \ div \ blockSize$.

$$inv_0\,(s) : [lookUp\,(fid, root(s)) = SOME(n)] \Rightarrow inDom(n, inodes(s))$$

$$\begin{aligned} inv_1\,(s) : \; & [lookUp\,(n, inodes(s)) = SOME(inode(fs, bc, bl))] \Rightarrow \\ & [inDom(k, bl) \Leftrightarrow k < bc] \end{aligned}$$

$$\begin{aligned} inv_2\,(s) : \; & [\, lookUp\,(n, inodes(s)) = SOME(inode) \wedge \\ & lookUp\,(bn, blockList(inode)) = SOME(bn')] \Rightarrow \\ & inDom(bn', blocks(s)) \end{aligned}$$

$$inv_3\,(s) : [lookUp\,(n, inodes(s)) = SOME(inode(fs, bc, bl))] \Rightarrow szInv(fs, bc)$$

$$inv_4\,(s) : inDom(bnum, blocks(s)) \Leftrightarrow bnum < stateBlockCount(s)$$

$$inv_5\,(s) : inDom(nodeNum, inodes(s)) \Leftrightarrow nodeNum < inodeCount(s)$$

$$\begin{aligned} inv_6\,(s) : \; & [lookUp\,(nodeNum, inodes(s)) = SOME(inode(fs, bc, bl)) \wedge bc = 0] \\ & \quad \Rightarrow fs = 0 \end{aligned}$$

$$\begin{aligned} inv_7\,(s) : \; & [fid_1 \neq fid_2 \wedge \\ & lookUp\,(fid_1, root(s)) = SOME(nodeNum_1) \wedge \\ & lookUp\,(fid_2, root(s)) = SOME(nodeNum_2)] \\ & \quad \Rightarrow nodeNum_1 \neq nodeNum_2 \end{aligned}$$

$$\begin{aligned} inv_8\,(s) : \; & [lookUp\,(nodeNum, inodes(s)) = SOME(node) \wedge \\ & lookUp\,(k, blockList(node)) = SOME(bnum) \wedge \\ & lookUp\,(bnum, blocks(s)) = SOME(block)] \Rightarrow \\ & \quad (inDom(j, block) \Leftrightarrow j < blockSize) \end{aligned}$$

$$\begin{aligned} inv_9\,(s) : \; & [lookUp\,(nodeNum_1, inodes(s)) = SOME(node_1) \wedge \\ & lookUp\,(nodeNum_2, inodes(s)) = SOME(node_2) \wedge \\ & lookUp\,(k_1, blockList(node_1)) = SOME(bnum_1) \wedge \\ & lookUp\,(k_2, blockList(node_2)) = SOME(bnum_2) \wedge \\ & nodeNum_1 \neq nodeNum_2] \\ & \quad \Rightarrow bnum_1 \neq bnum_2 \end{aligned}$$

$$\begin{aligned} inv_{10}(s) : \; & [lookUp\,(nodeNum, inodes(s)) = SOME(node) \wedge \\ & lookUp\,(k_1, blockList(node)) = SOME(bnum_1) \wedge \\ & lookUp\,(k_2, blockList(node)) = SOME(bnum_2) \wedge \\ & k_1 \neq k_2] \\ & \quad \Rightarrow bnum_1 \neq bnum_2 \end{aligned}$$

$$\begin{aligned} inv_{11}(s) : \; & [lookUp\,(nodeNum, inodes(s)) = SOME(inode(fs, bc, bl)) \wedge \\ & i \;div\; blockSize < bc \;\wedge\; fs \leq i \wedge \\ & lookUp\,(i \;div\; blockSize, bl) = SOME(bnum) \wedge \\ & lookUp\,(bnum, blocks(s)) = SOME(block)] \Rightarrow \\ & lookUp\,(i \;mod\; blockSize, block) = SOME(fillByte) \end{aligned}$$

Fig. 5. Reachability Invariants

fixed inode number n). However, it is more challenging to show that the two remaining preconditions hold, i.e., that there exist bn_1 and $block_1$ such that $lookUp\,(i \ div \ blockSize, bl_1) = SOME(bn_1)$ and

$$lookUp\,(bn_1, blocks(s_1)) = SOME(block_1).$$

But these would follow immediately if we could show that s_1 has inv_1 and inv_2 and that $i \ div \ blockSize < bc_1$. Showing that s_1 has inv_1 and inv_2 would also follow immediately if we strengthened our initial hypothesis (5) by additionally assuming that s has inv_1 and inv_2, provided we have shown elsewhere that *extendFile* preserves both of these invariants. However, showing $i \ div \ blockSize < bc_1$ presupposes that s_1 has inv_3. Consequently, we are led to assume that the original state s has all four invariants. Provided we have already shown that *extendFile* preserves each of the four invariants, it then follows that s_1 has all four of them, and hence that the preconditions of *writeNoExtend* hold.

6.1 Proving invariants

Showing that a unary state property $I(s)$ is an invariant proceeds in two steps:

1. proving that I holds for the initial state, $I(s_0)$; and
2. proving $\forall \, fid \ i \ v \ s \,.\, I(s) \Rightarrow I(write(fid, i, v, s))$.

Once both of these have been established, a routine induction on n will show that

$$\forall \, n \ s \,.\, reachableN(s, n) \Rightarrow I(s).$$

It then follows directly by the definition of reachability that all reachable states have I.

Proving that the initial state has an invariant inv_j is straightforward: in all twelve cases it is done automatically. The second step, proving that *write* preserves inv_j, is more involved. Including *write*, the implementation comprises ten state-transforming operations,[3] and control may flow from *write* to any one of them. Accordingly, we need to show that all ten operations preserve the invariant under consideration. This means that for a total of ten operations $f_0, \ldots, f_9$ and twelve invariants $inv_0, \ldots, inv_{11}$, we need to prove 120 lemmas, each stating that f_i preserves inv_j.

Most of the operations f_i are defined conditionally, in the form

$$\forall \, \boldsymbol{x}_i \ \boldsymbol{y}_i \,.\, PC_i(\boldsymbol{x}_i, \boldsymbol{y}_i) \Rightarrow f_i(\boldsymbol{x}_i) = \cdots$$

where $\boldsymbol{x}_i$, $\boldsymbol{y}_i$ are lists of distinct variables; $PC_i(\boldsymbol{x}_i, \boldsymbol{y}_i)$, the "precondition" of f_i, is usually a conjunction of equations in the variables $\boldsymbol{x}_i$ and $\boldsymbol{y}_i$ (if f_i is not defined conditionally then this can be regarded as the empty conjunction, i.e.,

[3] By a "state-transforming operation" we mean one that takes a state as an argument and produces a state as output. There are ten such operations, nine of which are auxiliary functions (such as *extendFile*) invoked by *write*.

as the constant *true*). Therefore, each of the 120 invariant-preservation lemmas is of the form

$$\forall\, \boldsymbol{x}_i\ \boldsymbol{y}_i\ s\,.\, [PC_i(\boldsymbol{x}_i, \boldsymbol{y}_i) \land I(s)] \Rightarrow inv_j(f_i(\boldsymbol{x}_i)) \tag{6}$$

for $i = 0, \ldots, 9$ and $j = 0, \ldots, 11$, and where $I(s)$ is of the form $inv_j(s) \land inv_{i_1} \land \cdots \land inv_{i_k}$ where $k \geq 0$ and $i_r \in \{0, 1, \ldots, 11\}$ for $1 \leq r \leq k$.

The large majority of the proof text (about 80% of it) is devoted to proving these lemmas. Some of them are surprisingly tricky to prove, and even those that are not particularly conceptually demanding can be challenging to manipulate, if for no other reason simply because of their volume. Given the size of the function preconditions and the size of the invariants (especially in those cases where we need to consider the conjunction of several invariants at once), an invariance lemma can span multiple pages of text. Proof goals of that scale test the limits even of cutting-edge ATPs. For instance, in the case of a proposition P that was several pages long (which arose in the proof of one of the invariance lemmas), Spass took over 10 minutes to prove the trivial goal $P \Rightarrow P'$, where P' was simply an alphabetically renamed copy of P (Vampire was not able to prove it at all, at least within 20 minutes). Heavily skolemizing the formula and blindly following the resolution procedure prevented these systems from recognizing the goal as trivial. By contrast, using Athena's native inference rules, the goal was derived instantaneously via the two-line deduction **assume** P **in claim** P', because Athena treats alphabetically equivalent propositions as identical and has an efficient implementation of proposition look-ups. This speaks to the need to have a variety of reasoning mechanisms available in a uniform, integrated framework.

There are many additional lemmas that were used in proving the invariants or in proving other results after all twelve invariants had already been proven. We mention two typical ones:

Lemma 1. *If* $fid_1 \neq fid_2$ *and*

$$lookUp\,(fid_2, root(\widehat{s})) = x$$

then $lookUp\,(fid_2, root(write(fid_1, i, v, \widehat{s}))) = x$.

Lemma 2. *If* $lookUp\,(n, inodes(s)) = SOME(inode_1)$ *and*

$$lookUp\,(n, inodes(allocBlocks(n, k, j, s))) = SOME(inode_2)$$

then $blockCount(inode_2) = blockCount(inode_1) + k$.

7 Proof automation with tactics

After proving a few invariance lemmas for some of the operations it became apparent that a large portion of the reasoning was the same in every case and could thus be factored away for reuse.

Athena makes it easy to abstract concrete proofs into natural-deduction proof algorithms called *methods*. For every state-transforming operation f_i we wrote a "preserver" method P_i that takes an arbitrary invariant I as input (expressed as a unary function that takes a state and constructs an appropriate proposition) and attempts to prove the corresponding invariance lemma.

$$\forall\, \boldsymbol{x}_i\ \boldsymbol{y}_i\ s\,.\,[PC_i(\boldsymbol{x}_i, \boldsymbol{y}_i) \wedge I(s)] \Rightarrow I(f_i(\boldsymbol{x}_i)) \tag{7}$$

P_i encapsulates all the generic reasoning involved in proving invariants for f_i. If any non-generic reasoning (specific to I) is additionally required, it is packaged into a proof continuation K and passed into P_i as a higher-order method argument. P_i can then invoke K at appropriate points within its body as needed. Similar methods for other functions made the overall proof substantially shorter—and easier to develop and to debug—than it would have been otherwise.

Consider, for example, proving that *allocBlocks* preserves a certain property I. This is always done by induction on k, the number of blocks to be allocated. Performing the base inductive step automatically, managing the inductive hypothesis, proving that the relevant precondition involving *getNextBlock* is satisfied in the context in which *allocBlocks* is called, deriving useful consequences of that fact, etc., these are all standard tasks that are repetitively performed regardless of I; we have abstracted all of them away in a higher-order method that accepts the I-specific reasoning as an input method.

Proof programmability was useful in streamlining several other recurring patterns of reasoning, apart from dealing with invariants. A typical example is this: given a reachable state $\widehat{s}$, an inode number n such that $\mathit{lookUp}\,(n, \mathit{inodes}(\widehat{s})) = \mathit{SOME}(\mathit{inode}(\mathit{fs}, \mathit{bc}, \mathit{bl}))$, and an index $i < \mathit{fs}$, we often need to prove the existence of bn and block such that $\mathit{lookUp}\,(i\ \mathit{div}\ \mathit{blockSize}, \mathit{bl}) = \mathit{SOME}(\mathit{bn})$ and

$$\mathit{lookUp}\,(\mathit{bn}, \mathit{blocks}(\widehat{s})) = \mathit{SOME}(\mathit{block})$$

The reasoning runs as follows: first, from the reachability of $\widehat{s}$, we infer that it has certain invariants, including inv_0, inv_1, inv_2, and inv_3. From these invariants, the assumption $i < \mathit{fs}$, and standard arithmetic laws we may deduce $(i\ \mathit{div}\ \mathit{blockSize}) < \mathit{bc}$. From this, our initial assumptions, and inv_1, we conclude that $i\ \mathit{div}\ \mathit{blockSize}$ is in the domain of the mapping bl. Thus the existence of an appropriate bn is ensured, and along with it, owing to inv_2, the existence of an appropriate block. We packaged this reasoning in a method *find-bn-block* that takes all the relevant quantities as inputs, assumes that the appropriate hypotheses are in the assumption base, and performs the appropriate inferences. The method also accepts a proof continuation K that is invoked once the goal has been successfully derived.

Another example is a slight extension of this method, named *find-bn-block-val*, that operates under the same assumptions but, in addition to a block number and the block itself, yields a value v such that $\mathit{lookUp}\,(i\ \mathit{mod}\ \mathit{blockSize}, \mathit{block}) = \mathit{SOME}(v)$, which is possible because $i\ \mathit{mod}\ \mathit{blockSize} < \mathit{blockSize}$. Yet another example of a streamlined proof method is an inductive method showing that an invariant holds for all reachable states.

8 A sample lemma proof

In this section we will prove lemma $[M_8]$, which can be viewed as a frame condition: it asserts that performing a *write* operation on a given file leaves the contents of every other file unchanged. More specifically, let fid_1 refer to the file to be written, let fid_2 be any file identifier distinct from fid_1, let s be any reachable state, and let s' be the state obtained from s by writing some value into some byte position of fid_1. Then $[M_8]$ says that reading any byte of fid_2 in s' yields the same result as reading that byte in s.

The proof relies on four auxiliary lemmas about *write*, given below. Lemmas (8) and (9) handle the case when fid_1 (the file to be written) already exists in s, while (10) and (11) apply to the case when fid_1 is unbound in the root of s. As usual, all the variables are assumed to be universally quantified.

$$\begin{array}{c}[lookUp\,(n_1, inodes(s)) = SOME(inode_1) \wedge n \neq n_1 \wedge \\ lookUp\,(bn, blockList(inode_1)) = SOME(bn') \wedge \\ lookUp\,(bn', blocks(s)) = SOME(block_1) \wedge lookUp\,(fid, root(s)) = SOME(n)] \Rightarrow \\ lookUp\,(n_1, inodes(write(fid, i, v, s))) = SOME(inode_1) \wedge \\ lookUp\,(bn', blocks(write(fid, i, v, s))) = SOME(block_1)\end{array} \tag{8}$$

$$\begin{array}{c}[lookUp\,(n_1, inodes(s)) = SOME(inode_1) \wedge n \neq n_1 \wedge \\ lookUp\,(fid, root(s)) = SOME(n)] \Rightarrow \\ lookUp\,(n_1, inodes(write(fid, i, v, s))) = SOME(inode_1)\end{array} \tag{9}$$

$$\begin{array}{c}[lookUp\,(n_1, inodes(s)) = SOME(inode_1) \wedge \\ lookUp\,(bn, blockList(inode_1)) = SOME(bn') \wedge \\ lookUp\,(bn', blocks(s)) = SOME(block_1) \wedge lookUp\,(fid, root(s)) = NONE] \Rightarrow \\ lookUp\,(n_1, inodes(write(fid, i, v, s))) = SOME(inode_1) \wedge \\ lookUp\,(bn', blocks(write(fid, i, v, s))) = SOME(block_1)\end{array} \tag{10}$$

$$\begin{array}{c}[lookUp\,(n_1, inodes(\widehat{s})) = SOME(inode_1) \wedge lookUp\,(fid, root(\widehat{s})) = NONE] \Rightarrow \\ lookUp\,(n_1, inodes(write(fid, i, v, s))) = SOME(inode_1)\end{array} \tag{11}$$

In turn, each of the above four lemmas about *write* relies on a number of other lemmas about the various operations in the call graph of *write* (see the relevant remarks in Section 4). We will state those lemmas after we present the proof of $[M_8]$.

We next present a natural-deduction style proof of $[M_8]$ to give the reader an idea of the abstraction level at which Athena proofs are written. We believe that the said level is roughly equivalent to the level at which a formally trained computer scientist would communicate the proof to another computer scientist of a similar background. The proof is rigorous and thorough, but does not descend to the level of primitive inference rules (such as introduction and elimination rules for the logical connectives or congruence rules for equality); the applications of such rules are fairly tedious steps that are filled in by Vampire. The overall proof is guided by constructs such as "pick any $\cdots$", "assume that such and such holds", "we distinguish two cases", "from P_1, P_2 and P_3 we infer P", and so on.

The proof of $[M_8]$ is given below in English, but the level of detail and the overall structure of the argument are isomorphic to those of the formal Athena deduction (for instance, the formal Athena proof runs to 120 lines, whereas the English proof below is about 64 lines).

Lemma 3 ([M_8])**.** *If* $\mathit{fid}_1 \neq \mathit{fid}_2$ *then* $\mathit{read}(\mathit{fid}_2, i, \mathit{write}(\mathit{fid}_1, j, v, \widehat{s})) = \mathit{read}(\mathit{fid}_2, i, \widehat{s})$.

Proof. Pick arbitrary $\mathit{fid}_1, \mathit{fid}_2, i, j, v$, and $\widehat{s}$, and suppose that

$$\mathit{fid}_1 \neq \mathit{fid}_2. \tag{12}$$

We will prove the goal

$$\mathit{read}(\mathit{fid}_2, i, \mathit{write}(\mathit{fid}_1, j, v, \widehat{s})) = \mathit{read}(\mathit{fid}_2, i, \widehat{s}) \tag{13}$$

by distinguishing two (mutually exclusive and jointly exhaustive) cases:

$$\mathit{lookUp}\,(\mathit{fid}_2, \mathit{root}(\widehat{s})) = \mathit{NONE} \tag{14}$$

and

$$\exists\, n_2 \,.\, \mathit{lookUp}\,(\mathit{fid}_2, \mathit{root}(\widehat{s})) = \mathit{SOME}(n_2). \tag{15}$$

If fid_2 is unbound in $\mathit{root}(\widehat{s})$ (case (14)), then, by the definition of *read*, we have

$$\mathit{read}(\mathit{fid}_2, i, \widehat{s}) = \mathit{FileNotFound}. \tag{16}$$

By Lemma 1, (12), (14), and the reachability of $\widehat{s}$ we conclude

$$\mathit{lookUp}\,(\mathit{fid}_2, \mathit{root}(\mathit{write}(\mathit{fid}_1, j, v, \widehat{s}))) = \mathit{NONE} \tag{17}$$

and therefore again by the definition of *read* we infer

$$\mathit{read}(\mathit{fid}_2, i, \mathit{write}(\mathit{fid}_1, j, v, \widehat{s})) = \mathit{FileNotFound} \tag{18}$$

and hence (13) follows from (16) and (18). We now consider case (15), whereby

$$\mathit{lookUp}\,(\mathit{fid}_2, \mathit{root}(\widehat{s})) = \mathit{SOME}(n_2) \tag{19}$$

for some inode number n_2. Since $\widehat{s}$ is reachable, it has inv_0, so that

$$\mathit{lookUp}\,(n_2, \mathit{inodes}(\widehat{s})) = \mathit{SOME}(\mathit{inode}(\mathit{fs}_2, \mathit{bc}_2, \mathit{bl}_2)) \tag{20}$$

for some fs_2, bc_2, and bl_2. Moreover, we note that by Lemma 1, (19), (12), and the reachability of $\widehat{s}$, we have

$$\mathit{lookUp}\,(\mathit{fid}_2, \mathit{root}(\mathit{write}(\mathit{fid}_1, j, v, \widehat{s}))) = \mathit{SOME}(n_2). \tag{21}$$

We proceed by distinguishing two cases, $i < \mathit{fs}_2$ and $\mathit{fs}_2 \leq i$. Suppose first that $i < \mathit{fs}_2$. In that case it becomes evident by inspection that all the preconditions of method *find-bn-block-val* are satisfied: $\widehat{s}$ has the required invariants because it is reachable; n_2 is mapped by the inode mapping of $\widehat{s}$ to the inode comprising fs_2, bc_2, and bl_2; and $i < fs_2$. Therefore, we are able to prove that there exist $bn_2, block_2$, and v_2 such that

$$lookUp\,(i\ div\ blockSize, bl_2) = SOME(bn_2) \tag{22}$$

$$lookUp\,(bn_2, blocks(\widehat{s})) = SOME(block_2) \tag{23}$$

and

$$lookUp\,(i\ mod\ blockSize, block_2) = SOME(v_2). \tag{24}$$

It now follows from (19), (20), the assumption $i < fs_2$, (22), (23), (24), and the definition of *read* that

$$read(fid_2, i, \widehat{s}) = Ok(v_2) \tag{25}$$

and therefore our goal (13) becomes reduced to proving

$$read(fid_2, i, write(fid_1, j, v, \widehat{s})) = Ok(v_2). \tag{26}$$

We establish (26) by considering two subcases. First, suppose that fid_1 is unbound in the root of $\widehat{s}$, i.e.,

$$lookUp\,(fid_1, root(\widehat{s})) = NONE. \tag{27}$$

Then by (27), (20), (22), (23), the reachability of $\widehat{s}$ and Lemma (10), we conclude

$$\begin{aligned} lookUp\,(n_2, inodes(write(fid_1, j, v, \widehat{s}))) = \\ SOME(inode(fs_2, bc_2, bl_2)) \end{aligned} \tag{28}$$

and

$$lookUp\,(bn_2, blocks(write(fid_1, j, v, \widehat{s}))) = SOME(block_2). \tag{29}$$

Accordingly, by the definition of *read*, (21), (28), the assumption $i < fs_2$, (22), (29), and (24), we obtain the desired (26).

Now suppose, by contrast, that

$$lookUp\,(fid_1, root(\widehat{s})) = SOME(n_1) \tag{30}$$

for some inode number n_1. Since $\widehat{s}$ is reachable, it has the invariant inv_7, so from (30), (19), and (12) we conclude

$$n_1 \neq n_2. \tag{31}$$

From (8), the reachability of $\widehat{s}$, (20), (31), (22), (23), and (30) we can now again derive (28) and (29). Hence, by the definition of *read*, (21), (28), the assumption $i < fs_2$, (22), (29), and (24) we obtain (26).

We finally consider the possibility $fs_2 \leq i$. In that case the definition of *read* in tandem with (19) and (20) entails

$$read(fid_2, i, \widehat{s}) = EOF. \tag{32}$$

As before, we again distinguish two subcases, according to whether or not fid_1 is bound in the root of $\widehat{s}$, and we use lemmas (9) and (11), respectively, to infer

(28). In combination with (21), it follows from the definition of *read* that in either case we have

$$read(fid_2, i, write(fid_2, j, v, \widehat{s})) = EOF \tag{33}$$

and the desired equality now follows from (32) and (33). This completes our case analysis and the proof. □

Finally, we list below the remaining lemmas needed for lemmas (8), (9), (10), and (11).

$$\begin{array}{l}
\mathit{writeSmallExtendPreservesINodeAndBlockMaps}: \\
\quad [\mathit{lookUp}\,(n_1, \mathit{inodes}(s)) = \mathit{SOME}(\mathit{inode}_1) \wedge n \neq n_1 \wedge \\
\quad \mathit{lookUp}\,(k, \mathit{blockList}(\mathit{inode}_1)) = \mathit{SOME}(\mathit{bn}_1) \wedge \\
\quad \mathit{lookUp}\,(\mathit{bn}_1, \mathit{blocks}(s)) = \mathit{SOME}(\mathit{block}_1) \wedge \mathit{inv}_{10}(s) \\
\quad \mathit{lookUp}\,(n, \mathit{inodes}(s)) = \mathit{SOME}(\mathit{inode}(\mathit{fs}, \mathit{bc}, \mathit{bl})) \wedge \\
\quad \mathit{lookUp}\,(i \ \mathit{div} \ \mathit{blockSize}, \mathit{bl}) = \mathit{SOME}(\mathit{bn}) \wedge \\
\quad \mathit{lookUp}\,(\mathit{bn}, \mathit{blocks}(s)) = \mathit{SOME}(\mathit{block}) \wedge \mathit{fs} \leq i] \Rightarrow \\
\quad \mathit{lookUp}\,(n_1, \mathit{inodes}(\mathit{writeSmallExtend}(n, i, v, s))) = \mathit{SOME}(\mathit{inode}_1) \wedge \\
\quad \mathit{lookUp}\,(\mathit{bn}_1, \mathit{blocks}(\mathit{writeSmallExtend}(n, i, v, s))) = \mathit{SOME}(\mathit{block}_1)
\end{array}$$

$$\begin{array}{l}
\mathit{writeSmallExtendPreservesINodeMap}: \\
\quad [\mathit{lookUp}\,(n_1, \mathit{inodes}(s)) = \mathit{SOME}(\mathit{inode}_1) \wedge n \neq n_1 \wedge \\
\quad \mathit{lookUp}\,(n, \mathit{inodes}(s)) = \mathit{SOME}(\mathit{inode}(\mathit{fs}, \mathit{bc}, \mathit{bl})) \wedge \\
\quad \mathit{lookUp}\,(i \ \mathit{div} \ \mathit{blockSize}, \mathit{bl}) = \mathit{SOME}(\mathit{bn}) \wedge \\
\quad \mathit{lookUp}\,(\mathit{bn}, \mathit{blocks}(s)) = \mathit{SOME}(\mathit{block}) \wedge \mathit{fs} \leq i] \Rightarrow \\
\quad \mathit{lookUp}\,(n_1, \mathit{inodes}(\mathit{writeSmallExtend}(n, i, v, s))) = \mathit{SOME}(\mathit{inode}_1)
\end{array}$$

$$\begin{array}{l}
\mathit{writeNoExtendPreservesINodeAndBlockMaps}: \\
\quad [\mathit{lookUp}\,(n_1, \mathit{inodes}(s)) = \mathit{SOME}(\mathit{inode}_1) \wedge n \neq n_1 \wedge \\
\quad \mathit{lookUp}\,(k, \mathit{blockList}(\mathit{inode}_1)) = \mathit{SOME}(\mathit{bn}_1) \wedge \\
\quad \mathit{lookUp}\,(\mathit{bn}_1, \mathit{blocks}(s)) = \mathit{SOME}(\mathit{block}_1) \wedge \mathit{inv}_{10}(s) \\
\quad \mathit{lookUp}\,(n, \mathit{inodes}(s)) = \mathit{SOME}(\mathit{inode}) \wedge \\
\quad \mathit{lookUp}\,(i \ \mathit{div} \ \mathit{blockSize}, \mathit{blockList}(\mathit{inode})) = \mathit{SOME}(\mathit{bn}) \wedge \\
\quad \mathit{lookUp}\,(\mathit{bn}, \mathit{blocks}(s)) = \mathit{SOME}(\mathit{block})] \Rightarrow \\
\quad \mathit{lookUp}\,(n_1, \mathit{inodes}(\mathit{writeNoExtend}(n, i, v, s))) = \mathit{SOME}(\mathit{inode}_1) \wedge \\
\quad \mathit{lookUp}\,(\mathit{bn}_1, \mathit{blocks}(\mathit{writeNoExtend}(n, i, v, s))) = \mathit{SOME}(\mathit{block}_1)
\end{array}$$

$$\begin{array}{l}
\mathit{writeNoExtendPreservesINodeMap}: \\
\quad [\mathit{lookUp}\,(n_1, \mathit{inodes}(s)) = \mathit{SOME}(\mathit{inode}_1) \wedge n \neq n_1 \wedge \\
\quad \mathit{lookUp}\,(n, \mathit{inodes}(s)) = \mathit{SOME}(\mathit{inode}) \wedge \\
\quad \mathit{lookUp}\,(i \ \mathit{div} \ \mathit{blockSize}, \mathit{blockList}(\mathit{inode})) = \mathit{SOME}(\mathit{bn}) \wedge \\
\quad \mathit{lookUp}\,(\mathit{bn}, \mathit{blocks}(s)) = \mathit{SOME}(\mathit{block})] \Rightarrow \\
\quad \mathit{lookUp}\,(n_1, \mathit{inodes}(\mathit{writeNoExtend}(n, i, v, s))) = \mathit{SOME}(\mathit{inode}_1)
\end{array}$$

allocBlocksPreservesINodeAndBlockMaps:

$$[lookUp\,(n_1, inodes(s)) = SOME(inode_1) \wedge n \neq n_1 \wedge \\ inDom(n, inodes(s)) \wedge inv_4(s) \wedge \\ lookUp\,(bn, blockList(inode_1)) = SOME(bn') \wedge \\ lookUp\,(bn', blocks(s)) = SOME(block)] \Rightarrow \\ lookUp\,(n_1, inodes(allocBlocks(n, k, fs, s))) = SOME(inode_1) \wedge \\ lookUp\,(bn', blocks(allocBlocks(n, k, fs, s))) = SOME(block)$$

allocBlocksPreservesINodeMap:

$$[lookUp\,(n_1, inodes(s)) = SOME(inode_1) \wedge n \neq n_1 \wedge \\ inDom(n, inodes(s))] \Rightarrow \\ lookUp\,(n_1, inodes(allocBlocks(n, k, fs, s))) = SOME(inode_1)$$

extendFilePreservesINodeAndBlockMaps:

$$[lookUp\,(n_1, inodes(s)) = SOME(inode_1) \wedge n \neq n_1 \wedge \\ inDom(n, inodes(s)) \wedge inv_4(s) \wedge \\ lookUp\,(bn, blockList(inode_1)) = SOME(bn') \wedge \\ lookUp\,(bn', blocks(s)) = SOME(block)] \Rightarrow \\ lookUp\,(n_1, inodes(extendFile(n, i, s))) = SOME(inode_1) \wedge \\ lookUp\,(bn', blocks(extendFile(n, i, s))) = SOME(block)$$

extendFilePreservesINodeMap:

$$[lookUp\,(n_1, inodes(s)) = SOME(inode_1) \wedge n \neq n_1 \wedge \\ inDom(n, inodes(s))] \Rightarrow \\ lookUp\,(n_1, inodes(extendFile(n, i, s))) = SOME(inode_1)$$

writeExistingPreservesINodeAndBlockMaps:

$$[inv_1(s) \wedge inv_2(s) \wedge inv_3(s) \wedge inv_4(s) \wedge inv_{10}(s) \\ lookUp\,(n_1, inodes(s)) = SOME(inode_1) \wedge \\ n \neq n_1 \wedge lookUp\,(bn, blockList(inode_1)) = SOME(bn') \wedge \\ lookUp\,(bn', blocks(s)) = SOME(block_1) \wedge \\ lookUp\,(n, inodes(s)) = SOME(inode(fs, bc, bl))] \Rightarrow \\ lookUp\,(n_1, inodes(writeExisting(n, i, v, s))) = SOME(inode_1) \wedge \\ lookUp\,(bn', blocks(writeExisting(n, i, v, s))) = SOME(block_1)$$

writeExistingPreservesINodeMap:

$$[inv_1(s) \wedge inv_2(s) \wedge inv_3(s) \wedge n \neq n_1 \\ lookUp\,(n_1, inodes(s)) = SOME(inode_1) \\ lookUp\,(n, inodes(s)) = SOME(inode(fs, bc, bl))] \Rightarrow \\ lookUp\,(n_1, inodes(writeExisting(n, i, v, s))) = SOME(inode_1)$$

9 Related work

Techniques for verifying the correct use of file system interfaces expressed as finite state machines are presented in [9, 10, 8, 2]. In this paper we have addressed the more difficult problem of showing that the file system implementation conforms to its specification. Consequently, our proof obligations are stronger and we have resorted to more general deductive verification. Static analysis techniques that handle more complex data structures include predicate abstraction and shape analysis [19, 18, 14, 6]. These approaches are promising for automating proofs of program properties, but have not been used so far to show full functional correctness, as we do here. Security properties of a Unix file system are studied in [23, Chapter 10]; these properties are orthogonal to the correct functioning of a file system for storing and reading data. A sample specification of a widely used file system is [1]. Simple abstract models of file systems have also been developed in Z [24, Chapter 15].

Alloy [12] is a specification language based on a first-order relational calculus that has been used to describe the directory structure of a file system (but without modelling read and write operations). The Alloy Analyzer is a model finder for Alloy specifications that can be used to check structural properties of file systems in finite scope. The use of Alloy is complementary to proofs [4]. Alloy is useful for debugging, whereas our proofs ensure that the refinement relation holds for any number of files, any file sizes, and all sequences of operations. In addition, readable, high-level proofs can be viewed as explanations of *why* the file system implementation is correct, and therefore provide guidance to developers on how to modify the system in the future while preserving its correctness.

It is interesting to consider whether the verification burden would be lighter with a system such as PVS [17] or ACL2 [13] that makes heavy use of automatic decision procedures for combinations of first-order theories such as arrays, lists, linear arithmetic, etc. We note that our use of high-performance off-the-shelf ATPs already provides a considerable degree of automation. In our experience, both Vampire and Spass have proven quite effective in non-inductive reasoning about lists, arrays, etc., simply on the basis of first-order axiomatizations of the these domains. Our experience supports a recent benchmark study by Armando et al. [5], which showed that a state-of-the-art paramodulation-based prover with a fair search strategy compares favorably with CVC [7] in reasoning about arrays with extensionality.

10 Conclusions

We have presented a correctness proof for the key operations (reading and writing) of a file system based on Unix implementations. We are not aware of any other file system verification attempts dealing with such strong properties as the simulation relation condition, for all possible sequences of file system operations and without a priori bounds on the number of files or their sizes. Despite the apparent simplicity of this particular specification and implementation, our proofs shed light on the general kinds of reasoning that would be required in establishing full functional correctness for any file system. Our results suggest that a combination of state-of-the art formal methods techniques greatly facilitates the deductive verification of crucial software infrastructure components such as file systems.

We have found Athena to be a powerful framework for carrying out a complex verification effort. Polymorphic sorts and structures allow for natural data modelling; strong support for structural induction facilitates inductive reasoning over such datatypes; a block-structured natural deduction format helps to make proofs more readable and writable; a higher-order functional metalanguage and assumption base semantics allow for powerful trusted proof tactics; and the use of first-order logic allows for smooth integration with state-of-the-art first-order ATPs, keeping the proof steps at a high level of detail. Our use of these features was essential in dealing with the strong properties arising from the simulation relation condition, where most of the complexity stems from the details of unbounded data structures.

Acknowledgements. We thank Darko Marinov and Alexandru Salcianu for useful comments on an earlier version of this manuscript.

References

1. NFS: Network file system version 3 protocol specification. Technical report, SUN microsystems, 1994.
2. Glenn Ammons, Rastislav Bodik, and James R. Larus. Mining specifications. In *Proc. 29th ACM POPL*, 2002.
3. K. Arkoudas. Denotational Proof Languages. PhD dissertation, MIT, 2000.
4. Konstantine Arkoudas, Sarfraz Khurshid, Darko Marinov, and Martin Rinard. Integrating model checking and theorem proving for relational reasoning. In *7th International Seminar on Relational Methods in Computer Science (RelMiCS 2003)*, 2003.
5. Alessandro Armando, Maria Paola Bonacina, Silvio Ranise, Michaël Rusinowitch, and Aditya Kumar Sehgal. High-performance deduction for verification: a case study in the theory of arrays. In Serge Autexier and Heiko Mantel, editors, *Notes of the Workshop on Verification, Third Federated Logic Conference (FLoC02)*, pages 103–112, 2002.
6. Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. ACM PLDI*, 2001.
7. Clark W. Barrett, David L. Dill, and Aaron Stump. A framework for cooperating decision procedures. In D. A. McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction (Pittsburgh, PA)*, volume 1831, pages 79–98. Springer, 2000.
8. Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proc. ACM PLDI*, 2002.
9. Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. ACM PLDI*, 2001.
10. Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proc. ACM PLDI*, 2002.
11. M. J. C. Gordon and T. F. Melham. *Introduction to HOL, a theorem proving environment for higher-order logic*. Cambridge University Press, Cambridge, England, 1993.
12. Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM TOSEM*, 11(2):256–290, 2002.
13. M. Kaufmann, P. Manolios, and J. S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, 2000.
14. Viktor Kuncak and Martin Rinard. Boolean algebra of shape analysis constraints. In *5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, 2004.

15. Nancy Lynch and Frits Vaandrager. Forward and backward simulations – Part I: Untimed systems. *Information and Computation, 121(2)*, 1995.
16. Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *Computer Systems*, 2(3):181–197, 1984.
17. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. SRI International, Computer Science Laboratory, 333 Ravenswood Avenue, Menlo Park CA 94025.
18. Thomas Reps, Mooly Sagiv, and Greta Yorsh. Symbolic implementation of the best transformer. In *VMCAI'04*, 2004.
19. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.
20. K. Thompson. UNIX implementation. *The Bell System Technical Journal*, 57(6 (part 2)), 1978.
21. A. Voronkov et al. The anatomy of Vampire (implementing bottom-up procedures with code trees). JAR, 15(2):237–265, 1995.
22. C. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 27, pages 1965–2013. Elsevier Science, 2001.
23. Markus Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Technische Universitaet Muenchen, 2002.
24. Jim Woodcock and Jim Davies. *Using Z*. Prentice-Hall, Inc., 1996.

A Some standard Athena libraries

A.1 Options

Options in Athena are represented as follows:

$$\textbf{datatype}\ \mathit{Option}(S) = \mathit{NONE} \mid \mathit{SOME}(S)$$

Here S is a *sort parameter*. Thus *Option* can be viewed as a sort constructor that takes an arbitrary sort S and builds a new sort, $\mathit{Option}(S)$.

Datatypes in Athena are free algebras with corresponding induction principles. For instance, the following axioms are automatically generated from the above definition:

$$\forall\, x : \mathit{Option}(S)\,.\, x = \mathit{NONE} \vee [\exists\, v : S\,.\, x = \mathit{SOME}(v)] \tag{34}$$

$$\forall\, v : S\,.\, \mathit{NONE} \neq \mathit{SOME}(v) \tag{35}$$

$$\forall\, v_1 : S, v_2 : S\,.\, \mathit{SOME}(v_1) = \mathit{SOME}(v_2) \Rightarrow v_1 = v_2 \tag{36}$$

Note that in the above axioms we annotated quantified variables with their sorts for readability purposes. In practice Athena uses a Hindley-Milner algorithm to infer the most general possible sorts of quantified variables, so such annotations are not necessary; we omit them in the remainder of this Appendix.

Structural induction may be performed on datatypes using a built-in syntax form that Athena offers for that purpose, and which automates much of the tedium associated with inductive proofs (e.g., managing inductive hypotheses in multiply nested inductive arguments).

A.2 Finite maps

Polymorphic finite maps are introduced in Athena as follows:

$$\textbf{structure}\ \mathit{FMap}(D,R) = \mathit{empty\text{-}map} \mid \mathit{update}(\mathit{FMap}(D,R),D,R)$$

Here D and R are sort parameters, representing the sorts of the domain and the range of the map, respectively. The declaration states that every finite map from D to R is either the *empty-map* or else it is of the form $\mathit{update}(m,x,v)$, i.e., it is an update of some other map m, obtained by binding the argument x to the value v (potentially overwriting whatever assignment x might have had in m).

Like data types, structures are inductively generated: axioms of the form (34) are valid for structures, and induction may be performed on them. However, structures are not necessarily freely generated (elements are not "uniquely readable"), hence Athena does not generate axioms such as (36) for structures.
We introduce two additional useful function symbols for finite maps:

$$\mathit{lookUp} : D \times \mathit{FMap}(D,R) \rightarrow \mathit{Option}(D)$$
$$\mathit{inDom} : D \times \mathit{FMap}(D,R) \rightarrow \mathit{Boolean}$$

whose semantics are captured by the following four axioms:

$$[M_1]\ \ \forall\, x\,.\,\mathit{lookUp}\,(x,\mathit{empty\text{-}map}) = \mathit{NONE}$$
$$[M_2]\ \ \forall\, x\ v\ m\,.\,\mathit{lookUp}\,(x,\mathit{update}(m,x,v)) = \mathit{SOME}(v)$$
$$[M_3]\ \ \forall\, x\ y\ v\ m\,.\,x \neq y \Rightarrow \mathit{lookUp}\,(x,\mathit{update}(m,y,v)) = \mathit{lookUp}\,(x,m)$$
$$[M_4]\ \ \forall\, x\ m\,.\,\mathit{inDom}(x,m) \Leftrightarrow [\exists\, v\,.\,\mathit{lookUp}\,(x,m) = \mathit{SOME}(v)]$$

We also have an extensionality axiom for finite maps:

$$[\mathit{FMExt}]\quad \forall\, m_1\ m_2\,.\,[\forall\, x\,.\,\mathit{lookUp}\,(x,m_1) = \mathit{lookUp}\,(x,m_2)] \Rightarrow m_1 = m_2$$

A.3 Resizable arrays

Resizable arrays are inductively generated by the following structure:

$$\begin{aligned}\textbf{structure}\ \mathit{RSArray}(S) = {} & \mathit{makeArray}(S,\mathit{Nat}) \\ \mid {} & \mathit{arrayWrite}(\mathit{RSArray}(S),\mathit{Nat},S,S)\end{aligned}$$

That is, a resizable array is either of the form $\mathit{makeArray}(x,n)$, which is a freshly constructed array of length n with the element x in every location from 0 to $n-1$; or else it is of the form $\mathit{arrayWrite}(A,i,x,f)$, i.e., obtained from an already existing array A by writing the value x into slot i. If i happens to be outside the bounds of A (i.e., $\mathit{arrayLen}(A) \leq i$), then the length will increase to $i+1$, the value x will be written into the i^{th} position of this extended array, and all the other newly allocated slots will be padded with the "fill" value f. This is made more clear in the axioms of Figure 6. Two additional useful functions are:

$$\mathit{arrayLen} : \mathit{RSArray}(S) \rightarrow \mathit{Nat}$$
$$\mathit{arrayRead} : \mathit{RSArray}(S) \times \mathit{Nat} \rightarrow \mathit{Option}(S)$$

Their semantics are captured by the nine axioms $[\mathbf{A}_1]$—$[\mathbf{A}_9]$ shown in Figure 6. Finally, we have an extensionality axiom for arrays:

$$[\mathit{RSAExt}]\ \ \forall\, A_1\ A_2\,.\,[\forall\, i\,.\,\mathit{arrayRead}(A_1,i) = \mathit{arrayRead}(A_2,i)] \Rightarrow A_1 = A_2.$$

$$[A_1]\ \forall\, A\ n\,.\, arrayLen(makeArray(A,n)) = n$$

$$[A_2]\ \forall\, A\ i\ v\ f\,.\, [i < arrayLen(A)] \Rightarrow arrayLen(arrayWrite(A,i,v,f)) = arrayLen(A)$$

$$[A_3]\ \forall\, A\ i\ v\ f\,.\, \neg\ [i < arrayLen(A)] \Rightarrow arrayLen(arrayWrite(A,i,v,f)) = i+1$$

$$[A_4]\ \forall\, A\ i\,.\, \neg\ [i < arrayLen(A)] \Rightarrow arrayRead(A,i) = NONE$$

$$[A_5]\ \forall\, x\ n\ i\,.\, i < n \Rightarrow arrayRead(makeArray(x,n),i) = SOME(x)$$

$$[A_6]\ \forall\, A\ i\ v\ f\,.\, arrayRead(arrayWrite(A,i,v,f),i) = SOME(v)$$

$$[A_7]\ \forall\, A\ i\ v\ f\,.\, i < arrayLen(A) \Rightarrow$$
$$[\forall\, j\,.\, i \neq j \Rightarrow arrayRead(arrayWrite(A,i,v,f),j) = arrayRead(A,j)]$$

$$[A_8]\ \forall\, A\ i\ v\ f\,.\, \neg\, [i < arrayLen(A)] \Rightarrow$$
$$[\forall\, j\,.\, j < arrayLen(A) \Rightarrow arrayRead(arrayWrite(A,i,v,f),j) = arrayRead(A,j)]$$

$$[A_9]\ \forall\, A\ i\ v\ f\,.\, \neg\, [i < arrayLen(A)] \Rightarrow$$
$$[\forall\, j\,.\, arrayLen(A) \leq j \wedge j < i \Rightarrow arrayRead(arrayWrite(A,i,v,f),j) = SOME(f)]$$

Fig. 6. The semantics of resizable arrays

A.4 Natural numbers

Numeric reasoning played an important role in this project. Although no deep number-theoretic results were needed, it was still necessary to introduce all the usual arithmetic operations, including the remainder operation, and derive many simple results for them. We start by introducing the natural numbers as an algebraic datatype:

$$\textbf{datatype Nat} = \textbf{zero} \mid \textbf{succ}(\textbf{Nat})$$

This definition automatically generates the following axioms:

$$\forall\, x\,.\, \textbf{zero} \neq \textbf{succ}(x)$$
$$\forall\, x, y\,.\, \textbf{succ}(x) = \textbf{succ}(y) \Rightarrow x = y$$
$$\forall\, x\,.\, x = \textbf{zero} \vee (\exists\, y\,.\, x = \textbf{succ}(y))$$

which are then added to the assumption base.

Next, we introduce function symbols for the predecessor operation:

$$\textbf{declare pred}\colon Nat \rightarrow Nat$$

as well as for (binary) addition, subtraction, multiplication, division, and remainder:

$$\textbf{declare}\ +, -, *, div, mod :\ Nat \rightarrow Nat$$

We also introduce operators for numeric comparisons:

$$\textbf{declare}\ \ <, \leq\colon\ Nat \times Nat \rightarrow Boolean$$

The semantics of these symbols are given via equational axioms (possibly conditionally equational axioms) that capture the usual primitive recursive definitions of these operations. For example, predecessor is defined as a total function as follows:

$$\mathbf{pred}(\mathbf{zero}) = \mathbf{zero} \wedge \forall\ x \,.\, \mathbf{pred}(\mathbf{succ}(x)) = x$$

The definition of binary addition is given via the two axioms:

$$\forall\ y \,.\, \mathbf{zero} + y = y$$
$$\forall\ x, y \,.\, \mathbf{succ}(x) + y = \mathbf{succ}(x + y)$$

The definitions of subtraction, multiplication, and numeric comparisons are given by the following axioms:

$$\forall\ x \,.\, \mathbf{zero} - x = \mathbf{zero}$$
$$\forall\ x \,.\, x - \mathbf{zero} = x$$
$$\forall\ x, y \,.\, \mathbf{succ}(x) - \mathbf{succ}(y) = x - y$$
$$\forall\ y \,.\, \mathbf{zero} * y = \mathbf{zero}$$
$$\forall\ x, y \,.\, \mathbf{succ}(x) * y = y + (x * y)$$
$$\forall\ x \,.\, (x < \mathbf{zero}) = \mathbf{false}$$
$$\forall\ x \,.\, (\mathbf{zero} < \mathbf{succ}(x)) = \mathbf{true}$$
$$\forall\ x, y \,.\, (\mathbf{succ}(x) < \mathbf{succ}(y)) = x < y$$

The less-than-or-equal symbol is defined in terms of less-than:

$$x \leq y \Leftrightarrow x = y \vee x < y$$

The definitions of quotient and remainder are as follows:

$$\forall\ x \,.\ \ x\ \mathit{div}\ \mathbf{zero} = \mathbf{zero}$$
$$\forall\ x, y \,.\, x < y \Rightarrow x\ \mathit{div}\ y = \mathbf{zero}$$
$$\forall\ x, y \,.\, (y \neq \mathbf{zero}) \wedge \neg(x < y) \Rightarrow x\ \mathit{div}\ y = \mathbf{succ}((x - y)\ \mathit{div}\ x)$$
$$\forall\ x \,.\ \ x\ \mathit{mod}\ \mathbf{zero} = x$$
$$\forall\ x, y \,.\, x < y \Rightarrow x\ \mathit{mod}\ y = x$$
$$\forall\ x, y \,.\, x < y \Rightarrow x\ \mathit{mod}\ y = x$$
$$\forall\ x, y \,.\, (y \neq \mathbf{zero}) \wedge \neg(x < y) \Rightarrow x\ \mathit{mod}\ y = (x -\ y)\ \mathit{mod}\ y$$

From the above definitions, a number of useful properties can be derived, e.g., that addition is commutative. Most of these properties are derivable only with the aid of a mathematical induction principle—in our case, structural induction on the datatype **Nat**. Structural induction in this case corresponds to conventional mathematical induction on the natural numbers. Occasionally it is very convenient to be able to use *strong induction* instead, whereby one inductively assumes the truth of the statement $P(n)$ for all $m < n$. For instance, the so-called "division algorithm" result, which states

$$0 < b \Rightarrow [(a\ \mathit{div}\ b) * b] + [a\ \mathit{mod}\ b] = a$$

can be readily proved by strong induction but is much more tedious with conventional induction. In Athena, a strong induction principle on natural numbers is currently formulated as a primitive method. Figure 7 depicts some numeric results that were needed at various points in the project. Most of them were proved automatically by Athena methods that mechanize induction, but a few of them required more detailed proofs. The reader can refer to the file `nat.ath` in the source code listing for details.

1. $\forall\ x, y, z\ .\ x < y \land y < z \Rightarrow x < z$
2. $\forall\ x, y\ .\ x < y \Rightarrow \neg(y < x)$
3. $\forall x\ .\ \neg(x < x)$
4. $\forall x, y\ .\ x < y \Rightarrow x \leq y$
5. $\forall x\ .\ x < \mathbf{succ}(x)$
6. $\forall x, y\ .\ x < \mathbf{succ}(y) \Leftrightarrow [x = y \lor x < y]$
7. $\forall x, y\ .\ x < y \Rightarrow x < \mathbf{succ}(y)$
8. $\forall x, y\ .\ x < y \land y \leq z \Rightarrow x < z$
9. $\forall x, y\ .\ x \leq y \land y < z \Rightarrow x < z$
10. $\forall x, y\ .\ x < y \Rightarrow x \neq y$
11. $\forall x\ .\ x \neq \mathbf{zero} \Rightarrow \mathbf{zero} < x$
12. $\forall x\ .\ x \neq \mathbf{zero} \Rightarrow [\exists\ y\ .\ y < x]$
13. $\forall x, y\ .\ x < y \Rightarrow \mathbf{succ}(x) \leq y$
14. $\forall\ x, y\ .\ x + \mathbf{zero} = x$
15. $\forall\ x, y\ .\ x + \mathbf{succ}(y) = \mathbf{succ}(x + y)$
16. $\forall\ x, y\ .\ x + y = y + x$
17. $\forall\ x, y, z\ .\ x + (y + z) = (x + y) + z$
18. $\forall\ x, y\ .\ \neg(y < x) \Rightarrow x + (y - x) = y$
19. $\forall\ x, y\ .\ \mathbf{zero} < y \Rightarrow [(x\ div\ y) * y] + x\ mod\ y = x$
20. $\forall\ x, y\ .\ \mathbf{zero} < y \Rightarrow x\ mod\ y < y$
21. $\forall\ x, y\ .\ \mathbf{zero} < y \land [\mathbf{succ}(x)\ mod\ y = \mathbf{zero}] \Rightarrow \mathbf{succ}(x\ div\ y) = \mathbf{succ}(x)\ div\ y$
22. $\forall\ x, y, z, w\ .\ x = \mathbf{succ}(y) \land \mathbf{succ}(y\ div\ z) = w \land \mathbf{zero} < z \land \mathbf{succ}(y)\ mod\ z = \mathbf{zero} \Rightarrow x = w * z$
23. $\forall\ x, y\ .\ \mathbf{zero} < x \land \mathbf{zero} < [\mathbf{succ}(x)\ mod\ y] \Rightarrow (\mathbf{succ}(x)\ mod\ y) = \mathbf{succ}(x\ mod\ y)$
24. $\forall\ x, y, z, w\ .\ x = \mathbf{succ}(y) \land \mathbf{succ}(y\ div\ z) = w \land \mathbf{zero} < z \land \mathbf{zero} < \mathbf{succ}(y)\ mod\ z \Rightarrow x = \mathbf{pred}(w) * z + x\ mod\ z$
25. $\forall\ x, y, z\ .\ x \leq y \Rightarrow x\ div\ z \leq y\ div\ z$
26. $\forall\ x, y, z\ .\ x \leq y \Rightarrow x * z \leq y * z$
27. $\forall\ x, y, z\ .\ x \leq y \land y \leq z \Rightarrow x \leq z$
28. $\forall\ x\ .\ x \leq \mathbf{pred}(x)$
29. $\forall\ x, y\ .\ \mathbf{zero} < y \Rightarrow (x * y)\ div\ y = x$
30. $\forall\ x, y\ .\ x \leq x + y$
31. $\forall\ x, y\ .\ \mathbf{pred}(x) \leq y \land y < x \Rightarrow \mathbf{succ}(y) = x$
32. $\forall\ x, y\ .\ x < \mathbf{succ}(y) \Rightarrow x = y \lor x < y$
33. $\forall\ x, y, z\ .\ x < y \Rightarrow x < y + z$
34. $\forall\ x, y\ .\ (x\ div\ y) * y \leq x$
35. $\forall\ x, y, z\ .\ x + y = z + y \Rightarrow x = z$
36. $\forall\ x, y, z\ .\ \mathbf{zero} < y \land x * y = z * y \Rightarrow x = z$
37. $\forall\ x, y, z\ .\ \mathbf{zero} < y \land x \leq \mathbf{pred}(y) \Rightarrow x < y$
38. $\forall\ x, y\ .\ x < y \Rightarrow x \leq y$
39. $\forall\ x, y\ .\ x = y \Rightarrow \neg(x < y)$
40. $\forall\ x, y\ .\ x < y \Rightarrow \mathbf{succ}(x) \leq y$
41. $\forall\ x\ .\ \mathbf{zero} \leq x$
42. $\forall\ x, y\ .\ x < y \lor x = y \lor y < x$
43. $\forall\ x\ .\ \mathbf{zero}\ mod\ x = \mathbf{zero}$
44. $\forall\ x\ .\ szInv(\mathbf{zero}, \mathbf{zero}, x)$
45. $\forall\ x, y, z, w\ .\ szInv(x, y, z) \land x \leq w \land \mathbf{zero} < z \Rightarrow \mathbf{pred}(y) \leq (w\ div\ z)$
46. $\forall\ x, y, z, w\ .\ szInv(x, y, z) \land w < x \land \mathbf{zero} < z \land \mathbf{zero} < y \Rightarrow (w\ div\ z) < y$
47. $\forall\ x, y, z\ .\ \mathbf{zero} < z \land x \neq y \Rightarrow (x\ div\ z) \neq (y\ div\ z) \lor (x\ mod\ z) \neq (y\ mod\ z)$

Fig. 7. Useful results about the natural numbers.