Imperative Insertion Sort

Christian Sternagel

August 16, 2018

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 Begin
begin

1.1 While Loops

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

\textbf{partial-function} (\textit{heap}) while :: \textit{bool} \textit{Heap} $\Rightarrow$ \textit{\textacute{'}b} \textit{Heap} $\Rightarrow$ \textit{unit} \textit{Heap}

\textbf{where}

[code]: while \textit{p} \textit{f} = do {
  \textit{b} $\leftarrow \textit{p}$;
  if \textit{b} then \textit{f} $\gg$ while \textit{p} \textit{f}
  else return ()
}

definition cond \textit{p} \textit{h} $\leftrightarrow$ \textit{fst} (\textit{the} (execute \textit{p} \textit{h}))

A locale that restricts to read-only loop conditions.

\textbf{locale} ro-cond =
  \textbf{fixes} \textit{p} :: \textit{bool} \textit{Heap}
assumes read-only: success p h \implies \text{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 \implies \text{cond} p h \implies \text{execute} p h = \text{Some} (\text{True}, h)
  success p h \implies \\neg \text{cond} p h \implies \text{execute} p h = \text{Some} (\text{False}, h)
  using read-only [of h] by (auto simp: cond-def success-def)

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

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

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

lemma success-not-cond-effect:
  success p h \implies \neg \text{cond} p h \implies \text{effect} p h h \text{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 \text{cond} p h'
  using assms(1)
  apply (induct rule: while.raw-induct [OF - assms(2)])
  apply (auto elim!: effect-elims effect-ifE simp: 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 \( \land h h' \ u. \ I h \implies \text{success } p 
\land h \implies \text{cond } p 
\land h \implies \text{effect } f 
\land h \implies I h' \)
shows \( I h' \)
using \text{assms}(1, 3−)

**proof** (induction \( p f h h' u \) rule: while.raw-induct)

\text{case } (1 \ w p f h h' u)

obtain \( b \)
where \text{effect } p h h b
and \( * : \text{effect } (\text{if } b \text{ then } f \ \text{else } \text{return }()) 
\land h' \ u 
\)
using 1.hyps and \text{ro-cond } p
by \text{(auto elim! : effect-elims intro: effect-intros)} \text{(metis effectE ro-cond.read-only)}

then have \text{cond: } success \ p 
\land h \ \text{cond } p 
\land h' \ u = b \text{ by (auto simp: cond-def elim!: effect-elims effectE)}

\text{show } ?case
\text{proof } (\text{cases } b)
\text{assume } \neg b
then show ?thesis using \( * \) and \( \langle I h \rangle \)
by \text{(auto elim: effect-elims)}

next
assume \( b \)
moreover
with \( * \) obtain \( h'' \) and \( r \)
where \text{effect } f 
\land h'' \ r 
\land \text{effect } (w p f) 
\land h'' \ u 
\text{by (auto elim: effect-elims)}

moreover
ultimately
\text{show } ?thesis using 1 \text{ and } \text{cond by blast}

qed

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

by \text{(auto simp: success-def effect-def)}

**context** \text{ro-cond}

begin

**lemmas**

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

\text{[OF ro-cond]}

A rule for proving total correctness of while loops.

**lemma** \text{wf-while-induct } \text{[consumes 1, case-names success-cond success-body base step]}:

\text{assumes } \text{wf } R \ — \ a \text{ well-founded relation on heaps proving termination of the loop}

\text{and } \text{success-p: } \land h. \ I h \implies \text{success } p 
\land h \implies \text{the loop-condition terminates}
\text{and } \text{success-f: } \land h. \ I h \implies \text{success } p 
\land h \implies \text{success } f 
\land h \implies \text{the loop-body terminates}
\text{and } I h \ — \ the \ invariant \ holds \ before \ the \ loop \ is \ entered
\text{and } \text{step: } \land h h' \ r. \ I h \implies \text{success } p 
\land h \implies \text{cond } p 
\land h \implies \text{effect } f 
\land h' \ r \implies \text{effect } f 
\land h' \ r \implies \text{effect } f 
\land h' \ r \implies \text{effect } f 
\land h' \ r \implies \text{effect } f
\[(h', h) \in R \land I h'\]

--- the invariant is preserved by iterating the loop

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

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

proof (induction \(h\))

case (\text{less } h)

show ?case

proof (cases \text{cond } p h)

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

using \(I h\) and \text{success-p \([of } h\) by \(\text{simp add: effect-def execute-simps}\)

next

assume \text{cond } p h

with \(I h\) and \text{success-f \([of } h\) and \text{step \([of } h\) 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 \(\text{auto simp: success-def effect-def}\)

with \text{less.IH \([of } h'\) show ?thesis

using \text{cond } p h by \(\text{auto simp: execute-simps effect-def}\)

qed

qed

A rule for proving termination of while loops.

lemmas

success-while-induct \([\text{consumes 1, case-names success-cond success-body base step}]\]

= \(\text{wf-while-induct \([unfolded effect-success-conv, THEN conjunct1]\)}\)

end

1.2 For Loops

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

where

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

\(\text{for } (x \# xs) f = f x \gg \text{for } xs f\)

A rule for proving partial correctness of for loops.

lemma effect-for-induct \([\text{consumes 2, case-names base step}]\):

assumes \(i \leq j\)

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

and \(I i h\)

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

shows \(I j h'\)

using \text{assms}

proof (induction \(j - i\) arbitrary; \(i h\))

case 0

then show ?case by \(\text{auto elim: effect-elims}\)

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 (effect (for [i..<j] f) h h' u obtain h'' r
  where *: effect (f i) h h'' r and **: effect (for [Suc i..<j] f) h'' h' u
by (auto elim: effect-elims)
from Suc(6) [OF - - ⟨I i h⟩] 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
qed

A rule for proving total correctness of for loops.

lemma for-induct [consumes 1, case-names succeed base step]:
  assumes i ≤ j
  and ∃k h. I k h ⇒ i ≤ k ⇒ k < j ⇒ success (f k) h
  and I i h
  and ∃k h h' r. I k h ⇒ i ≤ k ⇒ k < j ⇒ effect (f k) h h' r ⇒ I (Suc k) h'
  shows ∃h'. effect (for [i..<j] f) h 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 [simp]: [i..<j] = i # [Suc i..<j] by (metis upt-rec)
  obtain h' r where *: 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 \( \exists P \) (Suc \( i \)) \( h' \)
by (rule Suc(1)) [OF \( \langle k = j = \text{Suc} \ i \rangle \) \( \langle \text{Suc} \ i \leq j \rangle \) Suc(4) \( \langle I \ (\text{Suc} \ i \ h') \ (\text{Suc} 6) \rangle \)]
auto
ultimately
show \( \exists \) case by (auto simp: effect-def execute-simps)
qed
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
\( a.(i) ← x \equiv \text{Array.upd} \ i \ x \ a \)

abbreviation array-nth :: 'a::heap array ⇒ nat ⇒ 'a ⇒ 'a Heap (-.'(.-) [1000, 0] 14)
where
\( a.(i) \equiv \text{Array.nth} \ a \ i \)

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] (\( j \)) do {
    — Insert \( a[j] \) into the sorted subarray \( a[1..j-1] \),
    key ← a.(j);
    i ← ref \( j \);
    while (do {
    ...
\( i' \leftarrow ! i; \)
if \( i' > 0 \) then do \( \{ x \leftarrow a.(i' - 1); \text{return} (x > \text{key}) \} \)
else return False \}
\}

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

**Definition** \( \text{shiftr-p} \ a \ (\text{key}::a::\{ \text{heap}, \text{linorder} \}) \ i = \)
\( \{ \text{do} \{ i' \leftarrow ! i; \}
\text{x} \leftarrow a.(i' - 1); \)
a.(i') \leftarrow x; \)
i := i' - 1 \}
\}
i' \leftarrow ! i;
a.(i') \leftarrow \text{key} \}
\}

**Definition** \( \text{shiftr-f} \ a \ i = \text{do} \{ \)
i' \leftarrow ! i;
\text{x} \leftarrow a.(i' - 1); \)
a.(i') \leftarrow x; \)
i := i' - 1 \}
\}

**Definition** \( \text{shiftr} \ a \ i = \text{while} (\text{shiftr-p} a \ \text{key} \ i) (\text{shiftr-f} a \ i) \)

**Definition** \( \text{insert-elt} \ a = (\lambda \text{j}. \text{do} \{ \)
\text{key} \leftarrow a.\text{j}; \)
i \leftarrow \text{ref} \text{j}; \)
\text{shiftr} a \ \text{key} \ i; \)
i' \leftarrow ! i;
a.(i') \leftarrow \text{key} \}
\}}

**Definition** \( \text{sort-upto} \ a = (\lambda \text{l}. \text{for} [1..<\text{l}] (\text{insert-elt} a)) \)

**Lemma** \( \text{insertion-sort-alt-def}: \)
\( \text{insertion-sort} \ a = (\text{Array.len} a \geq \text{sort-upto} a) \)
by (simp add: \( \text{insertion-sort-def sort-upto-def shiftr-def shiftr-p-def shiftr-f-def insert-elt-def} \))

**2.2 Partial Correctness**

**Lemma** \( \text{effect-shiftr-f}: \)
assumes \( \text{effect (shiftr-f a i) h h' u} \)
show\( s \text{Ref.get h' i = Ref.get h i - 1} \land \)
Array.get h’a = list-update (Array.get h a) (Ref.get h i) (Array.get h a ! (Ref.get h i - 1))
using assms by (auto simp: shiftr-f-def elim!: effect-elims)

lemma success-shiftr-p:
Ref.get h i < Array.length h a ⇒ success (shiftr-p a key i) h
by (auto simp: success-def shiftr-p-def execute-simps)

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

definition [simp]: ini h a j = take j (Array.get h a)
definition [simp]: left h a i = take (Ref.get h i) (Array.get h a)
definition [simp]: right h a i j = take (j - Ref.get h i + 1) (Array.get h a))
definition [simp]: both h a j i = left h a i @ right h a j i

lemma effect-shiftr:
assumes Ref.get h i = j (is ?i h = -)
and j < Array.length h a
and sorted (take j (Array.get h a))
and effect (while (shiftr-p a key i) (shiftr-f a i)) h’ u
shows Array.length h a = Array.length h’ a ∧
?i h’ ≤ j ∧
\mset (list-update (Array.get h a) j key) = 
\mset (list-update (Array.get h’ a) (?i h’) key) ∧
ini h a j = both h’ a j i ∧
sorted (both h’ a j i) ∧
(∀x ∈ set (right h’ a j i). x > key)
using assms(4, 2)

proof (induction rule: ro-shiftr-p.effect-while-induct)
case base
  show ?case using assms by auto
next
case (step h’ h’’ u)
from (success (shiftr-p a key i) h’) h’ ∧ (cond (shiftr-p a key i) h’) h’
have ?i h’ > 0 and
key: Array.get h’ a ! (?i h’ - 1) > key
by (auto dest!: ro-shiftr-p.success-cond-effect)
(auto simp: shiftr-p-def elim!: effect-elims effect-ifE)
from effect-shiftr-f [OF effect (shiftr-f a i) h’ h’’ u]
have [simp]: ?i h'' = ?i h’ - 1
Array.get h'' a = list-update (Array.get h' a) (?i h') (Array.get h' a ! (?i h' -
by auto
from step have \( *: ?i h' < \text{length} (\text{Array.get} h' a) \)
and \( **: ?i h' - (\text{Suc} 0) \leq ?i h' ?i h' \leq \text{length} (\text{Array.get} h' a) \)
and \( ?i h' \leq j \)
and \( ?i h' < \text{Suc} j \)
and IH: \( \text{ini} h a j = \text{both} h' a j i \)
by (auto simp add: Array.length-def)
have \( \text{Array.length} h a = \text{Array.length} h'' a \) using step by (simp add: Array.length-def)
moreover
have \( ?i h'' \leq j \) using step by auto
moreover
have \( \text{mset} (\text{list-update} (\text{Array.get} h a) j \text{key}) = \)
\( \text{mset} (\text{list-update} (\text{Array.get} h'' a) (?i h'') \text{key}) \)
proof –
have \( ?i h' < \text{length} (\text{Array.get} h' a) \)
and \( ?i h' - 1 < \text{length} (\text{Array.get} h' a) \) using \( * \) by auto
then show \( \text{?thesis} \)
using step by (simp add: mset-update ac-simps nth-list-update)
qed
moreover
have \( \text{ini} h a j = \text{both} h'' a j i \)
using \( \langle 0 < ?i h' \rangle \) and \( \langle ?i h' \leq j \rangle \) and \( \langle ?i h' < \text{length} (\text{Array.get} h' a) \rangle \) and
\( ** \) and IH
by (auto simp: upd-conv-take-nth-drop Suc-diff-le min-absorb1)
(metis Suc-lessD Suc-pred take-Suc-conv-app-nth)
moreover
have \( \text{sorted} (\text{both} h'' a j i) \)
using step and \( \langle 0 < ?i h' \rangle \) and \( \langle ?i h' \leq j \rangle \) and \( \langle ?i h' < \text{length} (\text{Array.get} h' a) \rangle \) and
\( ** \)
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 \( \forall x \in \text{set} (\text{right} h'' a j i). x > \text{key} \)
using step and \( \langle 0 < ?i h' \rangle \) and \( \langle ?i h' < \text{length} (\text{Array.get} h' a) \rangle \) and key
by (auto simp: upd-conv-take-nth-drop Suc-diff-le)
ultimately show \( \text{?case} \) by blast
qed

lemma sorted-take-nth:
assumes \( 0 < i \) and \( i < \text{length} xs \) and \( xs ! (i - 1) \leq y \)
and sorted \( \langle \text{take} i xs \rangle \)
shows \( \forall x \in \text{set} (\text{take} i xs). x \leq y \)
proof –
have \( \text{take} i xs = \text{take} (i - 1) xs \odot [xs ! (i - 1)] \)
using \( \langle 0 < i \rangle \) and \( \langle i < \text{length} xs \rangle \)
by (metis Suc-diff-1 less-imp-diff-less take-Suc-conv-app-nth)
then show \( ?thesis \)
  using \( \text{sorted (take } i \; \text{xs)} \) and \( \text{(xs } ! (i - 1) \leq y) \)
  by (auto simp: sorted-append)
qed

lemma effect-for-insert-elt:
assumes \( l \leq \text{Array.length } h \; a \)
and \( l \leq l \)
and effect (for \([ 1 ..< l ] \) (insert-elt a)) \( h \; h' \; u \)
shows \( \text{Array.length } h \; a = \text{Array.length } h' \; a \wedge \)
  sorted (take \( l \) (Array.get \( h' \) a)) \wedge
  mset (Array.get \( h \) a) = mset (Array.get \( h' \) a)
using assms(2–)
proof (induction \( l \; h' \) rule: effect-for-induct)
case base
  show \( ?case \) by (cases Array.get \( h \) a) simp-all
next
case \( \text{(step } j \; h' \; h'' \; u) \)
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(3) [unfolded insert-elt-def]
  obtain key and \( h_1 \) and \( i \) and \( h_2 \) and \( i' \)
  where key: \( \text{key } = \text{Array.get } h' \; a \; ! \; j \)
  and effect (ref \( j \) \( h' \; h_1 \) \( i \)
  and ref1: Ref.get \( h_1 \) \( i = j \)
  and shiftr': effect (shiftr a key \( i \)) \( h_1 \; h_2 \) \()
  and [simp]: Ref.get \( h_2 \) \( i = i' \)
  and [simp]: \( h'' = \text{Array.update } a \; i' \; \text{key } h_2 \)
  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 \( j \) \( h' \; h_1 \) \( i \)) have [simp]: 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 ref1 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 \( i' < \text{Array.length } h_2 \; a \) length-def)
  have [simp]: min (Suc \( j \)) \( i' = i' \) using \( i' \leq j \) by simp
  have [simp]: \( \text{min (length } (\text{Array.get } h_2 \; a)) \; i' = i' \)
    using \( i' < \text{length } (\text{Array.get } h_2 \; a) \) by (simp)
  have take-Suc-j: (Suc \( j \)) (\text{list-update } (\text{Array.get } h_2 \; a) \; i' \; \text{key}) =
    take \( i' \) (\text{Array.get } h_2 \; a) @ key # take (\( j - i' \)) (\text{drop } (\text{Suc } i') (\text{Array.get } h_2 \; a))

10
unfolding upd-cone-take-nth-drop \([\text{OF } i' < \text{length } (\text{Array.get } h_2 a)]\)
by (auto) (metis Suc-diff-le \(i' \leq j\) take-Suc-Cons)

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

lemma effect-shiftr-index:
  assumes effect (shiftr a key i) h h' a
  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' a
  shows Array.length h' a = Array.length h a
  using assms unfolding shiftr-def
by (induct h' rule: ro-shiftr-p.effect-while-induct) (auto simp: length-def dest: effect-shiftr-f)

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) h 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) h 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 \( \text{key} \) and \( h_1 \) and \( i \) and \( h_2 \) and \( i' \)
where \( \text{key: key} = \text{Array.get} \ h' \ a \ ! \ j \)
  and effect (ref \( j \)) \( h' \ h_1 \ i \)
  and ref\(_1\): Ref.get \( h_1 \ i = j \)
  and shiftr': effect (shiftr \( a \) key \( i \)) \( h_1 \ h_2 () \)
  and [simp]: Ref.get \( h_2 \ i = i' \)
  and [simp]: \( h'' = \text{Array.update} \ a \ i' \text{ key} \ h_2 \)
  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 \( j \)) \( h' \ h_1 \)) have [simp]: Array.get \( h_1 \ a = \text{Array.get} \ h' \ a \)
  by (metis array-get-alloc effectE execute-ref option.sel)

have [simp]: 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 ref1 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 \( i' < \text{Array.length} \ h_2 \ a \) vs length-def)

have [simp]: min (Suc \( j \)) \( i' = i' \) using \( i' \leq j \) by simp

have [simp]: min (length (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 (Suc \( i' \)) (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 \( i' \leq j \) 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.effec-while-post [OF shiftr' [unfolded 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]: \( i' = 0 \)

  14
have \( \ast \): \( \text{take} (\text{Suc} j) \) (\( \text{list-update} (\text{Array.get} h_2 a) \) 0 key) =
\( \text{key} \# \) \( \text{take} j \) (\( \text{drop} 1 \) (\( \text{Array.get} h_2 a) \))
by (simp) (metis \( i' = 0 \) append-Nil takeSuc-j diff-zero take0)
from sorted and shiftr
  have sorted (\( \text{take} j \) (\( \text{drop} 1 \) (\( \text{Array.get} h_2 a) \)))
  and \( \forall x \in \text{set} \) (\( \text{take} j \) (\( \text{drop} 1 \) (\( \text{Array.get} h_2 a) \))). key < x by simp-all
then have sorted (\( \text{key} \# \) \( \text{take} j \) (\( \text{drop} 1 \) (\( \text{Array.get} h_2 a) \)))
by (metis less-imp-le sorted simps (2))
then show \( \ast \)thesis by (simp add: \( \ast \))
next
assume 0 < \( i' \) \& key \geq \( \text{Array.get} h_2 a \) ! (\( i' - 1 \))
moreover
have \( \text{sorted} \) (\( \text{take} i' \) (\( \text{Array.get} h_2 a) \) \@ (\( \text{take} j - i' \) (\( \text{drop} (\text{Suc} i') \) (\( \text{Array.get} h_2 a) \))))
  and \( \forall x \in \text{set} \) (\( \text{take} j - i' \) (\( \text{drop} (\text{Suc} i') \) (\( \text{Array.get} h_2 a) \))). key < x
using shiftr by auto
ultimately have \( \forall x \in \text{set} \) (\( \text{take} i' \) (\( \text{Array.get} h_2 a) \)). x \leq key
using sorted-take-nth [OF - \( i' < \text{length} (\text{Array.get} h_2 a) \)], of key
by (simp add: sorted-append)
then show \( \ast \)thesis
  using shiftr by (auto simp: takeSuc-j sorted-append less-imp-le)
qed
qed
ultimately
show \( \ast \)case by blast
qed

lemma \textbf{insertion-sort-correct}:
\( \exists h'. \text{effect} (\text{insertion-sort} a) h h' u \land \)
\( \text{mset} (\text{Array.get} h a) = \text{mset} (\text{Array.get} h' a) \land \)
\( \text{sorted} (\text{Array.get} h' a) \)
proof (cases \( \text{Array.length} h a = 0 \))
assume \( \text{Array.length} h a = 0 \)
then have \( \text{effect} (\text{insertion-sort} a) h h () \)
  and \( \text{mset} (\text{Array.get} h a) = \text{mset} (\text{Array.get} h a) \)
  and \( \text{sorted} (\text{Array.get} h a) \)
by (auto simp: insertion-sort-def length-def intro!: effect-intros)
then show \( \ast \)thesis by auto
next
assume \( \text{Array.length} h a \neq 0 \)
then have 1 \( \leq \) \( \text{Array.length} h a \) by auto
from for-insert-ell-correct [OF le-refl this]
  show \( \ast \)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 \textbf{insertion-sort} in Haskell

15
end