Imperative Insertion Sort

Christian Sternagel

September 19, 2015

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 ~~/src/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 heap ⇒ bool together with tap. However, this does not allow for code
generation due to breaking the heap-abstraction.

partial-function (heap) while :: bool Heap ⇒ 'b Heap ⇒ unit Heap
where
  [code]: while p f = do {
    b ← p;
    if b then f ≫ while p f
    else return ()
  }

definition cond p h ←→ 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 ⇒ snd (the (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 ⇒ cond p h ⇒ execute p h = Some (True, h)
success p h ⇒ ¬ cond p h ⇒ 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 ⇒ cond p h ⇒ execute (while p f) h = execute (f ≫ while p f) h
success p h ⇒ ¬ cond p h ⇒ execute (while p f) h = execute (return ()) h
by (auto simp: while.simps execute-simps)

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

lemma success-cond-effect:
success p h ⇒ cond p h ⇒ 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 ⇒ ¬ cond p h ⇒ 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' ∧ ¬ 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 \( \forall h, h'. I h \Rightarrow success p h \Rightarrow cond p h \Rightarrow effect f h h' u \Rightarrow I h' \)
shows \( I h' \)
using assms(1, 3–)

proof (induction \( p f h h' u \) rule: while.raw-induct)
case \((1 w p f h h' u)\)
  obtain \( b \) where \( effect p h h b \)
  and \( \ast \): \( effect (if b then f \Rightarrow w p f else return ()) h h' u \)
  using 1.hyps and \( ro-cond p \)
  by (auto elim!: effect-elims intro: effect-intros) (metis effectE ro-cond.read-only)

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

  show ?case
  proof (cases \( b \))
    assume \( \neg b \)
    then show ?thesis using \( \ast \) and \( I h' \) by (auto elim: effect-elims)
  next
    assume \( b \)
    moreover
    with \( \ast \) obtain \( h'' \) and \( r \)
      where \( effect f h h'' r \) and \( effect (w p f) h'' h' u \) \textbf{by} (auto elim: effect-elims)
    moreover
    ultimately
    show ?thesis using 1 and cond by blast
  qed

qed

fact

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

custom ro-cond
begin

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

A rule for proving total correctness of while loops.

lemma \( wf-while-induct \ [consumes 1, case-names success-cond success-body base step]: \)
\textbf{assumes} \( wf R \) — a well-founded relation on heaps proving termination of the loop
  \textbf{and} \( success-p : \forall h. I h \Rightarrow success p h \) — the loop-condition terminates
  \textbf{and} \( success-f : \forall h. I h \Rightarrow success p h \Rightarrow cond p h \Rightarrow success f h \) — the loop-body terminates
  \textbf{and} \( I h \) — the invariant holds before the loop is entered
  \textbf{and} \( step : \forall h h' r. I h \Rightarrow success p h \Rightarrow cond p h \Rightarrow effect f h h' r \Rightarrow \)
(h', h) ∈ R ∧ I h'
— the invariant is preserved by iterating the loop
shows ∃ h'. effect (while p f) h h' () ∧ I h'
using (wf R) and ⟨ I h ⟩
proof (induction h)
case (less h)
show ?case
proof (cases cond p h)
assume ¬ cond p h then show ?thesis
using ⟨ I h ⟩ and success-p [of h] by (simp add: effect-def execute-simps)
next
assume cond p h
with ⟨ I h ⟩ and success-f [of h] and step [of h] and success-p [of h]
obtain h' and r where effect f h h' r and (h', h) ∈ R and I h' and success p h
by (auto simp: success-def effect-def)
with less.IH [of h'] show ?thesis
using (cond p h) 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]
= wf-while-induct [unfolded effect-success-conv, THEN conjunct1] end

1.2 For Loops

fun for :: 'a list ⇒ ('a ⇒ 'b Heap) ⇒ unit Heap
where
for [] f = return () |
for (x # xs) f = f x ≫ for xs f

A rule for proving partial correctness of for loops.

lemma effect-for-induct [consumes 2, case-names base step]:
assumes i ≤ j
and effect (for [i..< j] f) h h' u
and I i h
and ∃ k h h' r. i ≤ k ⇒ k < j ⇒ I k h ⇒ effect (f k) h h' r ⇒ I (Suc k) h'
shows I j h'
using 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 (effect (for [i ..< j] f) h h' w) 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

A rule for proving total correctness of for loops.

lemma for-induct [consumes 1, case-names succeed base step]:
assumes i ≤ j
and \( \Land \ k. \ I \ k \ h \quad \Rightarrow \quad i \leq k \Rightarrow k < j \Rightarrow \text{success} (f \ k) \ h \)
and I i h
and \( \Land \ k \ h \ h'. \ I \ k \ h \ h' \quad \Rightarrow \quad i \leq k \Rightarrow k < j \Rightarrow \text{effect} (f \ k) \ h \ h' \Rightarrow I (Suc k) \ h' \)
shows \( \exists h'. \ \text{effect} (for [i ..< j] f) \ h \ h' \quad \land \quad I j \ h' \quad (\text{is} \ ?P \ i \ h) \)
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 \( \forall P \ (\text{Suc } i) \ h' \)
by (rule Suc(1) [OF \( \{ k = j - \text{Suc } i \wedge \text{Suc } i \leq j \} \ \text{Suc}(4) \ \{ (\text{Suc } i) \ h' \ \text{Suc}(6) \} ]
auto
ultimately
show \( \forall \text{case by } (\text{auto simp: effect-def execute-simps}) \)
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
~/src/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 ≡ Arrayupd i x a

abbreviation array-nth :: 'a::heap array ⇒ nat ⇒ 'a Heap (-.'-') [1000, 0] 14
where
a.(i) ≡ Arraynth 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 {
\[
i' \leftarrow ! i; \\
\text{if } i' > 0 \text{ then do } \{ x \leftarrow a.(i' - 1); \text{return } (x > \text{key}) \} \\
\text{else return False} \}
\]

(\text{do } \{ \\
i' \leftarrow ! i; \\
x \leftarrow a.(i' - 1); \\
a.(i') \leftarrow x; \\
i := i' - 1 \\
\})); \\
i' \leftarrow ! i; \\
a.(i') \leftarrow \text{key}
\}
\}

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

definition shiftr-p a (key::'a::{heap, linorder}) i = 
(\text{do } \{ i' \leftarrow ! i; \\
\text{if } i' > 0 \text{ then do } \{ x \leftarrow a.(i' - 1); \text{return } (x > \text{key}) \} \text{ else return False} \})

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

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

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

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

lemma insertion-sort-alt-def: 
\text{insertion-sort a} = (\text{Array.len a >= sort-upto a}) 
\text{by } (\text{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: 
\text{assumes } \text{effect } (\text{shiftr-f a i}) h h' u 
\text{shows } \text{Ref.get h'} i = \text{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 \implies 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 j i = take (j - Ref.get h i + 1) (drop (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 h' u
  shows Array.length h a = Array.length h' a \land
  \forall i h' \leq j \land
  mset (list-update (Array.get h a) j key) =
  mset (list-update (Array.get h' a) (?i h') key) \land
  ini h a j = both h' a j i \land
  sorted (both h' a j i) \land
  \forall x \in 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' \land
  (cond (shiftr-p a key i)) 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' - 1 - 8
1))

by auto

from step have \(*\): \(?i h' \leq \text{length } (\text{Array}.\text{get } h' a)\)
and \(\ast\ast: \(?i h' - (\text{Suc } 0) \leq \text{Suc } 0 \leq \text{length } (\text{Array}.\text{get } h' a)\)
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 \(\text{Array}.\text{length } h a = \text{Array}.\text{length } h'' a\) using step by (simp add: Array.length-def)

moreover
have \(?i h'' \leq j\) using step by auto

moreover
have \(\text{mset } (\text{list-update } (\text{Array}.\text{get } h a) j \text{ key}) = \text{mset } (\text{list-update } (\text{Array}.\text{get } h'' a) (?i h'') \text{ key})\)

proof –

have \(?i h' < \text{length } (\text{Array}.\text{get } h' a)\)
and \(?i h' - 1 < \text{length } (\text{Array}.\text{get } h' a)\) 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 \(0 < ?i h'\) and \(?i h' \leq j\) and \(?i h' < \text{length } (\text{Array}.\text{get } h' a)\) and \(\ast\ast\) 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 \(\forall x \in \text{set } (\text{right } h'' a j i), x > \text{key}\)

using step and \(0 < ?i h'\) and \(?i h' \leq j\) and \(?i h' < \text{length } (\text{Array}.\text{get } h' a)\) and \(\ast\ast\) 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 \(\forall x \in \text{set } (\text{right } h'' a j i), x > \text{key}\)

using step and \(0 < ?i h'\) and \(?i h' < \text{length } (\text{Array}.\text{get } h' a)\) and \(\text{key}\)

by (auto simp: upd-conv-take-nth-drop Suc-diff-le)

ultimately show \(?\text{case}\) by blast

qed

lemma \text{sorted-take-nth}:

assumes \(0 < i\) and \(i < \text{length } xs\) and \(xs ! (i - 1) \leq y\)

and \(\text{sorted } (\text{take } i xs)\)

shows \(\forall x \in \text{set } (\text{take } i xs), x \leq y\)

proof –

have \(\text{take } i xs = \text{take } (i - 1) xs @ [xs ! (i - 1)]\)

using \(0 < i\) and \(i < \text{length } xs\)

by (metis Suc-diff-1 less-imp-diff-less take-Suc-conv-app-nth)
then show ?thesis
  using (sorted (take i xs)) and (xs ! (i - 1) ≤ y)
  by (auto simp: sorted-append)
qed

lemma effect-for-insert-elt:
  assumes l ≤ Array.length h a
  and I ≤ l
  and effect (for [I..< l] (insert-elt a)) h h' u
  shows 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 l h' rule: effect-for-induct)
case base
  show ?case by (cases Array.get h a) simp-all
next
case (step j h' h'' u)
  with assms(1) have j < 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 h1 and i and h2 and i'
    where key: key = Array.get h' a ! j
    and effect (ref j) h' h1 i
    and ref1: Ref.get h1 i = j
    and shiftr': effect (shiftr a key i) h1 h2 ()
    and [simp]: Ref.get h2 i = i'
    and [simp]: h'' = Array.update a i' key h2
    and i' < Array.length h2 a
    by (elim effect-bindE effect-nthE effect-lookupE effect-updE)
      (auto intro: effect-intros, metis effect-refE)
  from (effect (ref j) h' h1 i) have [simp]: Array.get h1 a = Array.get h' a
    by (metis array-get-alloc effectE execute-ref option.sel)
  have [simp]: Array.length h1 a = Array.length h' a by (simp add: Array.length-def)
  from step and assms(1)
  have j < Array.length h1 a sorted (take j (Array.get h1 a)) by auto
  note shiftr = effect-shiftr [OF ref1 this shiftr' [unfolded shiftr-def], simplified]
  have i' ≤ j using shiftr by simp

  have i' < length (Array.get h2 a)
    by (metis (open) i' < Array.length h2 a length-def)
  have [simp]: min (Suc j) i' = i' using (i' ≤ j) by simp
  have [simp]: min (length (Array.get h2 a)) i' = i'
    using (i' < length (Array.get h2 a)) by (simp)
  have take-Suc-j: take (Suc j) (list-update (Array.get h2 a) i' key) =
    take i' (Array.get h2 a) @ key # take (j - i') (drop (Suc i') (Array.get h2 a))
lemma effect-insertion-sort:
assumes effect (insertion-sort a) 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 (Λ h. Ref.get h i)) by (metis wf-measure)
then show ?thesis
proof (induct taking: Λ h. 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' 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) (metis Ref.get-alloc array-get-alloc effect-heapI)
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 (l 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'' a)
with assms(1) have j < Array.length h' a by auto
from step have sorted: sorted (take j (Array.get h' a)) by blast
from step(f4) [unfolded insert-elt-def]
  obtain key and h1 and i and h2 and i'
  where key: key = Array.get h' a ! j
      and effect (ref j) h' h1 i
      and ref1: Ref.get h1 i = j
      and shiftr': effect (shiftr a key i) h1 h2 ()
      and [simp]: Ref.get h2 i = i'
      and [simp]: h'' = Array.update a i' key h2
      and i' < Array.length h2 a
  by (elim bindE nthE lookupE updE)
(auto intro: intros, metis refE)
from (effect (ref j) h' h1 i) have [simp]: Array.get h1 a = Array.get h' a
  by (metis array-get-alloc effectE execute-ref option.sel)
have [simp]: Array.length h1 a = Array.length h' a by (simp add: length-def)
from step and assms(1)
  have j < Array.length h1 a sorted (take j (Array.get h1 a)) by auto
  note shiftr = effect-shiftr [OF ref1 this shiftr' [unfolded shiftr-def], simplified]
have i' ≤ j using shiftr by simp
have i' < length (Array.get h2 a)
  by (metis i' < Array.length h2 a length-def)
have [simp]: min (Suc j) i' = i' using i' ≤ j by simp
have [simp]: min (length (Array.get h2 a)) i' = i'
  using i' < length (Array.get h2 a) by (simp)
have take-Suc-j: take (Suc j) (list-update (Array.get h2 a) i' key) =
    take i' (Array.get h2 a) @ key # take (j - i') (drop (Suc i') (Array.get h2 a))
  unfolding upd-one-take-nth-drop [OF i' < length (Array.get h2 a)]
  by (auto) (metis Suc-diff-le i' ≤ j) take-Suc-Cons
have Array.length h a = 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' [unfolded shiftr-def]]
    have i' = 0 ∨ (0 < i' ∧ key ≥ Array.get h2 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 \((\text{Suc } j)\) \((\text{list-update} \ (\text{Array.get} \ h_2 \ a) \ 0 \ \text{key}) =\)
key \# take \(j\) \((\text{drop } 1 \ (\text{Array.get} \ h_2 \ a))\)
by \((\text{simp})\) \((\text{metis} \ \langle i' = 0 \rangle \ \text{append-Nil} \ \text{take-Suc-j} \ \text{diff-zero} \ \text{take-0})\)
from \text{sorted} \ \text{and} \ \text{shiftr}

have \text{sorted} \ (\text{take } j \ (\text{drop } 1 \ (\text{Array.get} \ h_2 \ a)))
and \(\forall x \in \text{set} \ (\text{take } j \ (\text{drop } 1 \ (\text{Array.get} \ h_2 \ a))). \text{key} < x\) \text{by simp-all}
then have \text{sorted} \ (\text{key} \# \text{take } j \ (\text{drop } 1 \ (\text{Array.get} \ h_2 \ a)))
by \((\text{metis} \ \text{less-imp-le} \ \text{sorted-Cons})\)
then show \(\text{?thesis}\) by \((\text{simp add: *})\)

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 \(\forall x \in \text{set} \ (\text{take } i' \ (\text{Array.get} \ h_2 \ a)). \text{key} < x\) \text{by shiftr}
ultimately have \(\forall x \in \text{set} \ (\text{take } i' \ (\text{Array.get} \ h_2 \ a)). x \leq \text{key}\)
using \text{sorted-take-nth} \ [\text{OF} - \langle i' < \text{length} \ (\text{Array.get} \ h_2 \ a)\rangle, \ \text{of} \ \text{key}]\)
by \((\text{simp add: sorted-append})\)
then show \(\text{?thesis}\)
using \text{shiftr} \ \text{by} \ (\text{auto simp: take-Suc-j} \ \text{sorted-append}) \ (\text{metis} \ \text{less-imp-le} \ \text{sorted.} \ \text{Cons})\)
qed

lemma \text{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 \((\text{cases Array.length} \ h \ a = 0)\)
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 \((\text{auto simp: insertion-sort-def} \ \text{length-def intro!: effect-intros})\)
then show \(\text{?thesis}\) \text{by} \ (\text{auto})

next
assume \text{Array.length} \ h \ a \neq 0
then have \(1 \leq \text{Array.length} \ h \ a\) \text{by} \ (\text{auto})
from \text{for-insert-ell-correct} \ [\text{OF} \ \text{le-refl this}]\)
show \(\text{?thesis}\)
by \((\text{auto simp: insertion-sort-alt-def} \ \text{sort upto-def})\)
(metis \text{One-nat-def} \text{effect bind I} \text{effect insertion sort} \text{effect length I} \text{insertion sort alt-def} \text{sort upto-def})
qed
export-code insertion-sort in Haskell

end