Verifying a file system implementation

Arkoudas, Zee, Kuncak, Rinard

MIT

Unix-style file systems

Key operations: read/write

Their abstract specifications are straightforward

But implementations are more involved

Disk data managed by an *inode* scheme; file contents are not stored contiguously

Read/write algorithms go through multiple levels of indirection before accessing file data

Can we verify their correctness?

Motivation

File systems are essential infrastructure

Reliability is crucial

File reading and writing are the key operations

Explore some questions:

- What kind of proof techniques might be necessary/applicable?
- What types of invariants?
- Can we develop a formal model incrementally?

Use this as a testbed for Athena's approach to proof automation: integrate cutting-edge ATPs with true natural deduction tactics

Proof idea

We have one *abstract file system* model

And one *concrete file system* model

The abstract model hides implementation details; the concrete model includes them

We relate the two by an abstraction mapping that embeds the concrete model into the abstract one

We prove that the mapping is a morphism

This shows that one model simulates the other

Disclaimer

The “concrete” file system is still a rather abstract model

It does not directly correspond to an actual-code implementation

Rather, we distill the essence of the implementation in a logical formalism

Also, many important aspects of “real” file systems are left out: subdirectories, concurrency/caching, date/time stamps, file links.

Nevertheless, we did extend the model with user permissions; most proofs were reused

Proof Style

Proof too complex to be done automatically

Instead, we use *interactive* theorem proving

We build up the proof in a (long) series of lemmas

Each lemma is proved in natural deduction style in Athena

Each proof is mechanically checked

Automation is still heavily used in each proof

ATPs used to discharge tedious proof obligations

Abstract File System

An abstract state σ maps file ids to files:

$$\sigma : \mathit{FileId} \rightarrow \mathit{File}$$

A file is a resizable array of bytes: $\mathit{File} = \mathit{RSArray}(\mathit{Byte})$

Reading:

$$\mathit{abRead}(f, i, \sigma) \in \{\mathit{fileNotFound}, \mathit{EOF}, \mathit{Ok}(v)\}$$

Writing:

$$\mathit{abWrite}(f, i, v, \sigma) \;\; \textit{produces a new abstract state} \;\; \sigma'$$

Specification of Abstract Read

$abRead(f, i, \sigma)$: Is f bound in σ?

No: Return *fileNotFound*

Yes: Let A be the array $\sigma(f)$

Is $i \geq length(A)$?

Yes: Return *EOF*

No: Return $A[i]$

Specification of Abstract Write

$abWrite(f, i, v, \sigma)$: Is f bound in σ?

Yes: Let A be the array $\sigma(f)$

Is $i < length(A)$?

Yes: Return $\sigma[A \mapsto A[i \mapsto v]]$

No: Let A' be the (padded) extension of A to $i + 1$

Return $\sigma[A \mapsto A'[i \mapsto v]]$

No: Let A' be a new (padded) array of length $i + 1$

Return $\sigma[A \mapsto A'[i \mapsto v]]$

(Snapshot of) Athena Formalization

```
(domains Byte FileId)

(declare fillByte Byte)

(define File (RSArray-Of Byte))

(define AbState (FMap FileId File))

(datatype ReadResult
  EOF
  (Ok Byte)
  fileNotFound)

(declare abRead (-> (FileID Nat AbState) ReadResult))


(define abRead-clause-1
  (forall ?fid ?i ?sigma
    (if (= (lookUp ?fid ?sigma) NONE)
        (= (abRead ?fid ?i ?sigma) FileNotFound))))


(assert abRead-clause-1)
```

Concrete file system states

A disk *block* is a fixed-size (**blockSize**) array of bytes

An inode is a record consisting of:

- File size
- User permissions
- Block count (no. of blocks used by the file)
- List of block numbers

A concrete state s is a record consisting of:

- The total number of inodes
- The inode mapping (from natural numbers to inodes)
- The total number of disk blocks currently in use
- The block mapping (from block numbers to blocks)
- The file system *root*, mapping file identifiers to inode numbers

root (maps file identifiers to inode numbers)
foo.h
bar.c
0 1 2 ... ... 35 36 ...
inodes
(maps inode numbers to inode records)
inodes[2]
inodes[36]
file Size: 297 | user Perms | block Count: 4 | block List:
...
file Size: 7396 | user Perms | block Count: 49 | block List:
0 1 2 3
83 3 529 1
...
0 1 48
713 89 ...
0 1 2 3 ... 88 89 ...
stateBlocks
(maps blocks numbers to disk blocks)
H ... B
0 blockSize-1
A ... C
0 blockSize-1

$$\mathbf{stateBlocks}[\mathbf{inodes}[36].\mathbf{blockList}[1]][\mathbf{blockSize} - 1] = C$$

Figure 1: A snapshot of the file system's state

Reachable states

The *initial* state has:

- zero inodes
- zero blocks
- empty inode and block mappings
- an empty root

A *reachable* state $\widehat{s}$ is obtainable from the initial state through an arbitrary finite sequence of writes

We are only concerned with reachable states, since they are the ones that can occur in practice

Specification of Concrete Read

$conRead(f, i, s)$: Is f bound in s?

No: Return *fileNotFound*

Yes: Let n be the inode number $s(f)$

Let $r = inodes(s)[n]$

Is $i \geq fileSize(r)$?

Yes: Return *EOF*

No: Let $bn = blockList(r)[i \,/\, \mathbf{blockSize}]$

Let $b = blocks(s)[bn]$

Return $b[i \mod \mathbf{blockSize}]$

Specification of Concrete Write

The definition of *conWrite(f,i,v,s)* is a little more complicated. Assuming that f is bound in s to some inode number n, let $r = inodes(s)[n]$. We have 3 main cases:

- The index i is less than $fileSize(r)$. We just find the right block and update i. File size does not change.
- $i \geq fileSize(r)$, but still within the last block:

  $$(i \,/\, \mathbf{blockSize}) < blockCount(r)$$

  File size increases, but there is no need to allocate new blocks.
- Neither of the above. The proper number of blocks must be (recursively) allocated before the write is done. This case changes the greatest number of state components.

If not bound, we must allocate everything

10 auxiliary state-transforming functions necessary

Abstraction

Define an abstraction relation A from concrete to abstract states:

$$A(s, \sigma) \Leftrightarrow [\forall\, f, i \,.\, conRead(f, i, s) = abRead(f, i, \sigma)]$$

So the abstract state "agrees" with the concrete state everywhere

We can easily prove univalence:

$$\forall\, s, \sigma_1, \sigma_2 \,.\, A(s, \sigma_1) \wedge A(s, \sigma_2) \Rightarrow \sigma_1 = \sigma_2$$

Introduce an abstraction function α mapping *reachable* concrete states to abstract states:

$$[\alpha(\widehat{s}) = \sigma] \Leftrightarrow A(\widehat{s}, \sigma)$$

Because we are dealing with reachable states, the mapping α can be defined constructively

Correctness Formulation

We need to show that the abstraction function is a homomorphic embedding of concrete reachable states into abstract states:

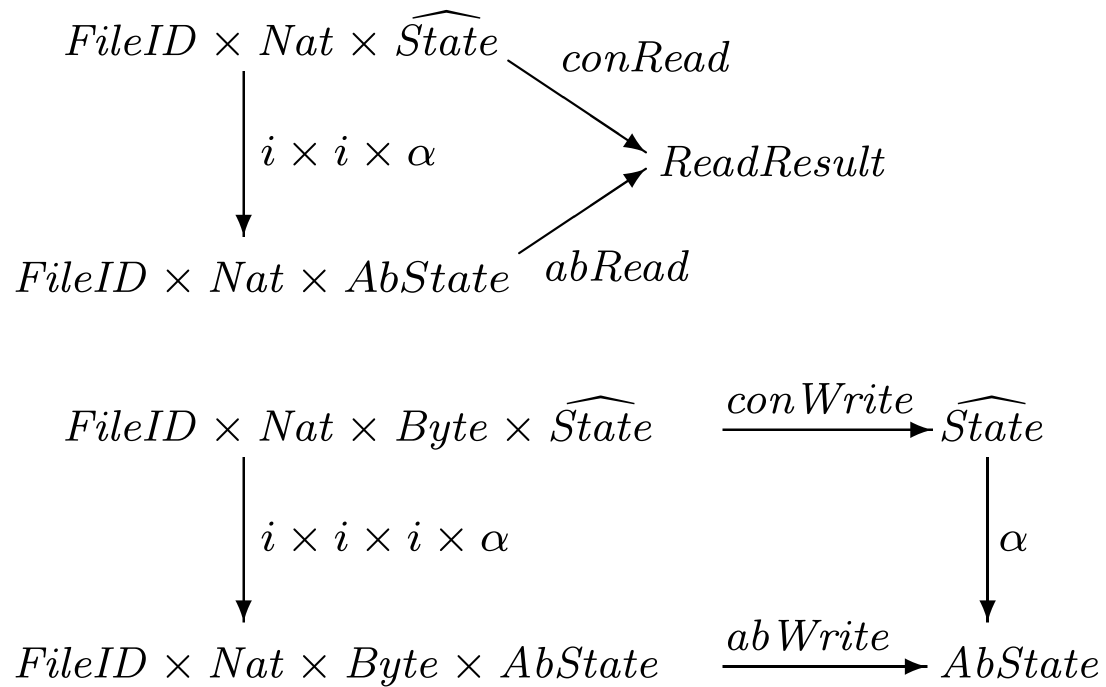
Correctness of concrete read:

$$\forall\, f, i, \widehat{s} \,.\, conRead(f, i, \widehat{s}) = abRead(f, i, \alpha(\widehat{s}))$$

Correctness of concrete write:

$$\forall\, f, i, v, \widehat{s} \,.\, \alpha(conWrite(f, i, v, \widehat{s}) = abWrite(f, i, v, \alpha(\widehat{s}))$$

Commuting diagrams



Analysis

Correctness of read is immediate

Correctness of write boils down to $L = R$ where

$$L = conRead(f', i, conWrite(f, j, v, \widehat{s}))$$

and

$$R = abRead(f', i, abWrite(f, j, v, \alpha(\widehat{s})))$$

(All variables assumed to be universally quantified)

This can be proved by a case analysis (8 cases), with the help of—many—lemmas

Invariants

A state property is an *invariant* if it holds for all reachable states

We prove an invariant I by induction:

1. Show that the initial state has I (basis step);
2. Prove: $I(s) \Rightarrow I(conWrite(f, i, v, s))$ (inductive step)

Invariants were crucial in the proof. E.g., we needed to show that different inodes use different blocks

This does not hold for arbitrary states, but does hold for reachable states

We had to prove 11 invariants for this proof

We had to show that each invariant was preserved by every state-changing operation

This was the largest part (80%) of the proof effort

Tactics

Proof-search programmability was crucial

Writing code to automate recurrent reasoning patterns is useful

But the code cannot be arbitrary: soundness must be ensured

HOL-style programmability is the answer

But programming with sequents is not easy

Proof programming in natural deduction style seems easier

Proof statistics

About 6,000 lines of proof text

About 280 lemmas

Average proof length for a lemma: 18 lines

Total time to check the proof: about 9 minutes

Average time for each lemma: about 2 seconds

Total number of calls to ATPs (Vampire and Spass): over 2000

Conclusions

Total deductive verification still quite challenging, but proof automation making promising strides

Complete proof automation for large system verification efforts is unlikely, even in the future

Therefore, interaction is inevitable

This means that there must be flexible ways of guiding the search

One such way: let the users highlight the main proof steps, as they would in an informal proof; let ATPs handle tedious steps

Greater automation is crucial. ATPs are great, but too general-purpose

Methods are a great tool for exploiting domain-specific reasoning knowledge

Athena allows for methods written in Fitch style

In our experience, methods are easier to write in that style

Some More on Athena

Proof system for first-order logic with sorts and polymorphism

Supports algebraic data types and structural induction

Emphasis on proof readability and writability: proofs are expressed in true natural deduction style

Proofs can be abstracted into proof algorithms very easily

Integrated with:

- high-performance ATPs (Vampire/Spass)
- model generators (Paradox, Alloy) for counter-examples when ATPs fail