Imperative Insertion Sort

Christian Sternagel

June 11, 2019

Contents

1 Looping Constructs for Imperative HOL 1
   1.1 While Loops ........................................... 1
   1.2 For Loops ............................................. 4

2 Insertion Sort 6
   2.1 The Algorithm ........................................ 6
   2.2 Partial Correctness ................................. 7
   2.3 Total Correctness .................................. 12

1 Looping Constructs for Imperative HOL

theory Imperative-Loops
imports HOL−Imperative-HOL, Imperative-HOL
begin

1.1 While Loops

We would have liked to restrict to read-only loop conditions using a condition of type \( \text{heap} \Rightarrow \text{bool} \) together with \( \text{tap} \). However, this does not allow for code generation due to breaking the heap-abstraction.

partial-function (heap) while :: bool Heap \( \Rightarrow \) 'b Heap \( \Rightarrow \) unit Heap
where

\[
\text{[code]: while } p \ f = \begin{cases} 
  \text{do} \\
  b \leftarrow p; \\
  \text{if } b \text{ then } f \Rightarrow \text{while } p \ f \\
  \text{else return } () 
\end{cases}
\]

definition cond p h \leftarrow fst (the (execute p h))

A locale that restricts to read-only loop conditions.

locale ro-cond =
  fixes p :: bool Heap
assumes read-only: success p h \Rightarrow snd (\text{execute p h}) = h
begin

lemma ro-cond: ro-cond p
  using read-only by (simp add: ro-cond-def)

lemma cond-cases [execute-simps]:
  success p h \Rightarrow cond p h \Rightarrow execute p h = Some (True, h)
  success p h \Rightarrow \neg cond p h \Rightarrow execute p h = Some (False, h)
  using read-only [of h] by (auto simp: cond-def success-def)

lemma execute-while-unfolds [execute-simps]:
  success p h \Rightarrow cond p h \Rightarrow execute (while p f) h = execute (f \Rightarrow while p f) h
  success p h \Rightarrow \neg cond p h \Rightarrow execute (while p f) h = execute (\text{return} ()) h
  by (auto simp: while.simps execute-simps)

lemma success-while-cond: success p h \Rightarrow cond p h \Rightarrow effect f h h' r \Rightarrow success (while p f) h' 
  success (while p f) h and
  success-while-not-cond: success p h \Rightarrow \neg cond p h \Rightarrow success (while p f) h
  by (auto simp: while.simps effect-def execute-simps intro!: success-intros)

lemma success-cond-effect:
  success p h \Rightarrow cond p h \Rightarrow effect p h h True
  using read-only [of h] by (auto simp: effect-def execute-simps)

lemma success-not-cond-effect:
  success p h \Rightarrow \neg cond p h \Rightarrow effect p h h False
  using read-only [of h] by (auto simp: effect-def execute-simps)
end

The loop-condition does no longer hold after the loop is finished.

lemma ro-cond-effect-while-post:
  assumes ro-cond p
  and effect (while p f) h h' r
  shows success p h' \land \neg cond p h'
  using assms(1)
  apply (induct rule: while.raw-induct [OF - assms(2)])
  apply (auto elim!: effect-elims effect-ifE simp: cond-def)
  apply (metis effectE ro-cond.read-only)+
done

A rule for proving partial correctness of while loops.

lemma ro-cond-effect-while-induct:
  assumes ro-cond p
  assumes effect (while p f) h h' u
  and I h
and \( \land h h' u. I h \implies success p h \implies \text{cond} p h \implies \text{effect} f h h' u \implies I h' \)
successful shows \( I h' \)
using assms(1, 3)
proof (induction \( p f h h' u \) rule: while.raw-induct)
case \((1 w f h h' u)\)
obtain \( b \)
where \( \text{effect} p h h b \)
and \( *: \text{effect} (if \ b \ then \ f \ \gg \ w f \ else \ return()) h h' u \)
using 1.hyps and ro-cond \( p \)
by (auto elim!: effect-elims intro: effect-intros) (metis effectE ro-cond.read-only)
then have \( \text{cond}\): \( success p h \ \text{cond} p h = b \) by (auto simp: cond-def elim!: effect-elims effectE)
show \(?case\)
proof (cases \( b \))
assume \( \neg b \)
then show \(?thesis\)
using \( *\) and 1 by (auto elim: effect-elims)
next
assume \( b \)
moreover
with \( *\) obtain \( h'' \) and \( r \)
where \( \text{effect} f h h'' r \) and \( \text{effect}(w f) h'' h' u \) by (auto elim: effect-elims)
moreover
ultimately
show \(?thesis\)
using 1 and \( \text{cond}\) by blast
qed
qed

lemma effect-success-conv:
\((\exists h'. \text{effect} c h h'() \land I h') \iff success c h \land I (\text{snd}(\text{execute} c h)))\)
by (auto simp: success-def effect-def)

context ro-cond
begin

lemmas
\text{effect-while-post} = ro-cond-effect-while-post [OF ro-cond] and
\text{effect-while-induct} [consumes 1, case-names base step] = ro-cond-effect-while-induct
[OF ro-cond]

A rule for proving total correctness of while loops.

lemma wf-while-induct [consumes 1, case-names success-cond success-body base step]:
assumes \( \text{wf} \ R \) — a well-founded relation on heaps proving termination of the loop
and \( \text{success-p}: \ \land h. I h \implies success p h \) — the loop-condition terminates
and \( \text{success-f}: \ \land h. I h \implies success p h \implies \text{cond} p h \implies success f h \) — the loop-body terminates
and \( I h \) — the invariant holds before the loop is entered
and \( \text{step}: \ \land h h' r. I h \implies success p h \implies \text{cond} p h \implies \text{effect} f h h' r \implies \)
\[(h', h) \in R \land I h'\]

— the invariant is preserved by iterating the loop

shows \(\exists h'. \text{effect (while } p \ f \ h h' \ () \land I h'\)

using \(\langle \text{wf } R \rangle \text{ and } \langle I h \rangle\)

**proof** (induction \(h\))

**case** (less \(h\))

**show** \(?case\)

**proof** (cases cond \(p \ h\))

**assume** \(\neg \text{cond } p \ h\) **then show** \(?thesis\)

using \(\langle I h \rangle\) by (simp add: effect-def execute-simps)

**next**

**assume** \(\text{cond } p \ h\)

with \(\langle I h \rangle\) and \(\text{success-f } [of h]\) and \(\text{success-p } [of h]\)

obtain \(h'\) and \(r\) where \(\text{effect } f h h' r\) and \((h', h) \in R\) and \(I h'\) and \(\text{success } p h\)

by (auto simp: success-def effect-def)

with less.IH \[of h'\]

**show** \(?thesis\)

by (auto simp: execute-simps effect-def)

**qed**

**qed**

A rule for proving termination of while loops.

**lemmas**

**success-while-induct** \[consumes 1, case-names success-cond success-body base step\]

\[=\]

\[\langle \text{wf-while-induct} \rangle \text{ [unfolded effect-success-conv, THEN conjunct1}\]

**end**

**1.2 For Loops**

**fun** for :: 

\(\text{'}a\text{ list } \Rightarrow (\text{'}a \Rightarrow \text{'b } Heap) \Rightarrow \text{ unit } Heap\)

**where**

for [] \(f = \text{return } ()\)

for (x \# xs) \(f = f x \gg\) for xs \(f\)

A rule for proving partial correctness of for loops.

**lemma** effect-for-induct \[consumes 2, case-names base step\]:

**assumes** \(i \leq j\)

and \(\text{effect (for } [i \ ..< j] \ f \ h h' u\)

and \(I i h\)

and \(\forall k \ h h'. i \leq k \implies k < j \implies I k h \implies \text{effect } (f k) h h' r \implies I (\text{Suc } k) h'\)

shows \(I j h'\)

**using** \(\text{assms}\)

**proof** (induction \(j - i\) arbitrary: \(i h\))

**case** 0

then **show** \(?case\) by (auto elim: effect-elims)

**next**
case (Suc k)
show ?case
proof (cases j = i)
  case True
  with Suc show ?thesis by auto
next
case False
with (i ≤ j) and (Suc k = j - i)
  have i < j and k = j - Suc i and Suc i ≤ j by auto
then have [i..<j] = i # [Suc i..<j] by (metis upt-rec)
with \textit{effect} (for [i..<j] f) h h' w obtain h'' r
  where \*: \textit{effect} (f i) h h'' r and \**: \textit{effect} (for [Suc i..<j] f) h'' h' u
by (auto elim: effect-elims)
from Suc(6) [OF - - ⟨i ≤ j⟩] and (i < j)
  have I (Suc i) h'' by auto
show ?thesis
  by (rule Suc(1) [OF \(k = j - Suc i \) (Suc i ≤ j) \*\* (I (Suc i) h'' Suc(6))])

auto
qed

A rule for proving total correctness of for loops.

\textbf{lemma} for-induct [consumes 1, case-names succeed base step]:
  assumes i ≤ j
  and \(\forall k h. I k h \rightarrow i \leq k \rightarrow k < j \rightarrow success (f k) h\)
  and I i h
  and \(\forall k h h' r. I k h \rightarrow i \leq k \rightarrow k < j \rightarrow effect (f k) h h' r \rightarrow I (Suc k) h'\)
  shows \(\exists h'. effect (for [i..<j] f) h h' \land I j h' (is ?P i h)\)
using assms
proof (induction j - i arbitrary: i h)
  case 0
  then show ?case by (auto simp: effect-def execute-simps)
next
case (Suc k)
show ?case
proof (cases j = i)
  assume j = i
  with Suc show ?thesis by auto
next
assume j ≠ i
with (i ≤ j) and (Suc k = j - i)
  have i < j and k = j - Suc i and Suc i ≤ j by auto
then have \(\text{simp}: [i..<j] = i # [Suc i..<j]\) by (metis upt-rec)
obtain h' r where \*: \textit{effect} (f i) h h' r
  using Suc(4) [OF (I i h) le-refl (i < j)] by (auto elim!: success-effectE)
moreover
then have I (Suc i) h' using Suc by auto
moreover

have ?P (Suc i) h' 
by (rule Suc(1) [OF \k. k = j - Suc i \ (Suc i ≤ j) \ Suc(4) \ I (Suc i) h' \ Suc(6)])
auto ultimately
ultimately show ?case by (auto simp: effect-def execute-simps)
qed

A rule for proving termination of for loops.

lemmas
success-for-induct [consumes 1, case-names succeed base step] =
for-induct [unfolded effect-success-conv, THEN conjunct1]

end

2 Insertion Sort

theory Imperative-Insertion-Sort
imports
Imperative-Loops
HOL-Library.Multiset
begin

2.1 The Algorithm

abbreviation
array-update :: 'a::heap array ⇒ nat ⇒ 'a ⇒ 'a array Heap ((-,('-')) ←/ -) [1000, 0, 13] 14)
where
a.(i) ← x ≡ Array upd i x a

abbreviation array-nth :: 'a::heap array ⇒ nat ⇒ 'a Heap ((-,('-')) [1000, 0] 14)
where
a.(i) ≡ Array nth a i

A definition of insertion sort as given by Cormen et al. in Introduction to Algorithms. Compared to the informal textbook version the variant below is a bit unwieldy due to explicit dereferencing of variables on the heap.

To avoid ambiguities with existing syntax we use OCaml’s notation for accessing (a.(i)) and updating (a.(i) ← x) an array a at position i.

definition
insertion-sort a = do {
  l ← Array.len a;
  for [1 ..< l] (λj. do {
    — Insert a[j] into the sorted subarray a[1 .. j - 1],
    key ← a.(j);
    i ← ref j;
    while (do {

The following definitions decompose the nested loops of the algorithm into more manageable chunks.

definition shiftr-p a (key::a::{heap, linorder}) i =
  (do {i' ← ! i;
    x ← a.(i' − 1);
    a.(i') ← x;
    i := i' − 1
  });
  i' ← ! i;
  a.(i') ← key
})
definition shiftr-f a i = do {
  i' ← ! i;
  x ← a.(i' − 1);
  a.(i') ← x;
  i := i' − 1
}
definition shiftr a key i = while (shiftr-p a key i) (shiftr-f a i)
definition insert-elt a = (λj. do {
  key ← a.(j);
  i ← ref j;
  shiftr a key i;
  i' ← ! i;
  a.(i') ← key
})
definition sort-upto a = (λl. for [1..< l] (insert-elt a))

lemma insertion-sort-alt-def:
  insertion-sort a = (Array.len a ≥ sort-upto a)
  by (simp add: insertion-sort-def sort-upto-def shiftr-def shiftr-p-def shiftr-f-def insert-elt-def)

2.2 Partial Correctness

lemma effect-shiftr-f:
  assumes effect (shiftr-f a i) h h' u
  shows Ref.get h' i = Ref.get h i − 1 ∧
\[
\begin{align*}
\text{Array.get } h' a &= \text{list-update} \ (\text{Array.get } h a) \ (\text{Ref.get } h i) \ (\text{Array.get } h a ! (\text{Ref.get } h i - 1)) \\
\text{using } \text{assms } \text{by } (\text{auto simp: shiftr-f-def elim!: effect-elims})
\end{align*}
\]

\textbf{lemma success-shiftr-p:}
\[
\text{Ref.get } h i < \text{Array.length } h a \implies \text{success} (\text{shiftr-p } a \text{ key } i ) h
\]
\text{by } (\text{auto simp: success-def shiftr-p-def execute-simps})

\textbf{interpretation ro-shiftr-p: ro-cond shiftr-p a key i for a key i}
\text{by } (\text{unfold-locales})
\text{(auto simp: shiftr-p-def success-def execute-simps execute-bind-case split: option.split, metis effectI effect-nthE})

\textbf{definition [simp]: ini } h a j = \text{take } j \ (\text{Array.get } h a)

\textbf{definition [simp]: left } h a i = \text{take} (\text{Ref.get } h i) \ (\text{Array.get } h a)

\textbf{definition [simp]: right } h a i j = \text{take} (\text{Ref.get } h i - 1) \ (\text{Array.get } h a ! (\text{Ref.get } h i - 1))

\textbf{definition [simp]: both } h a j i = \text{left } h a i @ \text{right } h a j i

\textbf{lemma effect-shiftr:}
\text{assumes } \text{Ref.get } h i = j \ (\text{is } ?i h = -) \\
\quad \text{and } j < \text{Array.length } h a \\
\quad \text{and sorted } (\text{take } j \ (\text{Array.get } h a)) \\
\quad \text{and effect } (\text{while } (\text{shiftr-p } a \text{ key } i) (\text{shiftr-f } a \text{ i})) h h' u
\text{shows} \text{Array.length } h a = \text{Array.length } h' a \land
\quad ?i h' \leq j \land \\
\quad \text{mset } (\text{list-update } (\text{Array.get } h a) \ j \text{ key}) = \\
\quad \text{mset } (\text{list-update } (\text{Array.get } h' a) \ (?i h') \text{ key}) \land \\
\quad \text{ini } h a j = \text{both } h' a j i \land \\
\quad \text{sorted } (\text{both } h' a j i) \land \\
\quad (\forall x \in \text{set } (\text{right } h' a j i). \ x > \text{key})
\text{using } \text{assms}(4, 2)
\text{proof } (\text{induction rule: ro-shiftr-p.effect-while-induct})
\text{case base}
\text{show } ?\text{case using } \text{assms } \text{by auto}
\text{next}
\text{case } (\text{step } h' h'' u)
\text{from } (\text{success } (\text{shiftr-p } a \text{ key } i) h') \ h'' \text{ and } (\text{cond } (\text{shiftr-p } a \text{ key } i) h') h''
\text{have } ?i h' > 0 \text{ and }
\text{key: } \text{Array.get } h' a! (\text{?i } h' - 1) > \text{key}
\text{by } (\text{auto dest!: ro-shiftr-p.success-cond-effect})
\text{(auto simp: shiftr-p-def elim!: effect-elims effect-ifE)}
\text{from } \text{effect-shiftr-f} \ [\text{OF } (\text{effect } (\text{shiftr-f } a \text{ i}) h' h'' u)]
\text{have } [\text{simp}]: ?i h'' = ?i h' - 1
\text{Array.get } h'' a = \text{list-update } (\text{Array.get } h' a) \ (?i h') (\text{Array.get } h' a ! (\text{?i } h' - 1) \text{ )}
by auto

from step have ∗: ?i h′ < length (Array.get h′ a)
and ∗∗: ?i h′ − (Suc 0) ≤ ?i h′ ?i h′ ≤ length (Array.get h′ a)
and ?i h′ ≤ j
and ?i h′ < Suc j
and IH: ini h a j = both h′ a j i
by (auto simp add: Array.length-def)

have Array.length h a = Array.length h′′ a using step by (simp add: Array.length-def)
moreover
have ?i h′′ ≤ j using step by auto
moreover
have mset (list-update (Array.get h a) j key) =
  mset (list-update (Array.get h′′ a) (?i h′′) key)
proof –
  have ?i h′ < length (Array.get h′ a)
  and ?i h′ − 1 < length (Array.get h′ a) using ∗ by auto
then show ?thesis
  using step by (simp add: mset-update ac-simps nth-list-update)

qed

moreover
have ini h a j = both h′′ a j i
  using ⟨0 < ?i h′⟩ and ⟨?i h′ ≤ j⟩ and ⟨?i h′ < length (Array.get h′ a)⟩ and
  ∗∗ IH
  by (auto simp: upd-conv-take-nth-drop Suc-diff-le min-absorb1)
  (metis Suc-lessD Suc-pred append.simps append-assoc take-Suc-conv-app-nth)
moreover
have sorted (both h′′ a j i)
  using step and ⟨0 < ?i h′⟩ and ⟨?i h′ ≤ j⟩ and ⟨?i h′ < length (Array.get h′ a)⟩ and
  ∗∗ IH
  by (auto simp: IH upd-conv-take-nth-drop Suc-diff-le min-absorb1)
  (metis Suc-lessD Suc-pred append.simps append-assoc take-Suc-conv-app-nth)
moreover
have ∀x ∈ set (right h′′ a j i). x > key
  using step and ⟨0 < ?i h′⟩ and ⟨?i h′ < length (Array.get h′ a)⟩ and key
  by (auto simp: upd-conv-take-nth-drop Suc-diff-le)
ultimately show ?case by blast

lemma sorted-take-nth:
assumes 0 < i and i < length xs and xs ! (i - 1) ≤ y
and sorted (take i xs)
shows ∀x ∈ set (take i xs). x ≤ y
proof –
have take i xs = take (i - 1) xs @ [xs ! (i - 1)]
  using ⟨0 < i⟩ and ⟨i < length xs⟩
  by (metis Suc-diff-1 less-imp-diff-less take-Suc-conv-app-nth)
then show \textit{thesis}
  using \texttt{sorted (take i xs) \textbf{and} (xs ! (i - 1) \leq y)}
  by (auto simp: sorted-append)
qed

\textbf{lemma} effect-for-insert-elt:
\textbf{assumes} \( I \leq \text{Array}\_\text{length} h a \)
\textbf{and} \( I \leq l \)
\textbf{and} effect (\texttt{for [1 ..< l] (insert-elt a)}) \( h h' u \)
\textbf{shows} \( \text{Array}\_\text{length} h a = \text{Array}\_\text{length} h' a \wedge \)
\texttt{sorted (take l (Array.get h' a)) \wedge}
\texttt{mset (Array.get h a) = mset (Array.get h' a)}
\textbf{using} \texttt{assms(2-)}
\textbf{proof (induction l h' rule: effect-for-induct)}
\textbf{case} base
\textbf{show} \( ?\text{case} \textbf{by} \texttt{(cases Array.get h a) simp-all} \)
\textbf{next}
\textbf{case} \( (\text{step} j h' h'' u) \)
\textbf{with} \texttt{assms(1)} \textbf{have} \( j < \text{Array}\_\text{length} h' a \textbf{by auto} \)
\textbf{from} \texttt{step} \textbf{have} \texttt{sorted: sorted (take j (Array.get h' a)) by blast} \)
\textbf{from} \texttt{step(3)} [\texttt{unfolded insert-elt-def}]
\textbf{obtain} \texttt{key and h1 and i and h2 and i'}
\textbf{where} \texttt{key: key = Array.get h' a ! j} \)
\textbf{and} \texttt{effect (ref \texttt{j}) h' h1 i} \)
\textbf{and} \texttt{ref1: Ref.get h1 i = j} \)
\textbf{and} \texttt{shiftr': effect (shiftr a key i) h1 h2 \{}} \)
\textbf{and} \texttt{[simp]: Ref.get h2 i = i'} \)
\textbf{and} \texttt{[simp]: h'' = Array.update a i' key h2} \)
\textbf{and} \texttt{i' < Array.length h2 a} \)
\textbf{by (elim effect-bindE effect-nthE effect-lookupE effect-updE)}
\textbf{\texttt{(auto intro: effect-intros,metis effect-refE)}}
\texttt{from (effect (ref \texttt{j}) h' h1 i) have [simp]: Array.get h1 a = Array.get h' a}
\textbf{by (metis array-get-alloc effectE execute-ref option.sel)}
\textbf{have [simp]: Array.length h1 a = Array.length h' a by (simp add: Array.length-def)}
\texttt{from step and assms(1)}
\textbf{have} \( j < \text{Array}\_\text{length} h' a \texttt{sorted (take j (Array.get h1 a)) by auto} \)
\textbf{note shiftr' = effect-shiftr [OF ref1 this shiftr' [unfolded shiftr-def], simplified]}
\textbf{have} \( i' \leq j \textbf{using shiftr by simp} \)
\textbf{have} \( i' < \text{length} (\text{Array.get h2 a}) \)
\textbf{by (metis \( i' < \text{Array}\_\text{length} h2 a \texttt{ length-def})}
\textbf{have [simp]: \texttt{min (Suc j) i' = i' using \( i' \leq j \texttt{ by simp} \)
\textbf{have [simp]: \texttt{min (length (Array.get h2 a)) i' = i'} \)
\textbf{using \( i' < \text{length} (\text{Array.get h2 a}) \texttt{ by (simp)}}
\textbf{have take-Suc-j: \texttt{take (Suc j) (list-update (Array.get h2 a) i' key)} =
\texttt{take i' (Array.get h2 a) \&\& take \( (j - i') \texttt{ (drop \( \Suc i') (Array.get h2 a)}) \)}}
unfolding upd-cone-take-nth-drop [OF \ i' < length (Array.get h_2 a)]:
by (auto) (metis Suc-diff-le (\ i' \leq \ j) take-Suc-Cons)

have Array.length h a = Array.length h'' a
using shiftr by (auto) (metis step.IH)
moreover
have mset (Array.get h a) = mset (Array.get h'' a)
using shiftr and step by (simp add: key)
moreover
have sorted (take (Suc j) (Array.get h'' a))
proof –
from ro-shiftr-p.effect-while-post [OF shiftr' [unfolded shiftr-def]]
have \ i' = 0 \lor (0 < \ i' \land key \geq Array.get h_2 a ! (\ i' - 1))
by (auto dest!: ro-shiftr-p.success-not-cond-effect)
(auto elim!: effect-elims simp add: shiftr-p-def)
next
assume [simp]: \ i' = 0
have \*: take (Suc j) (list-update (Array.get h_2 a) 0 key) =
key \# take j (drop 1 (Array.get h_2 a))
by (simp) (metis \ i' = 0: append-Nil take-Suc-j diff-zero take-0)
from sorted and shiftr
have sorted (take j (drop 1 (Array.get h_2 a)))
and \ \forall x \in set (take j (drop 1 (Array.get h_2 a))). key < x by simp-all
then have sorted (key \# take j (drop 1 (Array.get h_2 a)))
by (metis less-imp-le sorted.simps(2))
then show \?thesis by (simp add: *)
next
assume 0 < \ i' \land key \geq Array.get h_2 a ! (\ i' - 1)
moreover
have sorted (take \ i' (Array.get h_2 a) \@ take (j - \ i') (drop (Suc \ i') (Array.get h_2 a)))
and \ \forall x \in set (take \ i' (Array.get h_2 a)). key < x
using shiftr by auto
ultimately have \ \forall x \in set (take \ i' (Array.get h_2 a)). x \leq key
using sorted-take-nth [OF \ -\ i' < length (Array.get h_2 a)]. af key
by (simp add: sorted-append)
then show \?thesis
using shiftr by (auto simp: take-Suc-j sorted-append less-imp-le)
qed
qed
ultimately
show \?case by blast
qed

lemma effect-insertion-sort:
assumes effect (insertion-sort a) \ h \ h' \ u
shows mset (Array.get h a) = mset (Array.get h' a) \& sorted (Array.get h' a)
using assms
apply (cases Array.length h a)
apply (auto elim!:: effect-elims simp: insertion-sort-def Array.length-def)[1]
unfolding insertion-sort-def
unfolding shiftr-p-def [symmetric] shiftr-f-def [symmetric]
unfolding shiftr-def [symmetric] insert-elt-def [symmetric]
apply (elim effect-elims)
apply (simp only:)
apply (subgoal-tac Suc nat ≤ Array.length h a)
apply (drule effect-for-insert-elt)
apply (auto simp: Array.length-def)
done

2.3 Total Correctness

lemma success-shiftr-f:
  assumes Ref.get h i < Array.length h a
  shows success (shiftr-f a i) h
  using assms by (auto simp: success-def shiftr-f-def execute-simps)

lemma success-shiftr:
  assumes Ref.get h i < Array.length h a
  shows success (while (shiftr-p a key i) (shiftr-f a i)) h
proof –
  have wf (measure (λh. Ref.get h i)) by (metis wf-measure)
  then show ?thesis
  proof (induct taking: λh. Ref.get h i < Array.length h a rule: ro-shiftr-p.success-while-induct)
    case (success-cond h)
    then show ?case by (metis success-shiftr-p)
  next
    case (success-body h)
    then show ?case by (blast intro: success-shiftr-f)
  next
    case (step h h' r)
    then show ?case
      by (auto dest!: effect-shiftr-f ro-shiftr-p.success-cond-effect simp: length-def)
        (auto simp: shiftr-p-def elim!: effect-elims effect-ifE)
qed fact

lemma effect-shiftr-index:
  assumes effect (shiftr a key i) h h' u
  shows Ref.get h' i ≤ Ref.get h i
  using assms unfolding shiftr-def
  by (induct h' rule: ro-shiftr-p.effect-while-induct) (auto dest: effect-shiftr-f)

lemma effect-shiftr-length:
  assumes effect (shiftr a key i) h h' u
  shows Array.length h' a = Array.length h a
  using assms unfolding shiftr-def
lemma success-insert-elt:
  assumes k < Array.length h a
  shows success (insert-elt a k) h
proof –
  obtain key where effect (a.(k)) h h key
  using assms by (auto intro: effect-intros)
moreover 
  obtain i and h₁ where effect (ref k) h₁ h₁ i
  and [simp]: Ref.get h₁ i = k
  and [simp]: Array.length h₁ a = Array.length h a
  by (auto simp: ref-def length-def)
moreover 
  obtain h₂ where *: effect (shiftr a key i) h₁ h₂ ()
  using success-shiftr [of h₁ i a key] and assms
  by (auto simp: success-def effect-def shiftr-def)
moreover 
  have effect (! i) h₂ h₂ (Ref.get h₂ i)
  and Ref.get h₂ i ≤ Ref.get h₁ i
  and Ref.get h₂ i < Array.length h₂ a
  using effect-shiftr-index [OF *] and effect-shiftr-length [OF *] and assms
  by (auto intro!: effect-intros)
moreover 
  then obtain h₃ and r where effect (a.(Ref.get h₂ i) ← key) h₂ h₃ r
  using assms by (auto simp: execute-simps)
ultimately 
  have effect (insert-elt a k) h₃ h₃ r
  by (auto simp: insert-elt-def intro: effect-intros)
then show ?thesis by (metis effectE)
qed

lemma for-insert-elt-correct:
  assumes l ≤ Array.length h a
  and 1 ≤ l
  shows ∃ h'. effect (for [1 ..< l] (insert-elt a)) h h' () ∧
  Array.length h a = Array.length h' a ∧
  sorted (take l (Array.get h' a)) ∧
  mset (Array.get h a) = mset (Array.get h' a)
using assms(2)
proof (induction rule: for-induct)
  case (succeed k h)
  then show ?case using assms and success-insert-elt [of k h a] by auto
next
  case base
  show ?case by (cases Array.get h a) simp-all
next
  case (step j h' h'' u)
with assms(1) have \( j < \text{Array.length}\ h'\ a \) by auto
from step have sorted: sorted (take \( j \) (Array.get \( h'\ a \))) by blast
from step(4) [unfolded insert-elt-def]
  obtain key and \( h_1 \) and \( i \) and \( h_2 \) and \( i' \)
where key: \( key = \text{Array.get}\ h'\ a \ ! j \)
  and effect (ref \( j \)) \( h'^i h_1 \)
  and ref\( _1: \text{Ref.get}\ h_1\ i = j \)
  and shiftr\( ': \text{effect}\ (\text{shiftr}\ a\ key\ i)\ h_1\ h_2\ () \)
  and [simp]: \( \text{Ref.get}\ h_2\ i = i' \)
  and [simp]: \( h'' = \text{Array.update}\ a\ i'\ key\ h_2 \)
  and \( i' < \text{Array.length}\ h_2\ a \)
by (elim effect-bindE effect-nthE effect-lookupE effect-updE)
  (auto intro: effect-intros, metis effect-refE)

from (effect (ref \( j \)) \( h'^i h_1 \)) have [simp]: \( \text{Array.get}\ h_1\ a = \text{Array.get}\ h'\ a \)
by (metis array-get-alloc effectE execute-ref option.sel)

have [simp]: \( \text{Array.length}\ h_1\ a = \text{Array.length}\ h'\ a \) by (simp add: Array-length-def)

from step and assms(1)
  have \( j < \text{Array.length}\ h_1\ a \) sorted (take \( j \) (Array.get \( h_1\ a \))) by auto
note shiftr = effect-shiftr [OF ref\( _1\) this shiftr\( '\) [unfolded shiftr-def], simplified]
have \( i' \leq j \) using shiftr by simp

have \( i' < \text{length}\ (\text{Array.get}\ h_2\ a) \)
  by (metis \( i' < \text{Array.length}\ h_2\ a \) \( \text{length-def} \))
have [simp]: \( \text{min}\ (\text{Suc}\ j)\ i' = i'\) using \( i' \leq j \) by simp
have [simp]: \( \text{min}\ (\text{length}\ (\text{Array.get}\ h_2\ a))\ i' = i' \)
  using \( i' < \text{length}\ (\text{Array.get}\ h_2\ a) \) by (simp)
have take-Suc-j: take (Suc \( j \)) (list-update (Array.get \( h_2\ a \)) \( i' \) key) =
  take i' (Array.get \( h_2\ a \)) @ key # take (\( j - i' \)) (drop (Suc i') (Array.get \( h_2\ a \)))
unfolding upd-cone-take-nth-drop [OF \( i' < \text{length}\ (\text{Array.get}\ h_2\ a) \)]
by (auto) (metis Suc-diff-le \( i' \leq j \) \text{take-Suc-Cons})

have Array.length \( h\ a = \text{Array.length}\ h''\ a \)
  using shiftr by (auto) (metis step.hyps(1))
moreover
have mset (Array.get \( h\ a \)) = mset (Array.get \( h''\ a \))
  using shiftr and step by (simp add: key)
moreover
have sorted (take (Suc \( j \)) (Array.get \( h''\ a \)))
proof -
  from ro-shiftr-p.effect-while-post [OF shiftr\' \( \text{unfolded}\ \text{shiftr-def} \)]
  have \( i' = 0 \lor (0 < i' \land \text{key} \geq \text{Array.get}\ h_2\ a \ ! (i' - 1)) \)
  by (auto dest!: ro-shiftr-p.success-not-cond-effect)
  (auto elim!: effect-elims simp: shiftr-p-def)
then show ?thesis
proof
  assume [simp]: \( i' = 0 \)
have *: take (Suc j) (list-update (Array.get h2 a) 0 key) =
  key # take j (drop 1 (Array.get h2 a))
by (simp) (metis \( i' = 0 \) append-Nil take-Suc-j diff-zero take-0)
from sorted and shiftr
  have sorted (take j (drop 1 (Array.get h2 a)))
  and \( \forall x \in \text{set} \ (\text{take } j \ (\text{drop } 1 \ (\text{Array.get } h2 \ a))) \). key < x by simp-all
then have (key # take j (drop 1 (Array.get h2 a)))
  by (metis less-imp-le sorted simps 2)
then show ?thesis by (simp add: *)
next
assume 0 < \( i' \) \& key \( \geq \) Array.get h2 a ! (\( i' - 1 \))
moreover
have (take \( i' \) (Array.get h2 a) @ take (j - \( i' \)) (drop (Suc \( i' \)) (Array.get h2 a)))
  and \( \forall x \in \text{set} \ (\text{take } (j - \( i' \)) \ (\text{drop } (\text{Suc } i') \ (\text{Array.get } h2 \ a))) \). key < x
  using shiftr by auto
ultimately have \( \forall x \in \text{set} \ (\text{take } i' \ (\text{Array.get } h2 \ a)) \). x \( \leq \) key
  using sorted-take-nth [OF - \( i' < \text{length } (\text{Array.get } h2 \ a) \), of key]
  by (simp add: sorted-append)
then show ?thesis
  using shiftr by (auto simp: take-Suc-j sorted-append less-imp-le)
qed
qed
ultimately
show ?case by blast
qed

lemma insertion-sort-correct:
\exists h'. effect (insertion-sort a) h h' u \&
mset (Array.get h a) = mset (Array.get h' a) \&
sorted (Array.get h' a)
proof (cases Array.length h a = 0)
assume Array.length h a = 0
then have effect (insertion-sort a) h h ()
  and mset (Array.get h a) = mset (Array.get h a)
  and sorted (Array.get h a)
  by (auto simp: insertion-sort-def length-def intro!: effect-intros)
then show ?thesis by auto
next
assume Array.length h a \( \neq \) 0
then have 1 \( \leq \) Array.length h a by auto
from for-insert-ell-correct [OF le-refl this]
  show ?thesis
    by (auto simp: insertion-sort-alt-def sort-upto-def)
(metis One-nat-def effect-bindI effect-insertion-sort effect-lengthI insertion-sort-alt-def sort-upto-def)
qed

export-code insertion-sort in Haskell