# VerifyThis 2019 - Polished Isabelle Solutions 

Peter Lammich Simon Wimmer

September 13, 2023


#### 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 \(\Rightarrow\) - where
    take-incr [] = []
|take-incr \([x]=[x]\)
\(\mid\) take-incr \((x \# y \# x s)=(\) if \(x<y\) then \(x \#\) take-incr \((y \# x s)\) else \([x])\)
fun take-decr :: int list \(\Rightarrow\) - where
    take-decr [] = []
| take-decr \([x]=[x]\)
```

$\mid$ take-decr $(x \# y \# x s)=($ if $x \geq y$ then $x \#$ take-decr $(y \# x s)$ else $[x])$

```
fun take where
    take [] = []
|take \([x]=[x]\)
\(\mid\) take \((x \# y \# x s)=(\) if \(x<y\) then take-incr \((x \# y \# x s)\) else take-decr \((x \# y \# x s))\)
```

definition take2 $x s \equiv$ let l=take $x s$ in $(l$, drop (length $l) x s)$
- 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 \(x s=(\) if \(x s=[]\) then [] else let \((c, x s)=\) take2 \(x s\) in \(c \#\) cuts \(x s)\)
    \(\langle p r o o f\rangle\)
```


### 1.2 Termination

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

```
lemma take-non-empty:
    take xs \not= [] if xs \not= []
    \langleproof\rangle
termination
    <proof\rangle
declare cuts.simps[simp del]
```


### 1.3 Correctness

```
1.3.1 Property 1: The Exact Sequence is Covered
    lemma tdconc: \(\exists y s . x s=\) take-decr \(x s\) @ ys
        \(\langle p r o o f\rangle\)
    lemma ticonc: \(\exists\) ys. \(x s=\) take-incr xs @ ys
        \(\langle p r o o f\rangle\)
    lemma take-conc: \(\exists\) ys. \(x s=\) take \(x s @ y s\)
        \(\langle p r o o f\rangle\)
    theorem concat-cuts: concat (cuts xs) \(=x s\)
        \(\langle p r o o f\rangle\)
```


### 1.3.2 Property 2: Monotonicity

We define constants to specify increasing/decreasing sequences.
fun incr where

```
incr []\(\longleftrightarrow\) True
| incr [-] \(\longleftrightarrow\) True
\(\mid \operatorname{incr}(x \# y \# x s) \longleftrightarrow x<y \wedge \operatorname{incr}(y \# x s)\)
fun decr where
decr []\(\longleftrightarrow\) True
decr \([-] \longleftrightarrow\) True
\(\mid \operatorname{decr}(x \# y \# x s) \longleftrightarrow x \geq y \wedge \operatorname{decr}(y \# x s)\)
lemma tki: incr (take-incr xs)
\(\langle p r o o f\rangle\)
lemma tkd: decr (take-decr xs)
\(\langle p r o o f\rangle\)
lemma icod: incr (take xs) \(\vee\) decr (take xs)
\(\langle p r o o f\rangle\)
```

theorem cuts-incr-decr: $\forall c \in$ set (cuts xs). incr $c \vee$ decr $c$
$\langle p r o o f\rangle$

### 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
\(\operatorname{maxi}[] \longleftrightarrow\) True
\(\mid \operatorname{maxi}[c] \longleftrightarrow c \neq[]\)
\(\mid \operatorname{maxi}(c 1 \# c 2 \# c s) \longleftrightarrow(c 1 \neq[] \wedge c 2 \neq[] \wedge \operatorname{maxi}(c 2 \# c s) \wedge(\)
        incr c1 \(\wedge \neg(\) last \(c 1<h d c 2)\)
    \(\vee\) decr \(c 1 \wedge \neg(\) last \(c 1 \geq h d c 2)\)
        ))
```

Obviously, our specification implies that there are no empty segments

```
lemma maxi-imp-non-empty: maxi xs \(\Longrightarrow[] \notin\) set xs
\(\langle p r o o f\rangle\)
```

lemma tdconc': xs $\neq[] \Longrightarrow$
$\exists y s . x s=$ take-decr $x s @ y s \wedge(y s \neq[]$
$\longrightarrow \neg($ last (take-decr $x s) \geq$ hd ys))
$\langle p r o o f\rangle$
lemma ticonc': $x s \neq[] \Longrightarrow \exists y s . x s=$ take-incr $x s @ y s \wedge(y s \neq[] \longrightarrow \neg$ (last
(take-incr xs) < hd ys))
$\langle p r o o f\rangle$

```
lemma take-conc': xs }=[]\Longrightarrow\existsys.xs=take xs@ys ^(ys\not=[]\longrightarrow
    take xs=take-incr xs ^\neg(last (take-incr xs)<hd ys)
\checkmark ~ t a k e ~ x s = t a k e - d e c r ~ x s ~ \wedge ~ ᄀ ( l a s t ~ ( t a k e - d e c r ~ x s ) ~ \geq h d ~ y s )
))
    \langleproof\rangle
lemma take-decr-non-empty:
    take-decr xs \not= [] if xs \not= []
    \langleproof\rangle
lemma take-incr-non-empty:
    take-incr xs \not=[] if xs \not=[]
    <proof\rangle
lemma take-conc'\prime: xs }\not=[]\Longrightarrow\existsys.xs=take xs@ys \wedge(ys\not=[]\longrightarrow
    incr (take xs) ^ ᄀ(last (take xs)<hd ys)
\vee \operatorname { d e c r } ( \text { take xs )} \wedge \neg ( \text { last (take xs) } \geq \text { hd ys)}
))
\langleproof\rangle
lemma [simp]: cuts [] = []
    \langleproof\rangle
lemma [simp]: cuts xs }\not=[]\longleftrightarrowxs\not=[
    <proof>
lemma inv-cuts:cuts xs =c#cs\Longrightarrow\existsys.c=take xs }\wedgexs=c@ys \wedgecs=cuts y
    \langleproof\rangle
theorem maximal-cuts: maxi (cuts xs)
    \langleproof\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-induction [] = ()
| ii-induction \([-]=()\)
| ii-induction \((-\# y \# x s)=\) ii-induction \((y \# x s)\)
```

locale cnvSpec $=$

```
    fixes }fP
    assumes [simp]: fP [] \longleftrightarrow True
    assumes [simp]: fP[x]\longleftrightarrow True
    assumes [simp]:fP(a#b#xs)\longleftrightarrowPab\wedgefP(b#xs)
begin
    lemma idx-spec: fP xs \longleftrightarrow (\foralli<length xs - 1.P (xs!i) (xs!Suc i))
    \langleproof\rangle
end
locale cnvSpec'=
    fixes fP P P
    assumes [simp]: fP []\longleftrightarrow True
    assumes [simp]: fP [x]\longleftrightarrow \longleftrightarrow P'x
    assumes [simp]:fP(a#b#xs)\longleftrightarrow \longleftrightarrowP' a}^\mp@code{P}\mp@subsup{P}{}{\prime}b\wedgePab\wedgefP(b#xs
begin
    lemma idx-spec: fP xs \longleftrightarrow (\foralli<length xs. P' (xs!i)) ^(\foralli<length xs - 1.P
(xs!i)(xs!Suc i))
    <proof\rangle
end
interpretation INCR: cnvSpec incr (<)
    <proof>
interpretation DECR: cnvSpec decr ( }\geq\mathrm{ )
    <proof>
interpretation MAXI: cnvSpec' maxi \lambdac1 c2. ((
            incr c1 ^\neg(last c1<hd c2)
        \ decr c1 ^ ᄀ(last c1 \geqhd c2)
        ))
        \lambdax. x\not=[]
    <proof>
lemma incr-by-idx: incr xs =(\forall i<length xs - 1.xs!i< xs!Suc i)
\langleproof\rangle
lemma decr-by-idx: decr xs =(\foralli<length xs - 1.xs!i\geqxs!Suc i)
\langleproof\rangle
lemma maxi-by-idx: maxi xs \longleftrightarrow
    (\foralli<length xs. xs ! i\not= [])^
    (}\foralli<length xs - 1.
            incr (xs!i) ^ ᄀ last (xs!i)<hd (xs!Suc i)
        \veedecr (xs!i)^\neghd (xs!Suc i) \leqlast (xs!i)
    )
```

```
    <proof\rangle
```

    theorem all-correct:
    concat (cuts xs) \(=x s\)
    \(\forall c \in\) set (cuts xs). incr \(c \vee\) decr \(c\)
    maxi (cuts xs)
    []\(\notin\) set (cuts xs)
    〈proof〉
    end

## 2 Challenge 1.B

theory Challenge1B
imports Challenge1A HOL-Library.Multiset
begin
lemma mset-concat:
mset (concat $x s)=$ fold $(+)($ map mset $x s)\{\#\}$
$\langle p r o o f\rangle$

### 2.1 Merging Two Segments

```
fun merge \(::\) ' \(a::\{\) linorder \(\}\) list \(\Rightarrow\) ' \(a\) list \(\Rightarrow\) 'a list where
    merge [] \(12=12\)
    | merge l1[] = l1
\(\mid \operatorname{merge}(x 1 \# l 1)(x 2 \# 12)=\)
    (if \((x 1<x 2)\) then \(x 1\) \# (merge l1 (x2 \# l2)) else x2 \# (merge \((x 1 \#\) l1) l2))
lemma merge-correct:
    assumes sorted 11
    assumes sorted l2
    shows
    sorted (merge l1 l2)
\(\wedge\) mset \((\) merge l1 l2 \()=\) mset \(l 1+m\) set l2
\(\wedge\) set \((\) merge l1 l2 \()=\) set l1 \(\cup\) set l2
\(\langle p r o o f\rangle\)
```


### 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) $l s$
$\langle p r o o f\rangle$

## termination $\langle p r o o f\rangle$

lemma merge－list－correct：
assumes $\bigwedge l . l \in$ set $l s \Longrightarrow$ sorted $l$
assumes $\bigwedge l . l \in$ set as $\Longrightarrow$ sorted $l$
shows
sorted（merge－list as ls）
$\wedge \operatorname{mset}($ merge－list as ls）$=\operatorname{mset}(\operatorname{concat}($ as＠ls））
$\wedge \operatorname{set}($ merge－list as ls $)=\operatorname{set}(\operatorname{concat}($ as＠ls））
$\langle$ proof $\rangle$

### 2.3 GHC－Sort

## definition

```
ghc-sort xs = merge-list [] (map (\lambdays. 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 $x s$〈proof〉
lemma reverse－phase－sorted： $\forall y s \in \operatorname{set}(\operatorname{map}$（ $\lambda y s$ ．if decr ys then rev ys else ys）（cuts xs））．sorted ys $\langle p r o o f\rangle$
lemma reverse－phase－elements：
set（concat（map（ $\lambda y s$. if decr ys then rev ys else ys）$($ cuts $x s))$ ）$=$ set $x s$ $\langle p r o o f\rangle$
lemma reverse－phase－permutation：
$m s e t(\operatorname{concat}(\operatorname{map}(\lambda y s$. if decr ys then rev ys else ys）$($ cuts $x s)))=$ mset $x s$ $\langle p r o o f\rangle$

## 2．4 Correctness Lemmas

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

## theorem sorted－ghc－sort：

```
sorted (ghc-sort xs)
<proof\rangle
```

theorem permutation－ghc－sort： mset（ghc－sort xs）$=$ mset xs $\langle p r o o f\rangle$

```
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 $x s$ :: int list begin
We then specify the desired output: For each index $j$, return the greatest index $i<j$ such that $x s!i<x s!j$, 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 \((\exists i<j . x s!i<x s!j)\) then Some (GREATEST \(i . i<j \wedge x s!i\)
\(<x s!j)\) else None)
```

The output of the algorithm should be an array $l f$, containing the indexes of the left-smaller values:
definition all-left-spec lf $\equiv$ length $l f=$ length $x s \wedge(\forall i<$ length $x s . l f!i=$ 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 = ( }\existsi<j.xs!i<xs!j
definition the-left j=(GREATEST i. i<j\wedgexs!i<xs!j)
lemma left-alt:left-spec j = (if has-left j then Some (the-left j) else None)
    \langleproof\rangle
lemma the-leftI: has-left j\Longrightarrow the-left j<j^ xs!the-left j<xs!j
    <proof>
lemma the-left-decr[simp]: has-left i\Longrightarrow the-left i<i
    \langleproof\rangle
lemma le-the-leftI:
    assumes i\leqj xs!i<xs!j
    shows i\leq the-left j
    \langleproof\rangle
lemma the-left-leI:
    assumes }\forallk.j<k\wedgek<i\longrightarrow\negxs!k<xs!
    assumes has-left i
    shows the-left i\leqj
    <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 \(\left.i_{0}\right) \Longrightarrow k<i_{0}\)
    \(\langle\) 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 \((\lambda j . x s!j \geq v)\) stk
lemma pop-Nil[simp]: pop [] \(v=[]\langle\) proof \(\rangle\)
lemma pop-cons: pop ( \(j \# j s) v=(\) if \(x s!j \geq v\) then pop \(j\) s \(v\) else \(j \# j s)\)
    \(\langle p r o o f\rangle\)
definition all-left \(\equiv\) doN \(\{\)
    \((-, l f) \leftarrow\) nfoldli \([0 . .<\) length \(x s](\lambda-\). True \()(\lambda i(s t k, l f) . d o N\{\)
        ASSERT ( set stk \(\subseteq\{0 . .<\) length \(x s\})\);
        let stk \(=\) pop stk \((x s!i)\);
        ASSERT (stk = der-stack \(i\) );
        ASSERT ( \(i<\) length lf);
        if \((\) stk \(=[])\) then doN \(\{\)
            let \(l f=l f[i:=\) None \(]\);
        RETURN ( \(i \# s t k, l f\) )
        \} else doN \{
        let lf \(=l f[i:=\) Some ( \(h d s t k\) ) \(]\);
        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 io 
    shows if xs!i<xs!io then left-spec i}\mp@subsup{i}{0}{}=\mathrm{ Some }
        else left-spec ion 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 \(x s ;\) left-spec \(i_{0} \leq\) Some \(k \rrbracket \Longrightarrow\) pop \((k \#\)
der-stack \(k)\left(x s!i_{0}\right)=\) der-stack \(i_{0}\)
    〈proof〉
```


## 3．4．2 Main Algorithm

Ad－Hoc lemmas

```
lemma swap-adhoc[simp]:
    None \(=\) left \(i \longleftrightarrow\) left \(i=\) None
    Some \(j=\) left \(i \longleftrightarrow\) left \(i=\) Some \(j\langle\) proof \(\rangle\)
lemma left-spec-None-iff [simp]: left-spec \(i=\) None \(\longleftrightarrow ~ \neg h a s-l e f t ~ i\langle p r o o f\rangle\)
lemma [simp]: left-spec \(0=\) None \(\langle\) proof \(\rangle\)
lemma [simp]: has-left \(0=\) False
    〈proof〉
lemma [simp]: der-stack \(0=[]\)
    \(\langle p r o o f\rangle\)
```

lemma algo-correct: all-left $\leq$ SPEC all-left-spec
$\langle p r o o f\rangle$

## 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 effi－ cient．

## 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
        ( \(\lambda\)-. set stk \(\subseteq\{0 . .<\) length \(x s\}\) )
        \((\lambda] \Rightarrow\) RETURN False \(\mid k \# s t k \Rightarrow\) doN \(\{\) ASSERT ( \(k<\) length \(x s\) ); RETURN
\((v \leq x s!k)\})\)
    ( dstk. mop-list-tl stk)
    stk
lemma pop2-refine-aux: set stk \(\subseteq\{0 . .<\) length \(x s\} \Longrightarrow\) pop2 stk \(v \leq R E T U R N\)
(pop stk v)
    〈proof〉
end - Context fixing the input \(x s\).
```

The refinement lemma written in higher－order form．
lemma pop2－refine：（uncurry2 pop2，uncurry2（RETURN ooo pop $)$ ）$\in[\lambda((x s, s t k), v)$ ． set stk $\subseteq\{0 . .<$ length $x s\}]_{f}\left(I d \times_{r} I d\right) \times_{r} I d \rightarrow\langle I d\rangle$ nres－rel
$\langle$ proof〉
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 $i d$－assn $)^{k} *_{a}$ id－assn ${ }^{k} \rightarrow_{a}$ list－assn id－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－left ：：（array－assn id－assn）${ }^{k} \rightarrow_{a}$ array－assn （option－assn id－assn）
$\langle p r o o f\rangle$

## 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': (all-left, SPEC o all-left-spec)
    \in\langleId\ranglelist-rel }->\langle\langle\langleId\rangleoption-rel\ranglelist-rel\rangle nres-rel \, 
    <proof\rangle
```

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)
    \(\langle p r o o f\rangle\)
    Main correctness theorem as Hoare－Triple

```
theorem algo-impl-correct':
    <array-assn int-assn xs xsi>
        all-left-impl xsi
    <\lambdalfi. \exists
            * array-assn (option-assn id-assn) lf lfi
            * \uparrow(all-left-spec xs lf )>
    <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 algo－ rithm
$\langle M L\rangle$
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 $\mid$ Node int (lc: tree) (rc: tree)
Analogous to left-spec from 2.A.

## definition

right-spec xs $j=$
(if $(\exists i>j . x s!i<x s!j)$ then Some (LEAST $i . i>j \wedge x s!i<x s!j)$ else None)
context
fixes $x s$ :: 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 $x s!j>x s!p$
$\langle p r o o f\rangle$
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 foldfunction over a commutative operation concurrently, and prove it correct.

### 5.1 Misc

lemma (in comp-fun-commute) fold-mset-rewr: fold-mset fa(mset $l$ ) $=$ fold $f l a$ $\langle p r o o f\rangle$

```
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 $w f\{(x, y) . R y x\}$ and step: $\bigwedge x . R^{* *} a x \Longrightarrow P x \vee(\exists y . R x y)$
shows $\exists x$. $P x \wedge R^{* *} a x$
$\langle p r o o f\rangle$

### 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'
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

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

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 \(\Rightarrow\left({ }^{\prime} a,{ }^{\prime} b\right)\) state \(\Rightarrow\) bool where
    pick: step \((t \# t s, w s, s)(t s, w s(i:=\) Some \(t), s)\) if \(w s i=\) None and \(i<n\)
\(\mid\) exec: step \((t s, w s, s) \quad(t s, w s(i:=\) None \(), f a s)\) if \(w s i=\) Some \(a\) and \(i<n\)
lemma no-deadlock:
    assumes \(\neg\) final cfg
    shows \(\exists c f g^{\prime}\). step \(c f g c f g^{\prime}\)
    \(\langle p r o o f\rangle\)
lemma wf-step:
    \(w f\left\{\left(\left(t s^{\prime}, w s^{\prime}, r\right),(t s, w s, r)\right)\right.\).
    step \((t s, w s, r)\left(t s^{\prime}, w s^{\prime}, r^{\prime}\right) \wedge\) set \(t s^{\prime} \subseteq S \wedge\) dom ws \(\subseteq\{0 . .<n\} \wedge\) ran ws \(\left.\subseteq S\right\}\)
```

    if finite \(S\)
    $\langle p r o o f\rangle$
context
fixes $t s::$ 'a list and start $::$ ' $b$
begin
definition
$s_{0}=(t s, \lambda-$. None, start $)$
definition reachable $\equiv\left(\right.$ step $\left.^{* *}\right) s_{0}$
lemma reachable $0[$ simp $]$ : reachable $s_{0}$
$\langle p r o o f\rangle$
definition is-invar $I \equiv I s_{0} \wedge\left(\forall s s^{\prime}\right.$. reachable $s \wedge I s \wedge$ step $\left.s s^{\prime} \longrightarrow I s^{\prime}\right)$
lemma is-invarI[intro?]:
$\llbracket I s_{0} ; \bigwedge s s^{\prime}$. 【 reachable $s ; I s ;$ step $s s^{\prime} \rrbracket \Longrightarrow I s^{\prime} \rrbracket \Longrightarrow$ is-invar $I$
$\langle$ proof〉
lemma invar-reachable: is-invar $I \Longrightarrow$ reachable $s \Longrightarrow I s$
〈proof〉
definition
invar $\equiv \lambda(t s 2, w s, r)$.
( $\exists t s 1$.
mset $t s=t s 1+\{\#$ the (ws $i$ ). $i \in \#$ mset-set (dom ws $\cap\{0 . .<n\}) \#\}+$
mset ts2

```
    \wedger= fold-mset f start ts1
    set ts\mathcal{Z}\subseteq\mathrm{ set ts ^ ran ws }\subseteq\mathrm{ set ts ^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\rangle
lemma final-state-correct2:
    assumes reachable (ts', ms,r) final ( }t\mp@subsup{s}{}{\prime},ms,r
    shows r= fold-mset f start (mset ts)
    <proof\rangle
```

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\mp@subsup{s}{}{\prime}\mathrm{ . final s}\mp@subsup{s}{}{\prime}\wedge step** s s s
<proof\rangle
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 $=(n a t \times n a t) \times v a l$
We fix a size $n$ for the vector.

```
context
    fixes n :: nat
begin
```

An algorithm finishing triples in any order.

```
definition
    alg \((t s::\) 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 $(\lambda((r, c), v) y . y(c:=(y c:: v a l)+x r *$ v))
$\langle p r o o f\rangle$

### 6.2 Specification

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

$$
\begin{aligned}
& \text { definition } \alpha:: \text { triplet list } \Rightarrow(\text { nat } \times \text { nat }) \Rightarrow \text { val where } \\
& \alpha=\text { the-default } 0 \text { oo map-of }
\end{aligned}
$$

Abstract product.
definition pr m xi三 $\sum \mathrm{k}=0 . .<n . x k * m(k, i)$

### 6.3 Correctness

## lemma aux:

```
    distinct (map fst (ts1@ts2)) \(\Longrightarrow\)
    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 \((k, i))\)
```

```
<proof\rangle
```

lemma $1[$ simp]: distinct (map fst (ts1@ts2)) $\Longrightarrow$
$\operatorname{pr}(\alpha(t s 1 @ t s 2)) x i=p r(\alpha t s 1) x i+p r(\alpha t s 2) x i$
$\langle p r o o f\rangle$
lemmas $2=1[o f[((r, c), v)] t s$, simplified $]$ for $r c v t s$
lemma $[$ simp $]: \alpha[]=(\lambda-.0)\langle p r o o f\rangle$
lemma $[$ simp $]: \operatorname{pr}(\lambda-.0::$ val $) x=(\lambda-.0)$
$\langle p r o o f\rangle$
lemma aux3: the-default 0 (if b then Some x else None) $=($ if $b$ then $x$ else 0$)$
$\langle p r o o f\rangle$
lemma correct-aux: 【distinct (map fst ts); $\forall((r, c),-) \in$ set ts. $r<n \rrbracket$
$\Longrightarrow \forall i$.fold $(\lambda((r, c), v) y . y(c:=y c+x r * v)) t s m i=m i+p r(\alpha t s) x i$
$\langle p r o o f\rangle$
lemma correct-fold:
assumes distinct (map fst ts)
assumes $\forall((r, c),-) \in$ set ts. $r<n$
shows fold $(\lambda((r, c), v) y . y(c:=y c+x r * v)) t s(\lambda-.0)=p r(\alpha t s) x$
〈proof〉
lemma alg-by-fold: alg ts $x=$ fold $(\lambda((r, c), v) y . y(c:=y c+x r * v)) t s(\lambda-.0)$
$\langle p r o o f\rangle$
theorem correct：
assumes distinct（map fst ts）
assumes $\forall((r, c),-) \in$ set ts．$r<n$
shows alg ts $x=p r(\alpha t s) x$
$\langle p r o o f\rangle$

## 6．4 Multi－Threaded Implementation

Correctness of the parallel implementation：

```
theorem parallel-correct:
    assumes distinct (map fst ts) \forall((r,c),-)\inset ts. r<n
        and 0<n - At least on thread
        - We have reached a final state.
        and reachable x n ts (\lambda-.0) (ts',ms,r) final n (ts',ms,r)
    shows r=pr (\alphats)x
<proof>
```

We also know that the computation will always terminate.
theorem parallel-termination:
assumes $0<n$
and reachable $x n$ ts $(\lambda-0) s$
shows $\exists s^{\prime}$. final $n s^{\prime} \wedge(\text { step } x n)^{* *} s s^{\prime}$
〈proof〉
end - Context for fixed $n$.
end

