

Data refinement of representation of a file

Karen Zee and Viktor Kuncak

March 17, 2025

Abstract

This document illustrates the verification of basic file operations (file creation, file read and file write) in Isabelle theorem prover [4]. We describe a file at two levels of abstraction: an abstract file represented as a resizable array, and a concrete file represented using data blocks.

Contents

1	Introduction	1
2	Arrays without bounds	2
3	Resizable arrays	3
4	Data refinement of representation of a file	4
4.1	Abstract File	4
4.2	Concrete File	5
4.2.1	Writing File	5
4.3	Reachability Invariants for Concrete File	6
4.4	Initial File Satisfies Invariants	7
4.5	Properties of Concrete File Operations	7
4.6	Concrete File Operations Preserve Invariants	8
4.7	Commuting Diagrams for Simulation Relation	9
4.7.1	Abstraction Function	9
4.7.2	Creating a File	9
4.7.3	File Size	9
4.7.4	Read Operation	9
4.7.5	Write Operation	9

1 Introduction

This document is based on [1], which explores the challenges of verifying the core operations of a Unix-like file system [5, 3]. The paper [1] formalizes

the specification of the file system as a map from file names to sequences of bytes, then formalizes an implementation that uses such standard file system data structures as i-nodes and fixed-sized disk blocks. The correctness of the implementation is verified by proving the existence of a simulation relation [2] between the specification and the implementation. The original effort of [1] started in Isabelle. The process of developing the proof in Isabelle helped to remove the initial bugs in the concrete and abstract models (though the proof has not been completed so far).

Here we present a completed proof for a simplified problem: data refinement of a single file. We present operations on both abstract and concrete files, define a function mapping concrete files to abstract files, and prove that this function is a simulation relation.

We use two libraries of arrays: arrays without bounds checks, which can be thought of as an array with an unbounded number of elements, and resizable arrays, which have a notion of the current size, but expand in response to array writes that are outside the current bounds.

2 Arrays without bounds

```
theory CArrays imports Main begin
```

For these arrays there is no built-in protection against reading or writing out-of-bounds.

```
type-synonym 'a cArray = nat => 'a

definition makeCArray :: nat => 'a => 'a cArray where
  makeCArray arraySize fillValue index ==
    if index < arraySize then fillValue else undefined

definition readCArray :: 'a cArray => nat => 'a where
  readCArray array index == array index

definition writeCArray :: 'a cArray => nat => 'a => 'a cArray where
  writeCArray array index value == array(index := value)
```

```
lemma makeCArrayCorrect:
  index < arraySize ==>
    readCArray (makeCArray arraySize fillValue) index = fillValue
  ⟨proof⟩
```

```
lemma writeCArrayCorrect1:
  readCArray (writeCArray array index value) index = value
  ⟨proof⟩
```

```

lemma writeCArrayCorrect2:
  index1 ~ = index2 ==>
    readCArray (writeCArray array index1 value) index2 =
      readCArray array index2
  ⟨proof⟩

end

```

3 Resizable arrays

```
theory ResizableArrays imports Main begin
```

These arrays resize themselves, padding with `fillValue`.

```
type-synonym 'a rArray = nat * (nat => 'a)
```

```
definition fillAndUpdate :: nat => (nat => 'a) => nat => 'a => 'a => (nat
=> 'a) where
  fillAndUpdate len f i value fillValue j ==
    if j=i then value
    else if (len <= j & j < i) then fillValue
    else f j
```

```
definition raWrite :: 'a rArray => nat => 'a => 'a rArray where
  raWrite arr i value fillValue ==
    (let len = fst arr;
     f = snd arr
     in
       if i < len then (len,f(i:=value))
       else (i+1, fillAndUpdate len f i value fillValue)
    )
```

```
definition cutoff :: 'a => 'a rArray => 'a rArray where
  cutoff fill arr ==
    (fst arr,
     % i. if i < fst arr then snd arr i else fill)
```

```
lemma raWriteSizeSame [simp]: i < fst arr ==> fst (raWrite arr i value fillValue)
= fst arr
⟨proof⟩
```

```
lemma raWriteSizeGrows [simp]: (fst arr <= i) ==> fst (raWrite arr i value
fillValue) = i+1
⟨proof⟩
```

```
lemma raWriteContentChanged [simp]: snd (raWrite arr i value fillValue) i =
value
⟨proof⟩
```

```
lemma raWriteContentOld [simp]: [| j < fst arr; i ~ = j |] ==>
```

```

     $\text{snd} (\text{raWrite arr } i \text{ value } \text{fillValue}) j = \text{snd arr } j$ 
⟨proof⟩

lemma raWriteContentFill [simp]: [| fst arr < j; j < i |] ==>
     $\text{snd} (\text{raWrite arr } i \text{ value } \text{fillValue}) j = \text{fillValue}$ 
⟨proof⟩

end

```

4 Data refinement of representation of a file

theory *FileRefinement imports Complex-Main CArrays ResizableArrays begin*

We describe a file at two levels of abstraction: an abstract file, represented as a resizable array, and a concrete file, represented using data blocks. We consider the following operations:

```

makeAFS      :: AFile
afsRead       :: "AFile => nat \<rightharpoonup> byte"
afsWrite      :: "AFile => nat => byte \<rightharpoonup> AFile"
afsFileSize   :: "AFile => nat"

typeddecl
  — unit of file content
  byte
consts
  — value used for padding
  fillByte :: byte

axiomatization
  blockSize :: nat — in bytes and
  numBlocks :: nat — total number of blocks in the file system
where
  nonZeroBlockSize: blockSize > 0 and
  nonZeroNumBlocks: numBlocks > 0

```

4.1 Abstract File

type-synonym *AFile* = *byte rArray* — abstract file is a resizable array of bytes

```

definition makeAF :: AFile
where — initial file has size 0
makeAF == (0, % index. fillByte)

```

```

definition afSize :: AFile => nat where
  — file size is the length of the resizable array
afSize afle == fst afle

```

```

definition afRead :: AFile => nat -> byte where
  — reading from a file looks up the byte, reporting None if the index is out of file
  bounds
  afRead afile byteIndex ==
    if byteIndex < fst afile then Some ((snd afile) byteIndex) else None

definition afWrite :: AFile => nat => byte -> AFile where
  — writing to a file updates the file content and extends the file if there is enough
  space
  afWrite afile byteIndex value ==
    if byteIndex div blockSize < numBlocks then
      Some (raWrite afile byteIndex value fillByte)
    else None

```

4.2 Concrete File

type-synonym Block = byte cArray — array of blockSize bytes

```

record CFile =
  fileSize :: nat — in bytes
  nextFreeBlock :: nat — next block available for allocation
  data :: Block cArray — array of up to numBlocks blocks

```

```

definition makeCF :: CFile
where — initial file has no allocated blocks
makeCF ==
  (| fileSize = 0,
   nextFreeBlock = 0,
   data = makeCArray numBlocks (makeCArray blockSize fillByte)
  |)

```

```

definition cfSize :: CFile => nat where
cfSize cfile == fileSize cfile

```

```

definition cfRead :: CFile => nat -> byte where
  — Looks up correct data block and reads its content, if byteIndex is within
  bounds, else returns None.
  cfRead cfile byteIndex ==
    if byteIndex < fileSize cfile then
      (let i = byteIndex div blockSize in
       (let j = byteIndex mod blockSize in
        Some (readCArray (readCArray (data cfile) i) j)))
    else None

```

4.2.1 Writing File

We first present some auxiliary operations.

```

definition cfWriteNoExtend :: CFile => nat => byte => CFile where

```

— Writing to a file when *byteIndex* is within bounds.

```
cfWriteNoExtend cfile byteIndex value ==
let i = byteIndex div blockSize in
let j = byteIndex mod blockSize in
let block = readCArray (data cfile) i in
cfile(| data :=
  writeCArray (data cfile) i (writeCArray block j value) |)
```

definition *cfExtendFile* :: *CFile* => *nat* => *CFile* **where**

— Writing to a file when *byteIndex* is out of bounds. Involves allocating a new block.

```
cfExtendFile cfile byteIndex ==
cfile(| fileSize := Suc byteIndex,
  nextFreeBlock := Suc (byteIndex div blockSize) |)
```

The main file write operation.

definition *cfWrite* :: *CFile* => *nat* => *byte* → *CFile* **where**

— Writes the file at byte location *byteIndex*, automatically extending the file to that byte location if *byteIndex* is not within bounds.

```
cfWrite cfile byteIndex value ==
if byteIndex div blockSize < numBlocks then
  if byteIndex < fileSize cfile then
    Some (cfWriteNoExtend cfile byteIndex value)
  else
    Some (cfWriteNoExtend (cfExtendFile cfile byteIndex) byteIndex value)
else None
```

4.3 Reachability Invariants for Concrete File

definition *nextFreeBlockInvariant* :: *CFile* => *bool* **where**

```
nextFreeBlockInvariant state ==
(fileSize state + blockSize - 1) div blockSize = nextFreeBlock state
```

definition *unallocatedBlocksInvariant* :: *CFile* => *bool* **where**

— This invariant of the implementation is needed to prove *writeExtendCorrect*. It says that any unallocated block contains *fillByte*'s.

```
unallocatedBlocksInvariant state ==
 $\forall blockNum\ i.\ \sim blockNum < nextFreeBlock\ state \& blockNum < numBlocks \& i < blockSize \rightarrow data\ state\ blockNum\ i = fillByte$ 
```

definition *lastBlockInvariant* :: *CFile* => *bool* **where**

```
lastBlockInvariant state ==
 $\forall index.\ \sim index < fileSize\ state \& nextFreeBlock\ state = index\ div\ blockSize + 1 \rightarrow data\ state\ (index\ div\ blockSize)\ (index\ mod\ blockSize) = fillByte$ 
```

definition *reachabilityInvariant* :: *CFile* => *bool* **where**

```
reachabilityInvariant cfile ==
```

```

nextFreeBlockInvariant cfile &
unallocatedBlocksInvariant cfile &
lastBlockInvariant cfile

```

4.4 Initial File Satisfies Invariants

We prove each invariant individually and then summarize.

```

lemma nextFreeBlockInvariant1:
  nextFreeBlockInvariant makeCF
  ⟨proof⟩

lemma unallocatedBlocksInvariant1:
  unallocatedBlocksInvariant makeCF
  ⟨proof⟩

lemma lastBlockInvariant1:
  lastBlockInvariant makeCF
  ⟨proof⟩

lemma makeCFpreserves: reachabilityInvariant makeCF
  ⟨proof⟩

```

4.5 Properties of Concrete File Operations

```

lemma cfWriteNoExtendPreservesFileSize:
  [] index < fileSize cfile1;
    cfWrite cfile1 index value = Some cfile2
  [] ==>
    fileSize cfile2 = fileSize cfile1
  ⟨proof⟩

lemma cfWriteExtendFileSize:
  [] ~ index < fileSize cfile1;
    cfWrite cfile1 index value = Some cfile2
  [] ==> fileSize cfile2 = Suc index
  ⟨proof⟩

lemma fileSizeIncreases:
  cfWrite cfile1 index value = Some cfile2
  ==> fileSize cfile1 <= fileSize cfile2
  ⟨proof⟩

lemma nextFreeBlockIncreases:
  [] nextFreeBlockInvariant cfile1;
    cfWrite cfile1 index value = Some cfile2
  [] ==> nextFreeBlock cfile1 <= nextFreeBlock cfile2
  ⟨proof⟩

```

4.6 Concrete File Operations Preserve Invariants

There is only one top-level concrete operation: write, and we show that it preserves all reachability invariants.

```

lemma cfWritePreservesNextFreeBlockInvariant:
  [] reachabilityInvariant cfile1;
    cfWrite cfile1 byteIndex value = Some cfile2
  [] ==> nextFreeBlockInvariant cfile2
⟨proof⟩

lemma modInequalityLemma:
  (a::nat) ~= b & a mod c = b mod c ==> a div c ~= b div c
⟨proof⟩

lemma mod-round-lt:
  [] 0 < (c::nat);
    a < b
  [] ==> a div c < (b + c - 1) div c
⟨proof⟩

lemma blockNumNELemma:
  !!blockNum i.
  [] nextFreeBlockInvariant cfile1;
    cfile1
    (| data :=
      writeCArray (data cfile1) (byteIndex div blockSize)
      (writeCArray
        (readCArray (data cfile1) (byteIndex div blockSize))
        (byteIndex mod blockSize) value) |) =
    cfile2;
    ~ blockNum < nextFreeBlock cfile2; blockNum < numBlocks;
    i < blockSize; byteIndex div blockSize < numBlocks;
    byteIndex < fileSize cfile1 []
  ==> blockNum ~= byteIndex div blockSize
⟨proof⟩

lemma cfWritePreservesUnallocatedBlocksInvariant:
  [] reachabilityInvariant cfile1;
    cfWrite cfile1 byteIndex value = Some cfile2
  [] ==> unallocatedBlocksInvariant cfile2
⟨proof⟩

lemma cfWritePreservesLastBlockInvariant:
  [] reachabilityInvariant cfile1;
    cfWrite cfile1 byteIndex value = Some cfile2 [] ==>
    lastBlockInvariant cfile2
⟨proof⟩

```

Final statement that all invariants are preserved.

```

lemma cfWritePreserves:
  [] reachabilityInvariant cfile1;
    cfWrite cfile1 byteIndex value = Some cfile2 [] ==>
    reachabilityInvariant cfile2
  ⟨proof⟩

```

4.7 Commuting Diagrams for Simulation Relation

Here we show correctness of file system operations.

4.7.1 Abstraction Function

```

definition abstFn :: CFile => AFile where
  abstFn cfile == (fileSize cfile,
    % index . case cfRead cfile index of
      None => fillByte
      | Some value => value)

primrec oAbstFn :: CFile option => AFile option where
  oAbstFn None = None
  | oAbstFn (Some s) = Some (abstFn s)

```

4.7.2 Creating a File

```

lemma makeCFCorrect: abstFn makeCF = makeAF
  ⟨proof⟩

```

4.7.3 File Size

```

lemma fileSizeCorrect:
  cfSize cfile = afSize (abstFn cfile)
  ⟨proof⟩

```

4.7.4 Read Operation

```

lemma readCorrect:
  cfRead cfile = afRead (abstFn cfile)
  ⟨proof⟩

```

4.7.5 Write Operation

```

lemma writeNoExtendCorrect:
  [] index < fileSize cfile1;
    Some cfile2 = cfWrite cfile1 index value
  [] ==> Some (abstFn cfile2) = afWrite (abstFn cfile1) index value
  ⟨proof⟩

```

```

lemma writeExtendCorrect:
  [] nextFreeBlockInvariant cfile1;
    unallocatedBlocksInvariant cfile1;

```

```

lastBlockInvariant cfile1;
~ index < fileSize cfile1;
Some cfile2 = cfWrite cfile1 index value
[] ==> Some (abstFn cfile2) = afWrite (abstFn cfile1) index value
⟨proof⟩

lemma writeSucceedCorrect:
[] nextFreeBlockInvariant cfile1;
unallocatedBlocksInvariant cfile1;
lastBlockInvariant cfile1;
Some cfile2 = cfWrite cfile1 index value
[] ==> Some (abstFn cfile2) = afWrite (abstFn cfile1) index value
⟨proof⟩

lemma writeFailCorrect:
cfWrite cfile1 index value = None ==>
afWrite (abstFn cfile1) index value = None
⟨proof⟩

lemma writeCorrect:
reachabilityInvariant cfile1 ==>
oAbstFn (cfWrite cfile1 index value) = afWrite (abstFn cfile1) index value
⟨proof⟩

end

```

References

- [1] K. Arkoudas, K. Zee, V. Kuncak, and M. Rinard. Verifying a file system implementation. In *Sixth International Conference on Formal Engineering Methods (ICFEM'04)*, volume 3308 of *LNCS*, Seattle, Nov 8-12, 2004 2004.
- [2] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-oriented proof methods and their comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. 1998.
- [3] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *Computer Systems*, 2(3):181–197, 1984.
- [4] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283. 2002. <http://www.in.tum.de/~nipkow/LNCS2283/>.
- [5] K. Thompson. UNIX implementation. *The Bell System Technical Journal*, 57(6 (part 2)), 1978.