

VerifyThis 2019 – Polished Isabelle Solutions

Peter Lammich Simon Wimmer

December 14, 2021

Abstract

VerifyThis 2019 (<http://www.pm.inf.ethz.ch/research/verifythis.html>) was a program verification competition associated with ETAPS 2019. It was the 8th event in the VerifyThis competition series. In this entry, we present polished and completed versions of our solutions that we created during the competition.

Contents

1	Challenge 1.A	1
1.1	Implementation	1
1.2	Termination	2
1.3	Correctness	2
1.3.1	Property 1: The Exact Sequence is Covered	2
1.3.2	Property 2: Monotonicity	2
1.3.3	Property 3: Maximality	3
1.3.4	Equivalent Formulation Over Indexes	4
2	Challenge 1.B	6
2.1	Merging Two Segments	6
2.2	Merging a List of Segments	6
2.3	GHC-Sort	7
2.4	Correctness Lemmas	7
2.5	Executable Code	8
3	Challenge 2.A	8
3.1	Specification	8
3.2	Auxiliary Theory	8
3.2.1	Has-Left and The-Left	9
3.2.2	Derived Stack	9
3.3	Abstract Implementation	10
3.4	Correctness Proof	10
3.4.1	Popping From the Stack	10
3.4.2	Main Algorithm	11

3.5	Implementation With Arrays	11
3.5.1	Implementation of Pop	11
3.5.2	Implementation of Main Algorithm	12
3.5.3	Correctness Theorem for Concrete Algorithm	12
3.6	Code Generation	12
4	Challenge 2.B	13
4.1	Basic Definitions	13
4.2	Specification of the Parent	13
4.3	The Heap Property (Task 2)	13
5	Iterating a Commutative Computation Concurrently	14
5.1	Misc	14
5.2	The Concurrent System	14
6	Challenge 3	16
6.1	Single-Threaded Implementation	17
6.2	Specification	17
6.3	Correctness	17
6.4	Multi-Threaded Implementation	18

1 Challenge 1.A

```
theory Challenge1A
imports Main
begin
```

Problem definition: https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Verify%20This/Challenges%202019/ghc_sort.pdf

1.1 Implementation

We phrase the algorithm as a functional program. Instead of a list of indexes for segment boundaries, we return a list of lists, containing the segments.

We start with auxiliary functions to take the longest increasing/decreasing sequence from the start of the list

```
fun take-incr :: int list => - where
  take-incr [] = []
| take-incr [x] = [x]
| take-incr (x#y#xs) = (if x<y then x#take-incr (y#xs) else [x])
```

```
fun take-decr :: int list => - where
  take-decr [] = []
| take-decr [x] = [x]
```

| *take-decr* ($x\#y\#xs$) = (if $x \geq y$ then $x\#take-decr (y\#xs)$ else $[x]$)

fun *take* **where**

take [] = []

| *take* [x] = [x]

| *take* ($x\#y\#xs$) = (if $x < y$ then *take-incr* ($x\#y\#xs$) else *take-decr* ($x\#y\#xs$))

definition *take2* $xs \equiv$ let $l = take\ xs$ in ($l, drop\ (length\ l)\ xs$)

— Splits of a longest increasing/decreasing sequence from the list

The main algorithm then iterates until the whole input list is split

function *cuts* **where**

cuts $xs =$ (if $xs = []$ then [] else let $(c, xs) = take2\ xs$ in $c\#cuts\ xs$)

 ⟨*proof*⟩

1.2 Termination

First, we show termination. This will give us induction and proper unfolding lemmas.

lemma *take-non-empty*:

take $xs \neq []$ if $xs \neq []$

 ⟨*proof*⟩

termination

 ⟨*proof*⟩

declare *cuts.simps*[*simp del*]

1.3 Correctness

1.3.1 Property 1: The Exact Sequence is Covered

lemma *tdconc*: $\exists ys. xs = take-decr\ xs\ @\ ys$

 ⟨*proof*⟩

lemma *ticonc*: $\exists ys. xs = take-incr\ xs\ @\ ys$

 ⟨*proof*⟩

lemma *take-conc*: $\exists ys. xs = take\ xs\ @\ ys$

 ⟨*proof*⟩

theorem *concat-cuts*: $concat\ (cuts\ xs) = xs$

 ⟨*proof*⟩

1.3.2 Property 2: Monotonicity

We define constants to specify increasing/decreasing sequences.

```

fun incr where
  incr []  $\longleftrightarrow$  True
| incr [-]  $\longleftrightarrow$  True
| incr (x#y#xs)  $\longleftrightarrow$  x < y  $\wedge$  incr (y#xs)

```

```

fun decr where
  decr []  $\longleftrightarrow$  True
| decr [-]  $\longleftrightarrow$  True
| decr (x#y#xs)  $\longleftrightarrow$  x  $\geq$  y  $\wedge$  decr (y#xs)

```

lemma *tki*: incr (take-incr xs)
 <proof>

lemma *tkd*: decr (take-decr xs)
 <proof>

lemma *icod*: incr (take xs) \vee decr (take xs)
 <proof>

theorem *cuts-incr-decr*: $\forall c \in \text{set } (cuts\ xs).$ incr c \vee decr c
 <proof>

1.3.3 Property 3: Maximality

Specification of a cut that consists of maximal segments: The segments are non-empty, and for every two neighbouring segments, the first value of the last segment cannot be used to continue the first segment:

```

fun maxi where
  maxi []  $\longleftrightarrow$  True
| maxi [c]  $\longleftrightarrow$  c  $\neq$  []
| maxi (c1#c2#cs)  $\longleftrightarrow$  (c1  $\neq$  []  $\wedge$  c2  $\neq$  []  $\wedge$  maxi (c2#cs)  $\wedge$  (
  incr c1  $\wedge$   $\neg$ (last c1 < hd c2)
   $\vee$  decr c1  $\wedge$   $\neg$ (last c1  $\geq$  hd c2)
  ))

```

Obviously, our specification implies that there are no empty segments

lemma *maxi-imp-non-empty*: maxi xs \implies [] \notin set xs
 <proof>

lemma *tdconc'*: xs \neq [] \implies
 \exists ys. xs = take-decr xs @ ys \wedge (ys \neq []
 \longrightarrow \neg (last (take-decr xs) \geq hd ys))
 <proof>

lemma *ticonc'*: xs \neq [] \implies \exists ys. xs = take-incr xs @ ys \wedge (ys \neq [] \longrightarrow \neg (last (take-incr xs) < hd ys))
 <proof>

lemma *take-conc'*: $xs \neq [] \implies \exists ys. xs = take\ xs@ys \wedge (ys \neq [] \longrightarrow ($
 $take\ xs = take\text{-}incr\ xs \wedge \neg(last\ (take\text{-}incr\ xs) < hd\ ys)$
 $\vee take\ xs = take\text{-}decr\ xs \wedge \neg(last\ (take\text{-}decr\ xs) \geq hd\ ys)$
 $))$
 $\langle proof \rangle$

lemma *take-decr-non-empty*:
 $take\text{-}decr\ xs \neq []$ **if** $xs \neq []$
 $\langle proof \rangle$

lemma *take-incr-non-empty*:
 $take\text{-}incr\ xs \neq []$ **if** $xs \neq []$
 $\langle proof \rangle$

lemma *take-conc''*: $xs \neq [] \implies \exists ys. xs = take\ xs@ys \wedge (ys \neq [] \longrightarrow ($
 $incr\ (take\ xs) \wedge \neg(last\ (take\ xs) < hd\ ys)$
 $\vee decr\ (take\ xs) \wedge \neg(last\ (take\ xs) \geq hd\ ys)$
 $))$
 $\langle proof \rangle$

lemma [*simp*]: $cuts\ [] = []$
 $\langle proof \rangle$

lemma [*simp*]: $cuts\ xs \neq [] \longleftrightarrow xs \neq []$
 $\langle proof \rangle$

lemma *inv-cuts*: $cuts\ xs = c\#cs \implies \exists ys. c = take\ xs \wedge xs = c@ys \wedge cs = cuts\ ys$
 $\langle proof \rangle$

theorem *maximal-cuts*: $maxi\ (cuts\ xs)$
 $\langle proof \rangle$

1.3.4 Equivalent Formulation Over Indexes

After the competition, we got the comment that a specification of monotonic sequences via indexes might be more readable.

We show that our functional specification is equivalent to a specification over indexes.

fun *ii-induction* **where**
 $ii\text{-}induction\ [] = ()$
 $| ii\text{-}induction\ [-] = ()$
 $| ii\text{-}induction\ (-\#y\#xs) = ii\text{-}induction\ (y\#xs)$

locale *cnvSpec* =

```

fixes fP P
assumes [simp]: fP []  $\longleftrightarrow$  True
assumes [simp]: fP [x]  $\longleftrightarrow$  True
assumes [simp]: fP (a#b#xs)  $\longleftrightarrow$  P a b  $\wedge$  fP (b#xs)
begin

  lemma idx-spec: fP xs  $\longleftrightarrow$  ( $\forall i < \text{length } xs - 1. P (xs!i) (xs!Suc i)$ )
     $\langle$ proof $\rangle$ 

end

locale cnvSpec' =
  fixes fP P P'
  assumes [simp]: fP []  $\longleftrightarrow$  True
  assumes [simp]: fP [x]  $\longleftrightarrow$  P' x
  assumes [simp]: fP (a#b#xs)  $\longleftrightarrow$  P' a  $\wedge$  P' b  $\wedge$  P a b  $\wedge$  fP (b#xs)
begin

  lemma idx-spec: fP xs  $\longleftrightarrow$  ( $\forall i < \text{length } xs. P' (xs!i)$ )  $\wedge$  ( $\forall i < \text{length } xs - 1. P$ 
(xs!i) (xs!Suc i))
     $\langle$ proof $\rangle$ 

end

interpretation INCR: cnvSpec incr ( $<$ )
   $\langle$ proof $\rangle$ 

interpretation DECR: cnvSpec decr ( $\geq$ )
   $\langle$ proof $\rangle$ 

interpretation MAXI: cnvSpec' maxi  $\lambda c1 c2. ( ($ 
  incr c1  $\wedge$   $\neg(\text{last } c1 < \text{hd } c2)$ 
   $\vee$  decr c1  $\wedge$   $\neg(\text{last } c1 \geq \text{hd } c2)$ 
   $\rangle\rangle$ 
   $\lambda x. x \neq []$ 
   $\langle$ proof $\rangle$ 

lemma incr-by-idx: incr xs = ( $\forall i < \text{length } xs - 1. xs ! i < xs ! Suc i$ )
   $\langle$ proof $\rangle$ 

lemma decr-by-idx: decr xs = ( $\forall i < \text{length } xs - 1. xs ! i \geq xs ! Suc i$ )
   $\langle$ proof $\rangle$ 

lemma maxi-by-idx: maxi xs  $\longleftrightarrow$ 
  ( $\forall i < \text{length } xs. xs ! i \neq []$ )  $\wedge$ 
  ( $\forall i < \text{length } xs - 1.$ 
    incr (xs ! i)  $\wedge$   $\neg \text{last } (xs ! i) < \text{hd } (xs ! Suc i)$ 
     $\vee$  decr (xs ! i)  $\wedge$   $\neg \text{hd } (xs ! Suc i) \leq \text{last } (xs ! i)$ 
  )

```

<proof>

theorem *all-correct:*

concat (cuts xs) = xs
 $\forall c \in \text{set } (\text{cuts } xs). \text{incr } c \vee \text{decr } c$
maxi (cuts xs)
 $\square \notin \text{set } (\text{cuts } xs)$
<proof>

end

2 Challenge 1.B

theory *Challenge1B*

imports *Challenge1A HOL-Library.Multiset*
begin

lemma *mset-concat:*

mset (concat xs) = fold (+) (map mset xs) {#}
<proof>

2.1 Merging Two Segments

fun *merge* :: 'a::{linorder} list \Rightarrow 'a list \Rightarrow 'a list **where**

merge [] *l2* = *l2*
| *merge* *l1* [] = *l1*
| *merge* (*x1* # *l1*) (*x2* # *l2*) =
 (*if* (*x1* < *x2*) *then* *x1* # (*merge* *l1* (*x2* # *l2*)) *else* *x2* # (*merge* (*x1* # *l1*) *l2*))

lemma *merge-correct:*

assumes *sorted l1*

assumes *sorted l2*

shows

sorted (merge l1 l2)
 $\wedge \text{mset } (\text{merge } l1 \ l2) = \text{mset } l1 + \text{mset } l2$
 $\wedge \text{set } (\text{merge } l1 \ l2) = \text{set } l1 \cup \text{set } l2$
<proof>

2.2 Merging a List of Segments

function *merge-list* :: 'a::{linorder} list list \Rightarrow 'a list list \Rightarrow 'a list **where**

merge-list [] [] = []
| *merge-list* [] [*l*] = *l*
| *merge-list* (*la* # *acc2*) [] = *merge-list* [] (*la* # *acc2*)
| *merge-list* (*la* # *acc2*) [*l*] = *merge-list* [] (*l* # *la* # *acc2*)
| *merge-list* *acc2* (*l1* # *l2* # *ls*) =
 merge-list ((*merge* *l1* *l2*) # *acc2*) *ls*
<proof>

termination $\langle proof \rangle$

lemma *merge-list-correct*:

assumes $\bigwedge l. l \in set\ ls \implies sorted\ l$

assumes $\bigwedge l. l \in set\ as \implies sorted\ l$

shows

$sorted\ (merge-list\ as\ ls)$

$\wedge mset\ (merge-list\ as\ ls) = mset\ (concat\ (as\ @\ ls))$

$\wedge set\ (merge-list\ as\ ls) = set\ (concat\ (as\ @\ ls))$

$\langle proof \rangle$

2.3 GHC-Sort

definition

$ghc-sort\ xs = merge-list\ []\ (map\ (\lambda ys. if\ decr\ ys\ then\ rev\ ys\ else\ ys)\ (cuts\ xs))$

lemma *decr-sorted*:

assumes *decr xs*

shows $sorted\ (rev\ xs)$

$\langle proof \rangle$

lemma *incr-sorted*:

assumes *incr xs*

shows $sorted\ xs$

$\langle proof \rangle$

lemma *reverse-phase-sorted*:

$\forall ys \in set\ (map\ (\lambda ys. if\ decr\ ys\ then\ rev\ ys\ else\ ys)\ (cuts\ xs)).\ sorted\ ys$

$\langle proof \rangle$

lemma *reverse-phase-elements*:

$set\ (concat\ (map\ (\lambda ys. if\ decr\ ys\ then\ rev\ ys\ else\ ys)\ (cuts\ xs))) = set\ xs$

$\langle proof \rangle$

lemma *reverse-phase-permutation*:

$mset\ (concat\ (map\ (\lambda ys. if\ decr\ ys\ then\ rev\ ys\ else\ ys)\ (cuts\ xs))) = mset\ xs$

$\langle proof \rangle$

2.4 Correctness Lemmas

The result is sorted and a permutation of the original elements.

theorem *sorted-ghc-sort*:

$sorted\ (ghc-sort\ xs)$

$\langle proof \rangle$

theorem *permutation-ghc-sort*:

$mset\ (ghc-sort\ xs) = mset\ xs$

$\langle proof \rangle$

corollary *elements-ghc-sort*: $set (ghc-sort\ xs) = set\ xs$
<proof>

2.5 Executable Code

export-code *ghc-sort* **checking** *SML Scala OCaml? Haskell?*

value [*code*] *ghc-sort* [1,2,7,3,5,6,9,8,4]

end

3 Challenge 2.A

theory *Challenge2A*
imports *lib/VTcomp*
begin

Problem definition: https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Verify%20This/Challenges%202019/cartesian_trees.pdf

Polished and worked-over version.

3.1 Specification

We first fix the input, a list of integers

context *fixes xs :: int list* **begin**

We then specify the desired output: For each index j , return the greatest index $i < j$ such that $xs!i < xs!j$, or *None* if no such index exists.

Note that our indexes start at zero, and we use an option datatype to model that no left-smaller value may exist.

definition

left-spec $j = (if\ (\exists i < j. xs\ !\ i < xs\ !\ j)\ then\ Some\ (GREATEST\ i. i < j \wedge xs\ !\ i < xs\ !\ j)\ else\ None)$

The output of the algorithm should be an array *lf*, containing the indexes of the left-smaller values:

definition *all-left-spec* $lf \equiv length\ lf = length\ xs \wedge (\forall i < length\ xs. lf!i = left-spec\ i)$

3.2 Auxiliary Theory

We derive some theory specific to this algorithm

3.2.1 Has-Left and The-Left

We split the specification of nearest left value into a predicate and a total function

definition *has-left* $j = (\exists i < j. xs ! i < xs ! j)$

definition *the-left* $j = (GREATEST i. i < j \wedge xs ! i < xs ! j)$

lemma *left-alt*: *left-spec* $j = (if\ has-left\ j\ then\ Some\ (the-left\ j)\ else\ None)$
<proof>

lemma *the-leftI*: *has-left* $j \implies the-left\ j < j \wedge xs ! the-left\ j < xs ! j$
<proof>

lemma *the-left-decr[simp]*: *has-left* $i \implies the-left\ i < i$
<proof>

lemma *le-the-leftI*:
assumes $i \leq j\ xs ! i < xs ! j$
shows $i \leq the-left\ j$
<proof>

lemma *the-left-leI*:
assumes $\forall k. j < k \wedge k < i \longrightarrow \neg xs ! k < xs ! i$
assumes *has-left* i
shows $the-left\ i \leq j$
<proof>

3.2.2 Derived Stack

We note that the stack in the algorithm doesn't contain any extra information. It can be derived from the left neighbours that have been computed so far: The first element of the stack is the current index - 1, and each next element is the nearest left smaller value of the previous element:

fun *der-stack* **where**
der-stack $i = (if\ has-left\ i\ then\ the-left\ i\ \#\ der-stack\ (the-left\ i)\ else\ [])$
declare *der-stack.simps[simp del]*

Although the refinement framework would allow us to phrase the algorithm without a stack first, and then introduce the stack in a subsequent refinement step (or omit it altogether), for simplicity of presentation, we decided to model the algorithm with a stack in first place. However, the invariant will account for the stack being derived.

lemma *set-der-stack-lt*: $k \in set\ (der-stack\ i_0) \implies k < i_0$
<proof>

3.3 Abstract Implementation

We first implement the algorithm on lists. The assertions that we annotated into the algorithm ensure that all list index accesses are in bounds.

definition $pop\ stk\ v \equiv dropWhile\ (\lambda j. xs!j \geq v)\ stk$

lemma $pop\ Nil[simp]: pop\ []\ v = []\ \langle proof \rangle$

lemma $pop\ cons: pop\ (j\#\#js)\ v = (if\ xs!j \geq v\ then\ pop\ js\ v\ else\ j\#\#js)\ \langle proof \rangle$

definition $all\ left \equiv doN\ \{$
 $(-,lf) \leftarrow nfoldli\ [0..<length\ xs]\ (\lambda-. True)\ (\lambda i\ (stk,lf). doN\ \{$
 $ASSERT\ (set\ stk \subseteq \{0..<length\ xs\});$
 $let\ stk = pop\ stk\ (xs!i);$
 $ASSERT\ (stk = der-stack\ i);$
 $ASSERT\ (i < length\ lf);$
 $if\ (stk = [])\ then\ doN\ \{$
 $let\ lf = lf[i:=None];$
 $RETURN\ (i\#\#stk,lf)$
 $\}$ $else\ doN\ \{$
 $let\ lf = lf[i:=Some\ (hd\ stk)];$
 $RETURN\ (i\#\#stk,lf)$
 $\}$
 $\})\ ([],replicate\ (length\ xs)\ None);$
 $RETURN\ lf$
 $\}$

3.4 Correctness Proof

3.4.1 Popping From the Stack

We show that the abstract algorithm implements its specification. The main idea here is the popping of the stack. To obtain a left smaller value, it is enough to follow the left-values of the left-neighbour, until we have found the value or there are no more left-values.

The following theorem formalizes this idea:

theorem $find-left-rl:$

assumes $i_0 < length\ xs$

assumes $i < i_0$

assumes $left-spec\ i_0 \leq Some\ i$

shows $if\ xs!i < xs!i_0\ then\ left-spec\ i_0 = Some\ i$
 $else\ left-spec\ i_0 \leq left-spec\ i$

$\langle proof \rangle$

Using this lemma, we can show that the stack popping procedure preserves the form of the stack.

lemma *pop-aux*: $\llbracket k < i_0; i_0 < \text{length } xs; \text{left-spec } i_0 \leq \text{Some } k \rrbracket \implies \text{pop } (k \# \text{der-stack } k) (xs!i_0) = \text{der-stack } i_0$
 ⟨proof⟩

3.4.2 Main Algorithm

Ad-Hoc lemmas

lemma *swap-adhoc*[*simp*]:
 $\text{None} = \text{left } i \longleftrightarrow \text{left } i = \text{None}$
 $\text{Some } j = \text{left } i \longleftrightarrow \text{left } i = \text{Some } j$ ⟨proof⟩

lemma *left-spec-None-iff*[*simp*]: $\text{left-spec } i = \text{None} \longleftrightarrow \neg \text{has-left } i$ ⟨proof⟩

lemma [*simp*]: $\text{left-spec } 0 = \text{None}$ ⟨proof⟩

lemma [*simp*]: $\text{has-left } 0 = \text{False}$
 ⟨proof⟩

lemma [*simp*]: $\text{der-stack } 0 = []$
 ⟨proof⟩

lemma *algo-correct*: $\text{all-left} \leq \text{SPEC all-left-spec}$
 ⟨proof⟩

3.5 Implementation With Arrays

We refine the algorithm to use actual arrays for the input and output. The stack remains a list, as pushing and popping from a (functional) list is efficient.

3.5.1 Implementation of Pop

In a first step, we refine the pop function to an explicit loop.

definition *pop2* *stk* *v* \equiv
monadic-WHILEIT
 $(\lambda-. \text{set } stk \subseteq \{0..<\text{length } xs\})$
 $(\lambda[] \Rightarrow \text{RETURN False} \mid k \# stk \Rightarrow \text{doN } \{ \text{ASSERT } (k < \text{length } xs); \text{RETURN } (v \leq xs!k) \})$
 $(\lambda stk. \text{mop-list-tl } stk)$
stk

lemma *pop2-refine-aux*: $\text{set } stk \subseteq \{0..<\text{length } xs\} \implies \text{pop2 } stk \ v \leq \text{RETURN } (\text{pop } stk \ v)$
 ⟨proof⟩

end — Context fixing the input *xs*.

The refinement lemma written in higher-order form.

lemma *pop2-refine*: $(\text{uncurry2 } \text{pop2}, \text{uncurry2 } (\text{RETURN } \text{ooo } \text{pop})) \in [\lambda((xs,stk),v). \text{set } stk \subseteq \{0..\langle \text{length } xs \rangle\}]_f (Id \times_r Id) \times_r Id \rightarrow \langle Id \rangle \text{nres-rel}$
 $\langle \text{proof} \rangle$

Next, we use the Sepref tool to synthesize an implementation on arrays.

sepref-definition *pop2-impl* is $\text{uncurry2 } \text{pop2} :: (\text{array-assn } id\text{-assn})^k *_a (\text{list-assn } id\text{-assn})^k *_a id\text{-assn}^k \rightarrow_a \text{list-assn } id\text{-assn}$
 $\langle \text{proof} \rangle$

lemmas $[\text{sepref-fr-rules}] = \text{pop2-impl.refine}[\text{FCOMP } \text{pop2-refine}]$

3.5.2 Implementation of Main Algorithm

sepref-definition *all-left-impl* is $\text{all-left} :: (\text{array-assn } id\text{-assn})^k \rightarrow_a \text{array-assn } (\text{option-assn } id\text{-assn})$
 $\langle \text{proof} \rangle$

3.5.3 Correctness Theorem for Concrete Algorithm

We compose the correctness theorem and the refinement theorem, to get a correctness theorem for the final implementation.

Abstract correctness theorem in higher-order form.

lemma *algo-correct'*: $(\text{all-left}, \text{SPEC } o \text{ all-left-spec})$
 $\in \langle Id \rangle \text{list-rel} \rightarrow \langle \langle Id \rangle \text{option-rel} \rangle \text{list-rel} \text{nres-rel}$
 $\langle \text{proof} \rangle$

Main correctness theorem in higher-order form.

theorem *algo-impl-correct*:
 $(\text{all-left-impl}, \text{SPEC } o \text{ all-left-spec})$
 $\in (\text{array-assn } \text{int-assn}, \text{array-assn } \text{int-assn}) \rightarrow_a \text{array-assn } (\text{option-assn } \text{nat-assn})$
 $\langle \text{proof} \rangle$

Main correctness theorem as Hoare-Triple

theorem *algo-impl-correct'*:
 $\langle \text{array-assn } \text{int-assn } xs \text{ } xsi \rangle$
 $\text{all-left-impl } xsi$
 $\langle \lambda lfi. \exists_A lf. \text{array-assn } \text{int-assn } xs \text{ } xsi$
 $\quad * \text{array-assn } (\text{option-assn } id\text{-assn}) \text{ } lf \text{ } lfi$
 $\quad * \uparrow(\text{all-left-spec } xs \text{ } lf) \rangle_t$
 $\langle \text{proof} \rangle$

3.6 Code Generation

export-code *all-left-impl checking SML Scala Haskell? OCaml?*

The example from the problem description, in ML using the verified algorithm

$\langle ML \rangle$

end

4 Challenge 2.B

```
theory Challenge2B
  imports Challenge2A
begin
```

We did not get very far on this part of the competition. Only Task 2 was finished.

4.1 Basic Definitions

```
datatype tree = Leaf | Node int (lc: tree) (rc: tree)
```

Analogous to *left-spec* from 2.A.

definition

```
  right-spec xs j =
    (if ( $\exists i > j. xs ! i < xs ! j$ ) then Some (LEAST i.  $i > j \wedge xs ! i < xs ! j$ ) else
      None)
```

context

```
  fixes xs :: int list
  assumes distinct xs
begin
```

4.2 Specification of the Parent

definition

```
  parent i = (
    case (left-spec xs i, right-spec xs i) of
      (None, None)  $\Rightarrow$  None
    | (Some x, None)  $\Rightarrow$  Some x
    | (None, Some y)  $\Rightarrow$  Some y
    | (Some x, Some y)  $\Rightarrow$  Some (max x y)
  )
```

4.3 The Heap Property (Task 2)

lemma parent-heap:

```
  assumes parent j = Some p
  shows xs ! j > xs ! p
 $\langle proof \rangle$ 
```

end

end

5 Iterating a Commutative Computation Concurrently

```
theory Parallel-Multiset-Fold  
  imports HOL-Library.Multiset  
begin
```

This theory formalizes a deep embedding of a simple parallel computation model. In this model, we formalize a computation scheme to execute a fold-function over a commutative operation concurrently, and prove it correct.

5.1 Misc

```
lemma (in comp-fun-commute) fold-mset-rewr: fold-mset f a (mset l) = fold f l a  
  <proof>
```

```
lemma finite-set-of-finite-maps:  
  fixes A :: 'a set  
    and B :: 'b set  
  assumes finite A  
    and finite B  
  shows finite {m. dom m  $\subseteq$  A  $\wedge$  ran m  $\subseteq$  B}  
  <proof>
```

```
lemma wf-rtranclp-ev-induct[consumes 1, case-names step]:  
  assumes wf {(x, y). R y x} and step:  $\bigwedge x. R^{**} a x \implies P x \vee (\exists y. R x y)$   
  shows  $\exists x. P x \wedge R^{**} a x$   
  <proof>
```

5.2 The Concurrent System

A state of our concurrent systems consists of a list of tasks, a partial map from threads to the task they are currently working on, and the current computation result.

```
type-synonym ('a, 's) state = 'a list  $\times$  (nat  $\rightarrow$  'a)  $\times$  's
```

```
context comp-fun-commute  
begin
```

```
context  
  fixes n :: nat — The number of threads.  
  assumes n-gt-0[simp, intro]: n > 0  
begin
```

A state is *final* if there are no remaining tasks and if all workers have finished their work.

```
definition
```

$final \equiv \lambda(ts, ws, r). ts = [] \wedge dom\ ws \cap \{0..<n\} = \{\}$

At any point a thread can:

- pick a new task from the queue if it is currently not busy
- or execute its current task.

inductive $step :: ('a, 'b) state \Rightarrow ('a, 'b) state \Rightarrow bool$ **where**

$pick: step\ (t \# ts, ws, s)\ (ts, ws(i := Some\ t), s)$ **if** $ws\ i = None$ **and** $i < n$
 $| exec: step\ (ts, ws, s)\ (ts, ws(i := None), f\ a\ s)$ **if** $ws\ i = Some\ a$ **and** $i < n$

lemma *no-deadlock*:

assumes $\neg final\ cfg$

shows $\exists\ cfg'. step\ cfg\ cfg'$

<proof>

lemma *wf-step*:

$wf\ \{((ts', ws', r'), (ts, ws, r))\}.$

$step\ (ts, ws, r)\ (ts', ws', r') \wedge set\ ts' \subseteq S \wedge dom\ ws \subseteq \{0..<n\} \wedge ran\ ws \subseteq S$

if *finite* S

<proof>

context

fixes $ts :: 'a\ list$ **and** $start :: 'b$

begin

definition

$s_0 = (ts, \lambda-. None, start)$

definition *reachable* $\equiv (step^{**})\ s_0$

lemma *reachable0[simp]*: *reachable* s_0

<proof>

definition *is-invar* $I \equiv I\ s_0 \wedge (\forall s\ s'. reachable\ s \wedge I\ s \wedge step\ s\ s' \longrightarrow I\ s')$

lemma *is-invarI[intro?]*:

$\llbracket I\ s_0; \bigwedge s\ s'. \llbracket reachable\ s; I\ s; step\ s\ s' \rrbracket \Longrightarrow I\ s' \rrbracket \Longrightarrow is-invar\ I$

<proof>

lemma *invar-reachable*: *is-invar* $I \Longrightarrow reachable\ s \Longrightarrow I\ s$

<proof>

definition

$invar \equiv \lambda(ts2, ws, r).$

$(\exists\ ts1.$

$mset\ ts = ts1 + \{\#\ the\ (ws\ i). i \in \#\ mset-set\ (dom\ ws \cap \{0..<n\})\ \#\} +$
 $mset\ ts2$

$\wedge r = \text{fold-mset } f \text{ start } ts1$
 $\wedge \text{set } ts2 \subseteq \text{set } ts \wedge \text{ran } ws \subseteq \text{set } ts \wedge \text{dom } ws \subseteq \{0..<n\}$)

lemma *invariant*:

is-invar invar

<proof>

lemma *final-state-correct1*:

assumes *invar (ts', ms, r) final (ts', ms, r)*

shows *r = fold-mset f start (mset ts)*

<proof>

lemma *final-state-correct2*:

assumes *reachable (ts', ms, r) final (ts', ms, r)*

shows *r = fold-mset f start (mset ts)*

<proof>

Soundness: whenever we reach a final state, the computation result is correct.

theorem *final-state-correct*:

assumes *reachable (ts', ms, r) final (ts', ms, r)*

shows *r = fold f ts start*

<proof>

Termination: at any point during the program execution, we can continue to a final state. That is, the computation always terminates.

theorem *termination*:

assumes *reachable s*

shows $\exists s'. \text{final } s' \wedge \text{step}^{**} s s'$

<proof>

end

end

end

The main theorems outside the locale:

thm *comp-fun-commute.final-state-correct comp-fun-commute.termination*

end

6 Challenge 3

theory *Challenge3*

imports *Parallel-Multiset-Fold Refine-Imperative-HOL.IICF*

begin

Problem definition: <https://ethz.ch/content/dam/ethz/special-interest/infk/>

6.1 Single-Threaded Implementation

We define type synonyms for values (which we fix to integers here) and triplets, which are a pair of coordinates and a value.

type-synonym $val = int$

type-synonym $triplet = (nat \times nat) \times val$

We fix a size n for the vector.

context

fixes $n :: nat$

begin

An algorithm finishing triples in any order.

definition

$alg (ts :: triplet\ list) x = fold-mset (\lambda((r,c),v) y. y(c:=y\ c + x\ r * v)) (\lambda-. 0 :: int) (mset\ ts)$

We show that the folding function is commutative, i.e., the order of the folding does not matter. We will use this below to show that the computation can be parallelized.

interpretation $comp-fun-commute (\lambda((r, c), v) y. y(c := (y\ c :: val) + x\ r * v))$

$\langle proof \rangle$

6.2 Specification

Abstraction function, mapping a sparse matrix to a function from coordinates to values.

definition $\alpha :: triplet\ list \Rightarrow (nat \times nat) \Rightarrow val$ **where**

$\alpha = the-default\ 0\ oo\ map-of$

Abstract product.

definition $pr\ m\ x\ i \equiv \sum_{k=0..<n}. x\ k * m\ (k, i)$

6.3 Correctness

lemma $aux:$

$distinct (map\ fst (ts1@ts2)) \implies$
 $the-default (0::val) (case\ map-of\ ts1\ (k, i)\ of\ None \Rightarrow map-of\ ts2\ (k, i) \mid Some\ x \Rightarrow Some\ x)$

$= the-default\ 0 (map-of\ ts1\ (k, i)) + the-default\ 0 (map-of\ ts2\ (k, i))$

$\langle \text{proof} \rangle$

lemma 1 [simp]: $\text{distinct } (\text{map fst } (ts1 @ ts2)) \implies$
 $\text{pr } (\alpha (ts1 @ ts2)) \ x \ i = \text{pr } (\alpha \ ts1) \ x \ i + \text{pr } (\alpha \ ts2) \ x \ i$
 $\langle \text{proof} \rangle$

lemmas 2 = 1 [of $[(r,c),v]$ ts , simplified] **for** $r \ c \ v \ ts$

lemma [simp]: $\alpha \ [] = (\lambda \cdot. 0)$ $\langle \text{proof} \rangle$

lemma [simp]: $\text{pr } (\lambda \cdot. 0 :: \text{val}) \ x = (\lambda \cdot. 0)$
 $\langle \text{proof} \rangle$

lemma aux3: *the-default 0 (if b then Some x else None) = (if b then x else 0)*
 $\langle \text{proof} \rangle$

lemma correct-aux: $\llbracket \text{distinct } (\text{map fst } ts); \forall ((r,c),-) \in \text{set } ts. r < n \rrbracket$
 $\implies \forall i. \text{fold } (\lambda((r,c),v) \ y. y(c:=y \ c + x \ r * v)) \ ts \ m \ i = m \ i + \text{pr } (\alpha \ ts) \ x \ i$
 $\langle \text{proof} \rangle$

lemma correct-fold:

assumes $\text{distinct } (\text{map fst } ts)$

assumes $\forall ((r,c),-) \in \text{set } ts. r < n$

shows $\text{fold } (\lambda((r,c),v) \ y. y(c:=y \ c + x \ r * v)) \ ts \ (\lambda \cdot. 0) = \text{pr } (\alpha \ ts) \ x$
 $\langle \text{proof} \rangle$

lemma alg-by-fold: $\text{alg } ts \ x = \text{fold } (\lambda((r,c),v) \ y. y(c:=y \ c + x \ r * v)) \ ts \ (\lambda \cdot. 0)$

$\langle \text{proof} \rangle$

theorem correct:

assumes $\text{distinct } (\text{map fst } ts)$

assumes $\forall ((r,c),-) \in \text{set } ts. r < n$

shows $\text{alg } ts \ x = \text{pr } (\alpha \ ts) \ x$

$\langle \text{proof} \rangle$

6.4 Multi-Threaded Implementation

Correctness of the parallel implementation:

theorem parallel-correct:

assumes $\text{distinct } (\text{map fst } ts) \ \forall ((r,c),-) \in \text{set } ts. r < n$

and $0 < n$ — At least on thread

— We have reached a final state.

and $\text{reachable } x \ n \ ts \ (\lambda \cdot. 0) \ (ts', ms, r) \ \text{final } n \ (ts', ms, r)$

shows $r = \text{pr } (\alpha \ ts) \ x$

$\langle \text{proof} \rangle$

We also know that the computation will always terminate.

theorem *parallel-termination*:

assumes $0 < n$

and *reachable* $x\ n\ ts\ (\lambda-. 0)\ s$

shows $\exists s'. \text{final } n\ s' \wedge (\text{step } x\ n)^{**}\ s\ s'$

<proof>

end — Context for fixed n .

end