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*≥*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* ≡ *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*)
by *pat-completeness* *auto*

1.2 Termination

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

lemma *take-non-empty*:
take *xs* ≠ [] **if** *xs* ≠ []
using *that*
apply (*cases* *xs*)
apply *clarsimp*
subgoal for *x* *ys*
apply (*cases* *ys*)
apply *auto*
done
done

termination
apply (*relation* *measure* *length*)
apply (*auto* *simp*: *take2-def* *Let-def*)
using *take-non-empty*
apply *auto*
done

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

1.3 Correctness

1.3.1 Property 1: The Exact Sequence is Covered

lemma *tdconc*: ∃ *ys*. *xs* = *take-decr* *xs* @ *ys*
apply (*induction* *xs* *rule*: *take-decr.induct*)
apply *auto*
done

lemma *ticonc*: ∃ *ys. xs* = *take-incr xs* @ *ys*
apply (*induction xs rule*: *take-incr.induct*)
apply *auto*
done

lemma *take-conc*: ∃ *ys. xs* = *take xs*@*ys*
using *tdconc ticonc*
apply (*cases xs rule*: *take.cases*)
by *auto*

theorem *concat-cuts*: *concat* (*cuts xs*) = *xs*
apply (*induction xs rule*: *cuts.induct*)
apply (*subst cuts.simps*)
apply (*auto simp*: *take2-def Let-def*)
by (*metis append-eq-conv-conj take-conc*)

1.3.2 Property 2: Monotonicity

We define constants to specify increasing/decreasing sequences.

fun *incr* **where**
incr [] ⟷ *True*
| *incr* [-] ⟷ *True*
| *incr* (*x*#*y*#*xs*) ⟷ *x*<*y* ∧ *incr* (*y*#*xs*)

fun *decr* **where**
decr [] ⟷ *True*
| *decr* [-] ⟷ *True*
| *decr* (*x*#*y*#*xs*) ⟷ *x*≥*y* ∧ *decr* (*y*#*xs*)

lemma *tki*: *incr* (*take-incr xs*)
apply (*induction xs rule*: *take-incr.induct*)
apply *auto*
apply (*case-tac xs*)
apply *auto*
done

lemma *tkd*: *decr* (*take-decr xs*)
apply (*induction xs rule*: *take-decr.induct*)
apply *auto*
apply (*case-tac xs*)
apply *auto*
done

lemma *icod*: *incr* (*take xs*) ∨ *decr* (*take xs*)
apply (*cases xs rule*: *take.cases*)
apply (*auto simp*: *tki tkd simp del*: *take-incr.simps take-decr.simps*)
done

theorem *cuts-incr-decr*: ∀ *c*∈*set* (*cuts xs*). *incr c* ∨ *decr c*

apply (*induction xs rule*: *cuts.induct*)
apply (*subst cuts.simps*)
apply (*auto simp*: *take2-def Let-def*)
using *icod* **by** *blast*

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 [] ⟷ *True*
| *maxi* [*c*] ⟷ *c*≠[]
| *maxi* (*c1*#*c2*#*cs*) ⟷ (*c1*≠[] ∧ *c2*≠[] ∧ *maxi* (*c2*#*cs*) ∧ (
    incr c1 ∧ ¬(*last c1* < *hd c2*)
  ∨ *decr c1* ∧ ¬(*last c1* ≥ *hd c2*)
    ))

Obviously, our specification implies that there are no empty segments

lemma *maxi-imp-non-empty*: *maxi xs* ⟹ []∉*set xs*
  by (*induction xs rule*: *maxi.induct*) *auto*

lemma *tdconc′*: *xs*≠[] ⟹
  ∃ *ys*. *xs* = *take-decr xs* @ *ys* ∧ (*ys*≠[]
    ⟶ ¬(*last* (*take-decr xs*) ≥ *hd ys*))
  apply (*induction xs rule*: *take-decr.induct*)
  apply *auto*
  apply (*case-tac xs*) **apply** (*auto split*: *if-splits*)
  done

lemma *ticonc′*: *xs*≠[] ⟹ ∃ *ys*. *xs* = *take-incr xs* @ *ys* ∧ (*ys*≠[] ⟶ ¬(*last* (*take-incr xs*) < *hd ys*))
  apply (*induction xs rule*: *take-incr.induct*)
  apply *auto*
  apply (*case-tac xs*) **apply** (*auto split*: *if-splits*)
  done

lemma *take-conc′*: *xs*≠[] ⟹ ∃ *ys*. *xs* = *take xs*@*ys* ∧ (*ys*≠[] ⟶ (
  take xs=*take-incr xs* ∧ ¬(*last* (*take-incr xs*) < *hd ys*)
∨ *take xs*=*take-decr xs* ∧ ¬(*last* (*take-decr xs*) ≥ *hd ys*)
))
  using *tdconc′ ticonc′*
  apply (*cases xs rule*: *take.cases*)
  by *auto*

lemma *take-decr-non-empty*:

```
  take-decr xs ≠ [] if xs ≠ []
  using that
  apply (cases xs)
   apply auto
  subgoal for x ys
    apply (cases ys)
     apply (auto split: if-split-asm)
    done
  done

lemma take-incr-non-empty:
  take-incr xs ≠ [] if xs ≠ []
  using that
  apply (cases xs)
   apply auto
  subgoal for x ys
    apply (cases ys)
     apply (auto split: if-split-asm)
    done
  done

lemma take-conc'': xs≠[] ⟹ ∃ys. xs = take xs@ys ∧ (ys≠[] ⟶ (
  incr (take xs) ∧ ¬(last (take xs) < hd ys)
∨ decr (take xs) ∧ ¬(last (take xs) ≥ hd ys)
))
  using tdconc' ticonc' tki tkd
  apply (cases xs rule: take.cases)
  apply auto
  apply (auto simp add: take-incr-non-empty)
  apply (simp add: take-decr-non-empty)
  apply (metis list.distinct(1) take-incr.simps(3))
  by (smt (verit) list.simps(3) take-decr.simps(3))



lemma [simp]: cuts [] = []
  apply (subst cuts.simps) by auto

lemma [simp]: cuts xs ≠ [] ⟷ xs ≠ []
  apply (subst cuts.simps)
  apply (auto simp: take2-def Let-def)
  done

lemma inv-cuts: cuts xs = c#cs ⟹ ∃ys. c=take xs ∧ xs=c@ys ∧ cs = cuts ys
  apply (subst (asm) cuts.simps)
  apply (cases xs rule: cuts.cases)
  apply (auto split: if-splits simp: take2-def Let-def)
  by (metis append-eq-conv-conj take-conc)
```

```
theorem maximal-cuts: maxi (cuts xs)
  apply (induction cuts xs arbitrary: xs rule: maxi.induct)
  subgoal by auto
  subgoal for c xs
    apply (drule sym; simp)
    apply (subst (asm) cuts.simps)
    apply (auto split: if-splits prod.splits simp: take2-def Let-def take-non-empty)
    done
  subgoal for c1 c2 cs xs
    apply (drule sym)
    apply simp
    apply (drule inv-cuts; clarsimp)
    apply auto
    subgoal by (metis cuts.simps list.distinct(1) take-non-empty)
    subgoal by (metis append.left-neutral inv-cuts not-Cons-self)
    subgoal using icod by blast
    subgoal by (metis
          Nil-is-append-conv cuts.simps hd-append2 inv-cuts list.distinct(1)
          same-append-eq take-conc'' take-non-empty)
    subgoal by (metis
          append-is-Nil-conv cuts.simps hd-append2 inv-cuts list.distinct(1)
          same-append-eq take-conc'' take-non-empty)
    done
  done
```

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-induction [] = ()
| ii-induction [-] = ()
| ii-induction (-#y#xs) = ii-induction (y#xs)

locale cnvSpec =
  fixes fP P
  assumes [simp]: fP [] ⟷ True
  assumes [simp]: fP [x] ⟷ True
  assumes [simp]: fP (a#b#xs) ⟷ P a b ∧ fP (b#xs)
begin

  lemma idx-spec: fP xs ⟷ (∀ i<length xs − 1. P (xs!i) (xs!Suc i))
    apply (induction xs rule: ii-induction.induct)
    using less-Suc-eq-0-disj
    by auto
```

```
end

locale cnvSpec′ =
  fixes fP P P′
  assumes [simp]: fP [] ⟷ True
  assumes [simp]: fP [x] ⟷ P′ x
  assumes [simp]: fP (a#b#xs) ⟷ P′ a ∧ P′ b ∧ P a b ∧ fP (b#xs)
begin

  lemma idx-spec: fP xs ⟷ (∀ i<length xs. P′ (xs!i)) ∧ (∀ i<length xs − 1. P
(xs!i) (xs!Suc i))
    apply (induction xs rule: ii-induction.induct)
    apply auto []
    apply auto []
    apply clarsimp
    by (smt less-Suc-eq-0-disj nth-Cons-0 nth-Cons-Suc)

end

interpretation INCR: cnvSpec incr (<)
  apply unfold-locales by auto

interpretation DECR: cnvSpec decr (≥)
  apply unfold-locales by auto

interpretation MAXI: cnvSpec′ maxi λc1 c2. ( (
      incr c1 ∧ ¬(last c1 < hd c2)
    ∨ decr c1 ∧ ¬(last c1 ≥ hd c2)
      ))
    λx. x ≠ []
  apply unfold-locales by auto

lemma incr-by-idx: incr xs = (∀ i<length xs − 1. xs ! i < xs ! Suc i)
  by (rule INCR.idx-spec)

lemma decr-by-idx: decr xs = (∀ i<length xs − 1. xs ! i ≥ xs ! Suc i)
  by (rule DECR.idx-spec)

lemma maxi-by-idx: maxi xs ⟷
  (∀ i<length xs. xs ! i ≠ []) ∧
  (∀ i<length xs − 1.
      incr (xs ! i) ∧ ¬ last (xs ! i) < hd (xs ! Suc i)
    ∨ decr (xs ! i) ∧ ¬ hd (xs ! Suc i) ≤ last (xs ! i)
  )
  by (rule MAXI.idx-spec)

theorem all-correct:
  concat (cuts xs) = xs
  ∀ c∈set (cuts xs). incr c ∨ decr c
```

```
    maxi (cuts xs)
    [] ∉ set (cuts xs)
    using cuts-incr-decr concat-cuts maximal-cuts
          maxi-imp-non-empty[OF maximal-cuts]
    by auto

end
```

2 Challenge 1.B

```
theory Challenge1B
  imports Challenge1A HOL−Library.Multiset
begin


lemma mset-concat:
  mset (concat xs) = fold (+) (map mset xs) {#}
proof −
  have mset (concat xs) + a = fold (+) (map mset xs) a for a
  proof (induction xs arbitrary: a)
    case Nil
    then show ?case
      by auto
  next
    case (Cons x xs)
    show ?case
      using Cons.IH[of mset x + a, symmetric] by simp
  qed
  from this[of {#}] show ?thesis
    by auto
qed
```

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
  using assms
proof (induction l1 arbitrary: l2)
```

case *Nil* **thus** *?case*
by *simp*
next
case (*Cons x1 l1 l2*)
note *IH* = *Cons.IH*

show *?case*
using *Cons.prems*
proof (*induction l2*)
case *Nil* **then show** *?case*
by *simp*
next
case (*Cons x2 l2*)
then show *?case*
using *IH* **by** (*force split*: *if-split-asm*)
qed
qed

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*
by *pat-completeness simp-all*
termination by (*relation measure* (λ(*acc*, *ls*). *3* ∗ *length acc* + *2* ∗ *length ls*); *simp*)

lemma *merge-list-correct*:
assumes ⋀*l*. *l* ∈ *set ls* ⟹ *sorted l*
assumes ⋀*l*. *l* ∈ *set as* ⟹ *sorted l*
shows
sorted (*merge-list as ls*)
∧ *mset* (*merge-list as ls*) = *mset* (*concat* (*as* @ *ls*))
∧ *set* (*merge-list as ls*) = *set* (*concat* (*as* @ *ls*))
using *assms*
proof (*induction as ls rule*: *merge-list.induct*)
next
case (*4 la acc2 l*)
then show *?case*
by (*auto simp*: *algebra-simps*)
next
case (*5 acc2 l1 l2 ls*)
have *sorted* (*merge-list* (*merge l1 l2* # *acc2*) *ls*)
∧ *mset* (*merge-list* (*merge l1 l2* # *acc2*) *ls*) = *mset* (*concat* ((*merge l1 l2* # *acc2*) @ *ls*))

$\wedge$ *set* (*merge-list* (*merge l1 l2* # *acc2*) *ls*) = *set* (*concat* ((*merge l1 l2* # *acc2*) @ *ls*))
using *5(2−) merge-correct*[*of l1 l2*] **by** (*intro 5(1)*) *auto*
then show *?case*
using *merge-correct*[*of l1 l2*] *5(2−)* **by** *auto*
qed *simp+*

2.3 GHC-Sort

definition
ghc-sort xs = *merge-list* [] (*map* (*λys. if decr ys then rev ys else ys*) (*cuts xs*))

lemma *decr-sorted*:
assumes *decr xs*
shows *sorted* (*rev xs*)
using *assms* **by** (*induction xs rule*: *decr.induct*) (*auto simp*: *sorted-append*)

lemma *incr-sorted*:
assumes *incr xs*
shows *sorted xs*
using *assms* **by** (*induction xs rule*: *incr.induct*) *auto*

lemma *reverse-phase-sorted*:
$\forall$ *ys* $\in$ *set* (*map* (*λys. if decr ys then rev ys else ys*) (*cuts xs*)). *sorted ys*
using *cuts-incr-decr* **by** (*auto intro*: *decr-sorted incr-sorted*)

lemma *reverse-phase-elements*:
set (*concat* (*map* (*λys. if decr ys then rev ys else ys*) (*cuts xs*))) = *set xs*
proof −
have *set* (*concat* (*map* (*λys. if decr ys then rev ys else ys*) (*cuts xs*)))
= *set* (*concat* (*cuts xs*))
by *auto*
also have ... = *set xs*
by (*simp add*: *concat-cuts*)
finally show *?thesis* **.**
qed

lemma *reverse-phase-permutation*:
mset (*concat* (*map* (*λys. if decr ys then rev ys else ys*) (*cuts xs*))) = *mset xs*
proof −
have *mset* (*concat* (*map* (*λys. if decr ys then rev ys else ys*) (*cuts xs*)))
= *mset* (*concat* (*cuts xs*))
unfolding *mset-concat* **by** (*auto simp*: *comp-def intro*!: *arg-cong2*[**where** *f* = *fold* (+)])
also have ... = *mset xs*
by (*simp add*: *concat-cuts*)
finally show *?thesis* **.**
qed

2.4 Correctness Lemmas

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

theorem *sorted-ghc-sort*:
  sorted (*ghc-sort xs*)
  unfolding *ghc-sort-def* **using** *reverse-phase-sorted*
  by (*intro merge-list-correct*[*THEN conjunct1*]) *auto*

theorem *permutation-ghc-sort*:
  mset (*ghc-sort xs*) = *mset xs*
  unfolding *ghc-sort-def*
  apply (*subst merge-list-correct*[*THEN conjunct2*])
  subgoal
    using *reverse-phase-sorted* **by** *auto*
  subgoal
    using *reverse-phase-sorted* **by** *auto*
  apply (*subst* (*2*) *reverse-phase-permutation*[*symmetric*])
  apply *simp*
  done

corollary *elements-ghc-sort*: *set* (*ghc-sort xs*) = *set xs*
  using *permutation-ghc-sort* **by** (*metis set-mset-mset*)

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* (∃ *i*<*j*. *xs* ! *i* < *xs* ! *j*) *then Some* (*GREATEST i*. *i* < *j* ∧ *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* ≡ *length lf* = *length xs* ∧ (∀ *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* = (∃ *i*<*j*. *xs* ! *i* < *xs* ! *j*)
definition *the-left j* = (*GREATEST i*. *i* < *j* ∧ *xs* ! *i* < *xs* ! *j*)

lemma *left-alt*: *left-spec j* = (*if has-left j then Some* (*the-left j*) *else None*)
  by (*auto simp*: *left-spec-def has-left-def the-left-def*)

lemma *the-leftI*: *has-left j* ⟹ *the-left j* < *j* ∧ *xs*!*the-left j* < *xs*!*j*
  apply (*clarsimp simp*: *has-left-def the-left-def*)
  by (*metis* (*no-types*, *lifting*) *GreatestI-nat less-le-not-le nat-le-linear pinf*(*5*))

lemma *the-left-decr*[*simp*]: *has-left i* ⟹ *the-left i* < *i*
  by (*simp add*: *the-leftI*)

lemma *le-the-leftI*:
  assumes *i*≤*j* *xs*!*i* < *xs*!*j*
  shows *i* ≤ *the-left j*
  using *assms* **unfolding** *the-left-def*
  by (*metis* (*no-types*, *lifting*)
    Greatest-le-nat le-less-linear less-imp-not-less less-irrefl
    order.not-eq-order-implies-strict)

lemma *the-left-leI*:
  assumes ∀ *k*. *j*<*k* ∧ *k*<*i* ⟶ ¬*xs*!*k*<*xs*!*i*
  assumes *has-left i*
  shows *the-left i* ≤ *j*
  using *assms*

unfolding *the-left-def has-left-def*
apply *auto*
by (*metis* (*full-types*) *the-leftI assms*(*2*) *not-le the-left-def*)

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*: $k \in set\ (der\text{-}stack\ i_0) \Longrightarrow k<i_0$
apply (*induction* i_0 *rule*: *der-stack.induct*)
apply (*subst* (*asm*) *der-stack.simps*)
apply *auto*
using *less-trans the-leftI* **by** *blast*

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* = [] **by** (*auto simp*: *pop-def*)
lemma *pop-cons*: *pop* (*j#js*) *v* = (*if xs*!*j* ≥ *v then pop js v else j#js*)
by (*simp add*: *pop-def*)

definition *all-left* ≡ *doN* {
(-,*lf*) ← *nfoldli* [*0*..<*length xs*] (λ-. *True*) (λ*i* (*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 i₀ < length xs
  assumes i<i₀
  assumes left-spec i₀ ≤ Some i
  shows if xs!i < xs!i₀ then left-spec i₀ = Some i
        else left-spec i₀ ≤ left-spec i
  using assms
  apply (simp; intro impI conjI; clarsimp)
  subgoal
    apply (auto simp: left-alt split: if-splits)
    apply (simp add: le-antisym le-the-leftI)
    apply (auto simp: has-left-def)
    done
  subgoal
    apply (auto simp: left-alt split: if-splits)
    subgoal
      apply (drule the-leftI)
      using nat-less-le by (auto simp: has-left-def)
    subgoal
      using le-the-leftI the-leftI by fastforce
    done
  done
```

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

```
lemma pop-aux: ⟦ k<i₀; i₀<length xs; left-spec i₀ ≤ Some k ⟧ ⟹ pop (k #
der-stack k) (xs!i₀) = der-stack i₀
  apply (induction k rule: nat-less-induct)
  apply (clarsimp)
  by (smt der-stack.simps left-alt pop-def the-leftI dropWhile.simps(1) find-left-rl
leD less-option-None-Some option.inject pop-cons)
```

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 by auto

lemma left-spec-None-iff[simp]: left-spec i = None ⟷ ¬has-left i by (auto simp:
left-alt)
lemma [simp]: left-spec 0 = None by (auto simp: left-spec-def)
lemma [simp]: has-left 0 = False
  by (simp add: has-left-def)
lemma [simp]: der-stack 0 = []
  by (subst der-stack.simps) auto


lemma algo-correct: all-left ≤ SPEC all-left-spec
  unfolding all-left-def all-left-spec-def
  apply (refine-vcg nfoldli-upt-rule[where I=
    λk (stk,lf).
      (length lf = length xs)
    ∧ (∀ i<k. lf!i = left-spec i)
    ∧ (case k of Suc kk ⇒ stk = kk#der-stack kk | - ⇒  stk=[])
      ])
  apply (vc-solve split: nat.splits)
  subgoal using set-der-stack-lt by fastforce
  subgoal for lf k
   by (metis left-alt less-Suc-eq-le less-eq-option-None less-eq-option-Some nat-in-between-eq(2)
pop-aux the-leftI)
  subgoal
    by (metis der-stack.simps left-alt less-Suc-eq list.distinct(1) nth-list-update)
  subgoal
    by (metis der-stack.simps left-alt less-Suc-eq list.sel(1) nth-list-update)
  done
```

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})
```

$(\lambda[] \Rightarrow$ *RETURN False* | *k#stk* $\Rightarrow$ *doN* { *ASSERT* (*k<length xs*); *RETURN* ($v \leq xs!k$) })
(λ*stk. mop-list-tl stk*)
stk

lemma *pop2-refine-aux*: *set stk* $\subseteq$ {*0..<length xs*} $\Longrightarrow$ *pop2 stk v* $\leq$ *RETURN* (*pop stk v*)
apply (*induction stk*)
unfolding *pop-def pop2-def*
subgoal
apply (*subst monadic-WHILEIT-unfold*)
by *auto*
subgoal
apply (*subst monadic-WHILEIT-unfold*)
unfolding *mop-list-tl-def op-list-tl-def* **by** *auto*
done

end — Context fixing the input *xs*.

The refinement lemma written in higher-order form.

lemma *pop2-refine*: (*uncurry2 pop2*, *uncurry2* (*RETURN ooo pop*)) $\in$ [λ((*xs*,*stk*),*v*). *set stk* $\subseteq$ {*0..<length xs*}]$_f$ (*Id* $\times_r$ *Id*) $\times_r$ *Id* $\rightarrow$ $\langle$*Id*$\rangle$*nres-rel*
using *pop2-refine-aux*
by (*auto intro*!: *frefI nres-relI*)

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

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

3.5.2 Implementation of Main Algorithm

sepref-definition *all-left-impl* **is** *all-left* :: (*array-assn id-assn*)^k $\rightarrow_a$ *array-assn* (*option-assn id-assn*)
unfolding *all-left-def*
apply (*rewrite at nfoldli - - - (⌑,-) HOL-list.fold-custom-empty*)
apply (*rewrite* **in** *nfoldli - - - (-,⌑) array-fold-custom-replicate*)
by *sepref*

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*)

$\in \langle Id\rangle list\text{-}rel \rightarrow \langle\langle\langle Id\rangle option\text{-}rel\rangle list\text{-}rel\rangle nres\text{-}rel$
using *algo-correct* **by** (*auto simp*: *nres-relI*)

Main correctness theorem in higher-order form.

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

using *all-left-impl.refine*[*FCOMP algo-correct′*, *simplified*] **.**

Main correctness theorem as Hoare-Triple

theorem *algo-impl-correct′*:
<*array-assn int-assn xs xsi*>
all-left-impl xsi
<λ*lfi*. $\exists_A$*lf*. *array-assn int-assn xs xsi*
$*$ *array-assn* (*option-assn id-assn*) *lf lfi*
$*$ $\uparrow$(*all-left-spec xs lf*)>$_t$
apply (*rule cons-rule*[*OF - - algo-impl-correct*[*to-hnr*, *THEN hn-refineD*, *unfolded autoref-tag-defs*]])
apply (*simp add*: *hn-ctxt-def*, *rule ent-refl*)
by (*auto simp*: *hn-ctxt-def*)

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-val ‹

```
(* Convert from option to 1−based indexes *)
fun cnv NONE = 0
  | cnv (SOME i) = @{code integer-of-nat} i + 1

(* The verified algorithm, boxing the input list into an array,
  and unboxing the output to a list, and converting it from option to 1−based *)
fun all-left xs =
    @{code all-left-impl} (Array.fromList (map @{code int-of-integer} xs)) ()
  |> Array.foldr (op ::) []
  |> map cnv

val test = all-left [ 4, 7, 8, 1, 2, 3, 9, 5, 6 ]
```

›

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 (∃ i>j. xs ! i < xs ! j) then Some (LEAST i. i > j ∧ 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) ⇒ None
    | (Some x, None) ⇒ Some x
    | (None, Some y) ⇒ Some y
    | (Some x, Some y) ⇒ 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 −
  note [simp del] = left-spec-None-iff swap-adhoc
  show ?thesis
  proof (cases (∃ i<j. xs ! i < xs ! j))
    case True
    then have ∗: xs ! the (left-spec xs j) < xs ! j left-spec xs j ≠ None
      unfolding left-spec-def by auto (metis (no-types, lifting) GreatestI-nat True
less-le)
    show ?thesis
    proof (cases (∃ i>j. xs ! i < xs ! j))
      case True
      then have xs ! the (right-spec xs j) < xs ! j right-spec xs j ≠ None
```

```
      unfolding right-spec-def by auto (metis (no-types, lifting) LeastI)
    then show ?thesis
      using * assms unfolding parent-def by auto
  next
    case False
    then have right-spec xs j = None
      unfolding right-spec-def by auto
    then show ?thesis
      using * assms unfolding parent-def by auto
  qed
 next
  case False
  then have [simp]: left-spec xs j = None
    unfolding left-spec-def by auto
  show ?thesis
  proof (cases (∃ i>j. xs ! i < xs ! j))
    case True
    then have xs ! the (right-spec xs j) < xs ! j right-spec xs j ≠ None
      unfolding right-spec-def by auto (metis (no-types, lifting) LeastI)
    then show ?thesis
      using assms unfolding parent-def by auto
  next
    case False
    then have right-spec xs j = None
      unfolding right-spec-def by auto
    then show ?thesis
      using assms unfolding parent-def by auto
  qed
 qed
qed

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
  by (induction l arbitrary: a; clarsimp; metis fold-mset-fun-left-comm)

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 ⊆ A ∧ ran m ⊆ B}
proof −
  have {m. dom m ⊆ A ∧ ran m ⊆ B} ⊆ (⋃ S ∈ {S. S ⊆ A}. {m. dom m = S
∧ ran m ⊆ B})
    by auto
  moreover have finite ...
    using assms by (auto intro!: finite-set-of-finite-maps intro: finite-subset)
  ultimately show ?thesis
    by (rule finite-subset)
qed

lemma wf-rtranclp-ev-induct[consumes 1, case-names step]:
  assumes wf {(x, y). R y x} and step: ⋀ x. R** a x ⟹ P x ∨ (∃ y. R x y)
  shows ∃x. P x ∧ R** a x
proof −
  have ∃y. P y ∧ R** x y if R** a x for x
    using assms(1) that
  proof induction
    case (less x)
    from step[OF ‹R** a x›] have P x ∨ (∃y. R x y) .
    then show ?case
    proof
      assume P x
      then show ?case
        by auto
    next
      assume ∃y. R x y
      then obtain y where R x y ..
      with less(1)[of y] less(2) show ?thesis
       by simp (meson converse-rtranclp-into-rtranclp rtranclp.rtrancl-into-rtrancl)
    qed
  qed
  then show ?thesis
    by blast
qed
```

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* $\times$ $(nat \rightharpoonup 'a) \times 's$

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 $\exists\ cfg'.\ step\ cfg\ cfg'$
  using *assms*
  apply (*cases cfg*)
  apply *safe*
  subgoal for *ts ws s*
    by (*cases ts*; *cases ws 0*) (*auto 4 5 simp*: *final-def intro*: *step.intros*)
  done

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 –
  let $?R1 = \{(x, y).\ dom\ x \subset dom\ y \wedge ran\ x \subseteq S \wedge dom\ y \subseteq \{0..<n\} \wedge ran\ y \subseteq S\}$
  have $?R1 \subseteq \{y.\ dom\ y \subseteq \{0..<n\} \wedge ran\ y \subseteq S\} \times \{y.\ dom\ y \subseteq \{0..<n\} \wedge ran\ y \subseteq S\}$

by *auto*
then have *finite ?R1*
using ‹*finite S*› **by** − (*erule finite-subset*, *auto intro*: *finite-set-of-finite-maps*)
then have [*intro*]: *wf ?R1*
apply (*rule finite-acyclic-wf*)
apply (*rule preorder-class.acyclicI-order*[**where** $f = \lambda x.\ n - card\ (dom\ x)$])
apply *clarsimp*
by (*metis* (*full-types*)
cancel-ab-semigroup-add-class.diff-right-commute diff-diff-cancel domD domI psubsetI psubset-card-mono subset-eq-atLeast0-lessThan-card subset-eq-atLeast0-lessThan-finite zero-less-diff)
let *?R* = *measure length* <∗*lex*∗> *?R1* <∗*lex*∗> {}
have *wf ?R*
by *auto*
then show *?thesis*
apply (*rule wf-subset*)
apply *clarsimp*
apply (*erule step.cases*; *clarsimp*)
by (*smt*
Diff-iff domIff fun-upd-apply mem-Collect-eq option.simps(*3*) *psubsetI ran-def singletonI subset-iff*)
qed

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

definition
$s_0 = (ts,\ \lambda\text{-}.\ None,\ start)$

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

lemma *reachable0*[*simp*]: *reachable* s_0
unfolding *reachable-def* **by** *auto*

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

lemma *is-invarI*[*intro?*]:
$[\![\ I\ s_0;\ \bigwedge s\ s'.\ [\![\ reachable\ s;\ I\ s;\ step\ s\ s']\!] \Longrightarrow I\ s'\ ]\!] \Longrightarrow is\text{-}invar\ I$
by (*auto simp*: *is-invar-def*)

lemma *invar-reachable*: $is\text{-}invar\ I \Longrightarrow reachable\ s \Longrightarrow I\ s$
unfolding *reachable-def*
by *rotate-tac* (*induction rule*: *rtranclp-induct*, *auto simp*: *is-invar-def reachable-def*)

definition
invar ≡ λ(*ts2*, *ws*, *r*).
(∃ *ts1*.

mset ts = *ts1* + {# *the* (*ws i*). *i* ∈# *mset-set* (*dom ws* ∩ {*0*..<*n*}) #} + *mset ts2*
∧ *r* = *fold-mset f start ts1*
∧ *set ts2* ⊆ *set ts* ∧ *ran ws* ⊆ *set ts* ∧ *dom ws* ⊆ {*0*..<*n*})

lemma *invariant*:
is-invar invar
apply *rule*
subgoal
unfolding s_0*-def* **unfolding** *invar-def* **by** *simp*
subgoal
unfolding *invar-def*
apply (*elim step.cases*)
apply (*clarsimp split*: *option.split-asm*)
subgoal for *ws i t ts ts1*
apply (*rule exI*[**where** *x* = *ts1*])
apply (*subst mset-set.insert*)
apply (*auto intro*!: *multiset.map-cong0*)
done
apply (*clarsimp split*!: *prod.splits*)
subgoal for *ws i a ts ts1*
apply (*rule exI*[**where** *x* = *add-mset a ts1*])
apply (*subst Diff-Int-distrib2*)
apply (*subst mset-set.remove*)
apply (*auto intro*!: *multiset.map-cong0 split*: *if-split-asm simp*: *ran-def*)
done
done
done

lemma *final-state-correct1*:
assumes *invar* (*ts′*, *ms*, *r*) *final* (*ts′*, *ms*, *r*)
shows *r* = *fold-mset f start* (*mset ts*)
using *assms* **unfolding** *invar-def final-def* **by** *auto*

lemma *final-state-correct2*:
assumes *reachable* (*ts′*, *ms*, *r*) *final* (*ts′*, *ms*, *r*)
shows *r* = *fold-mset f start* (*mset ts*)
using *assms* **by** − (*rule final-state-correct1*, *rule invar-reachable*[*OF invariant*])

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*
using *final-state-correct2*[*OF assms*] **by** (*simp add*: *fold-mset-rewr*)

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 ∃s'. final s' ∧ step** s s'
proof -
  have {(s', s). step s s' ∧ reachable s} ⊆ {(s', s). step s s' ∧ reachable s ∧ reachable s'}
    unfolding reachable-def by auto
  also have ... ⊆ {((ts', ws', r'), (ts1, ws, r)).
    step (ts1, ws, r) (ts', ws', r') ∧ set ts' ⊆ set ts ∧ dom ws ⊆ {0..<n} ∧ ran ws ⊆ set ts}
    by (force dest!: invar-reachable[OF invariant] simp: invar-def)
  finally have wf {(s', s). step s s' ∧ reachable s}
    by (elim wf-subset[OF wf-step, rotated]) simp
  then have ∃s'. final s' ∧ (λs s'. step s s' ∧ reachable s)** s s'
  proof (induction rule: wf-rtranclp-ev-induct)
    case (step x)
    then have (λs s'. step s s')** s x
      by (elim mono-rtranclp[rule-format, rotated] conjE)
    with ‹reachable s› have reachable x
      unfolding reachable-def by auto
    then show ?case
      using no-deadlock[of x] by auto
  qed
  then show ?thesis
    apply clarsimp
    apply (intro exI conjI, assumption)
    apply (rule mono-rtranclp[rule-format])
     apply auto
    done
qed

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*))
apply *unfold-locales*
apply (*auto intro*!: *ext*)
done

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*))

apply (*auto split*: *option.splits*)

```
 by (metis disjoint-iff-not-equal img-fst map-of-eq-None-iff the-default.simps(2))

lemma 1[simp]: distinct (map fst (ts1@ts2)) ⟹
  pr (α (ts1@ts2)) x i = pr (α ts1) x i + pr (α ts2) x i
  apply (auto simp: pr-def α-def map-add-def aux split: option.splits)
  apply (auto simp: algebra-simps)
  by (simp add: sum.distrib)

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

lemma [simp]: α [] = (λ-. 0) by (auto simp: α-def)

lemma [simp]: pr (λ-. 0::val) x = (λ-. 0)
  by (auto simp: pr-def[abs-def])

lemma aux3: the-default 0 (if b then Some x else None) = (if b then x else 0)
  by auto

lemma correct-aux: ⟦distinct (map fst ts); ∀((r,c),-)∈set ts. r<n⟧
  ⟹ ∀i. fold (λ((r,c),v) y. y(c:=y c + x r ∗ v)) ts m i = m i + pr (α ts) x i
  apply (induction ts arbitrary: m)
  apply auto
  subgoal
    apply (subst 2)
    apply auto
    unfolding pr-def α-def
    apply (auto split: if-splits cong: sum.cong simp: aux3)
    apply (auto simp: if-distrib[where f=λx. -∗x] cong: sum.cong if-cong)
    done

  subgoal
    apply (subst 2)
    apply auto
    unfolding pr-def α-def
    apply (auto split: if-splits cong: sum.cong simp: aux3)
    done
  done


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
  apply (rule ext)
  using correct-aux[OF assms, rule-format, where m = λ-. 0, simplified]
  by simp

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

unfolding *alg-def* **by** (*simp add*: *fold-mset-rewr*)

theorem *correct*:
assumes *distinct* (*map fst ts*)
assumes $\forall\,((r,c),\text{-})\in set\ ts.\ r<n$
shows *alg ts x* = *pr* (α *ts*) *x*
using *alg-by-fold correct-fold*[*OF assms*] **by** *simp*

6.4 Multi-Threaded Implementation

Correctness of the parallel implementation:

theorem *parallel-correct*:
assumes *distinct* (*map fst ts*) $\forall\,((r,c),\text{-})\in set\ ts.\ r<n$
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*
unfolding *final-state-correct*[*OF assms*(*3*−)] *correct*[*OF assms*(*1*,*2*)] *alg-by-fold*[*symmetric*]
..

We also know that the computation will always terminate.

theorem *parallel-termination*:
assumes $0 < n$
and *reachable x n ts* (λ-. *0*) *s*
shows $\exists\, s'.$ *final n s'* $\wedge$ (*step x n*)** *s s'*
using *assms* **by** (*rule termination*)

end — Context for fixed *n*.

end