VerifyThis 2019 – Polished Isabelle Solutions

Peter Lammich Simon Wimmer

March 17, 2025

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

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

fun *take* **where**
take [] = []
| *take* [*x*] = [*x*]
| *take* $(x\#y\#xs) = (if\ x<y\ then\ take\text{-}incr\ (x\#y\#xs)\ else\ take\text{-}decr\ (x\#y\#xs))$

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**
$incr\ []\ \longleftrightarrow\ True$
$|\ incr\ [\text{-}]\ \longleftrightarrow\ True$
$|\ incr\ (x\#y\#xs)\ \longleftrightarrow\ x<y \land incr\ (y\#xs)$

fun *decr* **where**
$decr\ []\ \longleftrightarrow\ True$
$|\ decr\ [\text{-}]\ \longleftrightarrow\ True$
$|\ decr\ (x\#y\#xs)\ \longleftrightarrow\ x\geq y \land decr\ (y\#xs)$

lemma *tki*: $incr\ (take\text{-}incr\ xs)$
⟨*proof*⟩

lemma *tkd*: $decr\ (take\text{-}decr\ xs)$
⟨*proof*⟩

lemma *icod*: $incr\ (take\ xs) \lor decr\ (take\ xs)$
⟨*proof*⟩

theorem *cuts-incr-decr*: $\forall\, c\in set\ (cuts\ xs).\ incr\ c \lor decr\ c$
⟨*proof*⟩

1.3.3 Property 3: Maximality

Specification of a cut that consists of maximal segments: The segements 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**
$maxi\ []\ \longleftrightarrow\ True$
$|\ maxi\ [c]\ \longleftrightarrow\ c\neq[]$
$|\ maxi\ (c1\#c2\#cs)\ \longleftrightarrow\ (c1\neq[] \land c2\neq[] \land maxi\ (c2\#cs) \land ($
$incr\ c1 \land \neg(last\ c1\ <\ hd\ c2)$
$\lor\ decr\ c1 \land \neg(last\ c1\ \geq\ hd\ c2)$
$))$

Obviously, our specification implies that there are no empty segments

lemma *maxi-imp-non-empty*: $maxi\ xs \Longrightarrow []\notin set\ xs$
⟨*proof*⟩

lemma *tdconc'*: $xs\neq[] \Longrightarrow$
$\exists\, ys.\ xs = take\text{-}decr\ xs\ @\ ys \land (ys\neq[]$
$\longrightarrow \neg(last\ (take\text{-}decr\ xs) \geq hd\ ys))$
⟨*proof*⟩

lemma *ticonc'*: $xs\neq[] \Longrightarrow \exists\, ys.\ xs = take\text{-}incr\ xs\ @\ ys \land (ys\neq[] \longrightarrow \neg(last\ (take\text{-}incr\ xs) < hd\ ys))$
⟨*proof*⟩

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

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

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

lemma *take-conc′′*: $xs \neq [] \Longrightarrow \exists ys.\ xs = take\ xs@ys \wedge (ys \neq [] \longrightarrow ($
$incr\ (take\ xs) \wedge \neg(last\ (take\ xs) < hd\ ys)$
$\vee\ decr\ (take\ xs) \wedge \neg(last\ (take\ xs) \geq hd\ ys)$
$))$
⟨*proof*⟩

lemma [*simp*]: $cuts\ [] = []$
⟨*proof*⟩

lemma [*simp*]: $cuts\ xs \neq [] \longleftrightarrow xs \neq []$
⟨*proof*⟩

lemma *inv-cuts*: $cuts\ xs = c\#cs \Longrightarrow \exists ys.\ c = take\ xs \wedge xs = c@ys \wedge cs = cuts\ ys$
⟨*proof*⟩

theorem *maximal-cuts*: $maxi\ (cuts\ xs)$
⟨*proof*⟩

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

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*: $fP\ xs\longleftrightarrow(\forall\, i{<}length\ xs - 1.\ P\ (xs!i)\ (xs!Suc\ i))$
$\langle proof\rangle$

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

end

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

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

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

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

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

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

⟨*proof*⟩

theorem *all-correct*:
concat (*cuts xs*) = *xs*
∀ *c*∈*set* (*cuts xs*). *incr c* ∨ *decr c*
maxi (*cuts xs*)
[] ∉ *set* (*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* ⇒ ′*a list* ⇒ ′*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*)
∧ *mset* (*merge l1 l2*) = *mset l1* + *mset l2*
∧ *set* (*merge l1 l2*) = *set l1* ∪ *set l2*
⟨*proof*⟩

2.2 Merging a List of Segments

function *merge-list* :: ′*a*::{*linorder*} *list list* ⇒ ′*a list list* ⇒ ′*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 ⟨*proof*⟩

lemma *merge-list-correct*:
assumes $\bigwedge l.\ l \in set\ ls \Longrightarrow sorted\ l$
assumes $\bigwedge l.\ l \in set\ as \Longrightarrow sorted\ l$
shows
$sorted\ (merge\text{-}list\ as\ ls)$
$\wedge\ mset\ (merge\text{-}list\ as\ ls) = mset\ (concat\ (as\ @\ ls))$
$\wedge\ set\ (merge\text{-}list\ as\ ls) = set\ (concat\ (as\ @\ ls))$
⟨*proof*⟩

2.3 GHC-Sort

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

lemma *decr-sorted*:
assumes $decr\ xs$
shows $sorted\ (rev\ xs)$
⟨*proof*⟩

lemma *incr-sorted*:
assumes $incr\ xs$
shows $sorted\ xs$
⟨*proof*⟩

lemma *reverse-phase-sorted*:
$\forall\, ys \in set\ (map\ (\lambda ys.\ if\ decr\ ys\ then\ rev\ ys\ else\ ys)\ (cuts\ xs)).\ sorted\ ys$
⟨*proof*⟩

lemma *reverse-phase-elements*:
$set\ (concat\ (map\ (\lambda ys.\ if\ decr\ ys\ then\ rev\ ys\ else\ ys)\ (cuts\ xs))) = set\ xs$
⟨*proof*⟩

lemma *reverse-phase-permutation*:
$mset\ (concat\ (map\ (\lambda ys.\ if\ decr\ ys\ then\ rev\ ys\ else\ ys)\ (cuts\ xs))) = mset\ xs$
⟨*proof*⟩

2.4 Correctness Lemmas

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

theorem *sorted-ghc-sort*:
$sorted\ (ghc\text{-}sort\ xs)$
⟨*proof*⟩

theorem *permutation-ghc-sort*:
$mset\ (ghc\text{-}sort\ xs) = mset\ xs$
⟨*proof*⟩

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 exists.

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*)
$\langle proof \rangle$

lemma *the-leftI*: *has-left* $j \Longrightarrow$ *the-left* $j < j \wedge xs!$*the-left* $j < xs!j$
$\langle proof \rangle$

lemma *the-left-decr*[*simp*]: *has-left* $i \Longrightarrow$ *the-left* $i < i$
$\langle proof \rangle$

lemma *le-the-leftI*:
assumes $i{\le}j$ $xs!i < xs!j$
shows $i \le$ *the-left* j
$\langle proof \rangle$

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* $i \le j$
$\langle proof \rangle$

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) $\Longrightarrow k{<}i_0$
$\langle proof \rangle$

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* ≡ *dropWhile* (λ*j. xs*!*j*≥*v*) *stk*

lemma *pop-Nil*[*simp*]: *pop* [] *v* = [] ⟨*proof*⟩
lemma *pop-cons*: *pop* (*j*#*js*) *v* = (*if xs*!*j* ≥ *v then pop js v else j*#*js*)
⟨*proof*⟩

```
definition all-left ≡ doN {
  (-,lf) ← nfoldli [0..<length xs] (λ-. True) (λi (stk,lf). doN {
    ASSERT (set stk ⊆ {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. Top 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 ≤ *Some i*
shows *if xs*!*i* < *xs*!i_0 *then left-spec* i_0 = *Some i*
else left-spec i_0 ≤ *left-spec i*
⟨*proof*⟩

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

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

3.4.2 Main Algorithm

Ad-Hoc lemmas

lemma *swap-adhoc*[*simp*]:
None = *left i* ⟷ *left i* = *None*
Some j = *left i* ⟷ *left i* = *Some j* ⟨*proof*⟩

lemma *left-spec-None-iff*[*simp*]: *left-spec i* = *None* ⟷ ¬*has-left i* ⟨*proof*⟩
lemma [*simp*]: *left-spec 0* = *None* ⟨*proof*⟩
lemma [*simp*]: *has-left 0* = *False*
⟨*proof*⟩
lemma [*simp*]: *der-stack 0* = []
⟨*proof*⟩

lemma *algo-correct*: *all-left* ≤ *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* ≡
monadic-WHILEIT
(λ-. *set stk* ⊆ {*0*..<*length xs*})
(λ[] ⇒ *RETURN False* | *k*#*stk* ⇒ *doN* { *ASSERT* (*k*<*length xs*); *RETURN* (*v* ≤ *xs*!*k*) })
(λ*stk*. *mop-list-tl stk*)
stk

lemma *pop2-refine-aux*: *set stk* ⊆ {*0*..<*length xs*} ⟹ *pop2 stk v* ≤ *RETURN* (*pop stk v*)
⟨*proof*⟩

end — Context fixing the input *xs*.

The refinement lemma written in higher-order form.

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

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

sepref-definition *pop2-impl* **is** $uncurry2\ pop2 :: (array\text{-}assn\ id\text{-}assn)^k *_a (list\text{-}assn\ id\text{-}assn)^k *_a id\text{-}assn^k \rightarrow_a list\text{-}assn\ id\text{-}assn$
  ⟨*proof*⟩
lemmas [*sepref-fr-rules*] = *pop2-impl.refine*[*FCOMP pop2-refine*]

3.5.2 Implementation of Main Algorithm

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

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

Main correctness theorem in higher-order form.

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

  ⟨*proof*⟩

Main correctness theorem as Hoare-Triple

theorem *algo-impl-correct'*:
  $<array\text{-}assn\ int\text{-}assn\ xs\ xsi>$
    all-left-impl xsi
  $<\lambda lfi.\ \exists_A lf.\ array\text{-}assn\ int\text{-}assn\ xs\ xsi$
      $*\ array\text{-}assn\ (option\text{-}assn\ id\text{-}assn)\ lf\ lfi$
      $*\ \uparrow(all\text{-}left\text{-}spec\ xs\ lf)>_t$
  ⟨*proof*⟩

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

⟨*ML*⟩

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*
⟨*proof*⟩

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*: $\mathit{fold\text{-}mset}\ f\ a\ (\mathit{mset}\ l) = \mathit{fold}\ f\ l\ a$
  ⟨*proof*⟩

lemma *finite-set-of-finite-maps*:
  fixes $A :: {'a}\ \mathit{set}$
    and $B :: {'b}\ \mathit{set}$
  assumes *finite* A
    and *finite* B
  shows *finite* $\{m.\ \mathit{dom}\ m \subseteq A \wedge \mathit{ran}\ m \subseteq B\}$
⟨*proof*⟩

lemma *wf-rtranclp-ev-induct*[*consumes 1*, *case-names step*]:
  assumes $\mathit{wf}\ \{(x,\ y).\ R\ y\ x\}$ **and** *step*: $\bigwedge x.\ R^{**}\ a\ x \Longrightarrow P\ x \vee (\exists\ y.\ R\ x\ y)$
  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 $({'a},\ {'s})\ \mathit{state} = {'a}\ \mathit{list} \times (\mathit{nat} \rightharpoonup {'a}) \times {'s}$

context *comp-fun-commute*
begin

context
  fixes $n :: \mathit{nat}$ — 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* ⇒ ('*a*, '*b*) *state* ⇒ *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 ¬ *final cfg*
  shows ∃ *cfg*'. *step cfg cfg*'
  ⟨*proof*⟩

lemma *wf-step*:
  wf {((*ts*', *ws*', *r*'), (*ts*, *ws*, *r*)).
    step (*ts*, *ws*, *r*) (*ts*', *ws*', *r*') ∧ *set ts*' ⊆ *S* ∧ *dom ws* ⊆ {*0*..<*n*} ∧ *ran ws* ⊆ *S*}
  if *finite S*
⟨*proof*⟩

context
  fixes *ts* :: '*a list* **and** *start* :: '*b*
begin

definition
  s_0 = (*ts*, λ-. *None*, *start*)

definition *reachable* ≡ (*step***) s_0

lemma *reachable0*[*simp*]: *reachable* s_0
  ⟨*proof*⟩

definition *is-invar I* ≡ *I* s_0 ∧ (∀ *s s*'. *reachable s* ∧ *I s* ∧ *step s s*' ⟶ *I s*')

lemma *is-invarI*[*intro?*]:
  ⟦ *I* s_0; ⋀*s s*'. ⟦ *reachable s*; *I s*; *step s s*'⟧ ⟹ *I s*' ⟧ ⟹ *is-invar I*
  ⟨*proof*⟩

lemma *invar-reachable*: *is-invar I* ⟹ *reachable s* ⟹ *I s*
  ⟨*proof*⟩

definition
  invar ≡ λ(*ts2*, *ws*, *r*).
    (∃ *ts1*.
      mset ts = *ts1* + {# *the* (*ws i*). *i* ∈# *mset-set* (*dom ws* ∩ {*0*..<*n*}) #} +
mset ts2

$\wedge$ r = *fold-mset f start ts1*
$\wedge$ *set ts2* $\subseteq$ *set ts* $\wedge$ *ran ws* $\subseteq$ *set ts* $\wedge$ *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 $\exists s'$. *final* s' $\wedge$ *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/

chair-program-method/pm/documents/Verify%20This/Challenges%202019/sparse_matrix_multiplication.pdf

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* × *nat*) × *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* (λ((*r*,*c*),*v*) *y*. *y*(*c*:=*y c* + *x r* ∗ *v*)) (λ-. *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* (λ((*r*, *c*), *v*) *y*. *y*(*c* := (*y c* :: *val*) + *x r* ∗ *v*))
⟨*proof*⟩

6.2 Specification

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

definition α :: *triplet list* ⇒ (*nat* × *nat*) ⇒ *val* **where**
α = *the-default 0 oo map-of*

Abstract product.

definition *pr m x i* ≡ ∑ *k*=*0*..<*n*. *x k* ∗ *m* (*k*, *i*)

6.3 Correctness

lemma *aux*:

distinct (*map fst* (*ts1*@*ts2*)) ⟹
the-default (*0*::*val*) (*case map-of ts1* (*k*, *i*) *of None* ⇒ *map-of ts2* (*k*, *i*) | *Some x* ⇒ *Some x*)

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

⟨*proof*⟩

lemma *1*[*simp*]: *distinct* (*map fst* (*ts1*@*ts2*)) $\Longrightarrow$
pr (α (*ts1*@*ts2*)) *x i* = *pr* (α *ts1*) *x i* + *pr* (α *ts2*) *x i*
⟨*proof*⟩

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

lemma [*simp*]: α [] = (λ-. *0*) ⟨*proof*⟩

lemma [*simp*]: *pr* (λ-. *0*::*val*) *x* = (λ-. *0*)
⟨*proof*⟩

lemma *aux3*: *the-default 0* (*if b then Some x else None*) = (*if b then x else 0*)
⟨*proof*⟩

lemma *correct-aux*: ⟦*distinct* (*map fst ts*); ∀ ((*r*,*c*),-)∈*set ts*. *r*<*n*⟧
$\Longrightarrow$ ∀ *i*. *fold* (λ((*r*,*c*),*v*) *y*. *y*(*c*:=*y c* + *x r* ∗ *v*)) *ts m i* = *m i* + *pr* (α *ts*) *x i*
⟨*proof*⟩

lemma *correct-fold*:
assumes *distinct* (*map fst ts*)
assumes ∀ ((*r*,*c*),-)∈*set ts*. *r*<*n*
shows *fold* (λ((*r*,*c*),*v*) *y*. *y*(*c*:=*y c* + *x r* ∗ *v*)) *ts* (λ-. *0*) = *pr* (α *ts*) *x*
⟨*proof*⟩

lemma *alg-by-fold*: *alg ts x* = *fold* (λ((*r*,*c*),*v*) *y*. *y*(*c*:=*y c* + *x r* ∗ *v*)) *ts* (λ-. *0*)

⟨*proof*⟩

theorem *correct*:
assumes *distinct* (*map fst ts*)
assumes ∀ ((*r*,*c*),-)∈*set ts*. *r*<*n*
shows *alg ts x* = *pr* (α *ts*) *x*
⟨*proof*⟩

6.4 Multi-Threaded Implementation

Correctness of the parallel implementation:

theorem *parallel-correct*:
assumes *distinct* (*map fst ts*) ∀ ((*r*,*c*),-)∈*set ts*. *r*<*n*
and *0* < *n* — At least on thread
— We have reached a final state.
and *reachable x n ts* (λ-. *0*) (*ts′*, *ms*, *r*) *final n* (*ts′*, *ms*, *r*)
shows *r* = *pr* (α *ts*) *x*
⟨*proof*⟩

We also know that the computation will always terminate.

theorem *parallel-termination*:
assumes $0 < n$
and *reachable* $x\ n\ ts\ (\lambda\text{-}.\ 0)\ s$
shows $\exists\, s'.\ \mathit{final}\ n\ s' \wedge (\mathit{step}\ x\ n)^{**}\ s\ s'$
$\langle \mathit{proof} \rangle$

end — Context for fixed n.

end