Impressive Insertion Sort

Christian Sternagel

May 28, 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

1
assumes read-only: $\text{success } p \ h \Rightarrow \text{snd\ (the\ (\text{execute\ } p \ h))} = h$

begin

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

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

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

lemma success-while-cond: $\text{success } p \ h \Rightarrow \text{cond } p \ h \Rightarrow \text{effect } h \ h' \ r \Rightarrow \text{success\ (while } p \ f) \ h' \Rightarrow$
  $\text{success\ (while } p \ f) \ h$ and

lemma success-while-not-cond: $\text{success } p \ h \Rightarrow \neg \text{cond } p \ h \Rightarrow \text{success\ (while } p \ f) \ h$
  by (auto simp: while-simps effect-def execute-simps intro!: success-intros)

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

lemma success-not-cond-effect:
  $\text{success } p \ h \Rightarrow \neg \text{cond } p \ h \Rightarrow \text{effect } p \ h\ False$
  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 $p\ h'\ 
\ 
\neg\ \text{cond } p\ h'$
  using assms(1)
  apply (induct rule: while.raw-induct [OF - assms(2)])
  apply (auto elim!: effect-elims effect-ifE simp: 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 $I\ h$

2
and $\land h \land h' u$. $I h \implies success p h \implies cond p h \implies effect f h' u \implies I h'$
shows $I h'$
using $\text{assms}(1, 3)$

proof (induction $p f h' u$ rule: while.raw-induct)
case ($1 w p f h' u$)

obtain $b$
where $effect p h b$
and $\ast$: $effect (if b then f \Rightarrow w \ast p \ast else return ()) h' u$
using 1.hyps and (ro-cond $p$)
by (auto elim!: effect-elims intro: effect-intros) (metis effectE ro-cond.read-only)
then have cond: $success p h \land cond p h = b$ by (auto simp: cond-def elim!: effect-elims effectE)

show $\ast$case
proof (cases $b$)
assume $\neg b$
then show $\ast$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 $effect (w p f) h'' h' u$ by (auto elim: effect-elims)
moreover
ultimately
show $\ast$thesis using 1 and cond by blast

qed

qed fact

lemma effect-success-conv:
$(\exists h'. effect c h h' (I h') \land I h') \iff success c h \land I (\snd (the \ (execute c h)))$

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

context 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]:
assumes $wf R$ — a well-founded relation on heaps proving termination of the loop
and success-p: $\land h. I h \Rightarrow success p h$ — the loop-condition terminates
and success-f: $\land h. I h \Rightarrow success p h \Rightarrow cond p h \Rightarrow success f h$ — the loop-body terminates
and $I h$ — the invariant holds before the loop is entered
and step: $\land h h' r. I h \Rightarrow success p h \Rightarrow cond p h \Rightarrow effect f h' r \Rightarrow$
\((h', h) \in R \land I h'\)
— the invariant is preserved by iterating the loop

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

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

\textbf{proof} (induction \(h\))

\textbf{case} (less \(h\))

\textbf{show} \(?\text{case}\)

\textbf{proof} (cases cond \(p \ h\))

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

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

\textbf{next}

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

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

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

\(p \ h\)

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

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

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

\textbf{qed}

\textbf{qed}

A rule for proving termination of while loops.

\textbf{lemmas}

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

\(=\)

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

\textbf{end}

1.2 For Loops

\textbf{fun} \textit{for} :: \('a list \Rightarrow ('a ⇒ 'b Heap) ⇒ unit Heap\)
\textbf{where}
\(\text{for } [] \ f = \text{return } () |\)
\(\text{for } (x \ # \ xs) \ f = f \ x \# \text{ for } xs \ f\)

A rule for proving partial correctness of for loops.

\textbf{lemma} \textit{effect-for-induct} [\textit{consumes} 2, \textit{case-names} base step]:
\textbf{assumes} \(i \leq j\)
\(\text{and} \text{effect (for } [i ..< j] \ f \ h \ h' u\)
\(\text{and} I i h\)
\(\text{and} \ l k h \ h' r. i \leq k \Longrightarrow k < j \Longrightarrow \text{success-p [of } h'\) \text{ and success}

\(j \ h'\)

\textbf{shows} \(I j h'\)

\textbf{using} \textit{assms}

\textbf{proof} (induction \(j - i \text{ arbitrary; } i h)\)

\textbf{case} 0

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

\textbf{next}
case \((\text{Suc} \, k)\)

show ?case

proof (cases \(j = i\))

  case True

  with Suc show ?thesis by auto

next

  case False

  with \((i \leq j) \land (\text{Suc} \, k = j - i)\)

  have \(i < j\) and \(k = j - \text{Suc} \, i \land \text{Suc} \, i \leq j\) by auto

  then have \([i ..< j] = i \# [\text{Suc} \, i ..< j]\) by (metis upt-rec)

  with effect (for \([i ..< j] \, f\) \(h \, h'\)) obtain \(h'' \, r\)

  where \(*: \text{effect} \, (f \, i) \, h \, h'' \, r \land **: \text{effect} \, (\text{Suc} \, i \, ..< \, j) \, h'' \, h' \, u\)

  by (auto elim: effect-elims)

  from Suc(6) \([OF - (i \, h) \, \star] \land \langle \langle i < j \rangle\rangle\)

  have \(I \langle \text{Suc} \, i \rangle \, h''\) by auto

  show ?thesis

  by (rule Suc(1) \([OF \, \langle \text{Suc} \, i \rangle \, h \, h'' \, \langle \text{Suc} \, (6)\rangle\])

auto

qed

A rule for proving total correctness of for loops.

lemma for-induct [consumes 1, case-names succeed base step]:

assumes \(i \leq j\)

and \(\langle k \, h, I \, k \, h \Rightarrow i \leq k \Rightarrow k < j \Rightarrow \text{success} \, (f \, k) \, h\rangle\)

and \(I \, i \, h\)

and \(\langle h \, h', I \, k \, h \Rightarrow i \leq k \Rightarrow k < j \Rightarrow \text{effect} \, (f \, k) \, h \, h' \, r \Rightarrow I \, (\text{Suc} \, k) \, h'\rangle\)

shows \(\exists h', \text{effect} \, (\text{for} \, [i \, ..< \, j] \, f) \, h \, h'(\) \land \(I \, j \, h'(\) \,(\text{is} \, \langle P \, i \, h\rangle)\)

using assms

proof (induction \(j - i \, \text{arbitrary}: i \, h\))

  case 0

  then show ?case by (auto simp: effect-def execute-simps)

next

  case \((\text{Suc} \, k)\)

  show ?case

  proof (cases \(j = i\))

    assume \(j = i\)

    with Suc show ?thesis by auto

next

  assume \(j \neq i\)

  with \((i \leq j) \land (\text{Suc} \, k = j - i)\)

  have \(i < j\) and \(k = j - \text{Suc} \, i \land \text{Suc} \, i \leq j\) by auto

  then have \([simp]: \langle i \, ..< \, j \rangle = i \# [\text{Suc} \, i \, ..< \, j]\) by (metis upt-rec)

  obtain \(h' \, r\) where \(*: \text{effect} \, (f \, i) \, h \, h' \, r\)

  using Suc(4) \([OF \, (i \, h) \, \text{le-refl} \, (i < j)]\) by (auto elim!: success-effectE)

  moreover

  then have \(I \, (\text{Suc} \, i) \, h'\) using Suc by auto

  moreover

  qed
have \( ?P (\text{Suc } i) \ h' \)
by (rule Suc(1) \[ OF \ k = j - \text{Suc } i \ oSuc i \leq j \ oSuc(4) \ oI (\text{Suc } i) \ h' \ Suc(6)]]
auto
ultimately
show \( ?\text{case by (auto simp: effect-def execute-simps)} \)
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
   ~/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 ≡ 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\}) \( i \) =

\[
\begin{align*}
&\text{(do \{} \\
&\quad i' \leftarrow ! i; \\
&\quad x \leftarrow a.(i' - 1); \\
&\quad a.(i') \leftarrow x; \\
&\quad i := i' - 1 \\
&\}) \\
&\quad i' \leftarrow ! i; \\
&\quad a.(i') \leftarrow key \\
&\}
\]

**Definition** shiftr-f \( a \ i \) = do \{

\[
\begin{align*}
&\quad i' \leftarrow ! i; \\
&\quad x \leftarrow a.(i' - 1); \\
&\quad a.(i') \leftarrow x; \\
&\quad i := i' - 1
\]
\}

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

**Definition** insert-elt \( a \) = \( \lambda j. \) do \{

\[
\begin{align*}
&\quad key \leftarrow a.(j); \\
&\quad i \leftarrow \text{ref} \ j; \\
&\quad shiftr a key i; \\
&\quad i' \leftarrow ! i; \\
&\quad a.(i') \leftarrow key
\]
\}

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

**Lemma** insertion-sort-alt-def:

\[
\begin{align*}
&\text{insertion-sort} \ a = (\text{Array.len} \ a \geq \text{sort-uppto} \ a) \\
&\text{by} \ (\text{simp add:} \ \text{insertion-sort-def} \ \text{sort-uppto-def} \ \text{shiftr-def} \ \text{shiftr-p-def} \ \text{shiftr-f-def} \ \text{insert-elt-def})
\]

### 2.2 Partial Correctness

**Lemma** effect-shiftr-f:

\[
\begin{align*}
&\text{assumes} \ \text{effect} \ (\text{shiftr-f} \ a \ i) \ h \ h' \ u \\
&\text{shows} \ \text{Ref}.\text{get} \ h' \ i = \text{Ref}.\text{get} \ h \ i - 1 \land
\]

7
\begin{align*}
\text{Array.get } \text{h'} \text{ 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 by} \text{ (auto simp: shiftr-f-def elim!: effect-elims)}
\end{align*}

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

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

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

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

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

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

\textbf{lemma effect-shiftr:}
\begin{align*}
\text{assumes Ref.get h i} = j \text{ (is } ?i h = \_)
\text{ and } j < \text{Array.length h a}
\text{ and sorted (take j (Array.get h a))}
\text{ and effect (while (shiftr-p a key i) (shiftr-f a i)) h' u}
\text{shows Array.length h a = Array.length h' a} \\
\text{?i h' \leq j} \\
\text{multiset-of (list-update (Array.get h a) j key) =}
\text{multiset-of (list-update (Array.get h' a) (?i h') key)} \\
\text{ini h a j = both h' a j i } \\
\text{sorted (both h' a j i) } \\
(\forall x \in \text{set (right h' a j i)}. x > \text{key})
\text{using assms(4, 2)}
\text{proof (induction rule: ro-shiftr-p.effect-while-induct)}
\text{case base}
\text{show ?case using assms by auto}
\text{next}
\text{case (step h' h'' u)}
\text{from (success (shiftr-p a key i) h') and (cond (shiftr-p a key i) h')}
\text{have ?i h' > 0 and}
\text{key: Array.get h' a ! (?i h' - 1) > key}
\text{by (auto dest!: ro-shiftr-p.success-cond-effect)}
\text{(auto simp: shiftr-p-def elim!: effect-elims effect-ifE)}
\text{from effect-shiftr-f [OF (effect (shiftr-f a i) h' h'' u)]}
\text{have [simp]: ?i h'' = ?i h' - 1}
\text{Array.get h'' a = list-update (Array.get h' a) (?i h') (Array.get h' a) (! ?i h' - 1)}}
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 multiset-of (list-update (Array.get h a) j key) =
multiset-of (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: multiset-of-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
∗∗ and IH
by (auto simp: upd-conv-take-nth-drop Suc-diff-le min-absorb1)
(metis Suc-lessD Suc-pred append.take-Suc-cone-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
∗∗ 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-cone-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
qed

lemma sorted-take-nth:
assumes 0 < i and i < length xs and xs ! (i - 1) ≤ y
and sorted (take i xs)
sows ∀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-cone-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 [1..< l] (insert-elt a)) h h' u
  shows Array.length h a = Array.length h' a ∧
  sorted (take l (Array.get h' a)) ∧
  multiset-of (Array.get h a) = multiset-of (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 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-cone-take-nth-drop \[\text{OF } i' < \text{length } \text{Array.get } h_2 a\]:
by (auto) (metis Suc-diff-le \(i' < j\) take-Suc-Cons)

have \(\text{Array.length } h \ a = \text{Array.length } h'' \ a\)
using shiftr by (auto) (metis step.IH)
moreover
have \(\text{multiset-of } (\text{Array.get } h \ a) = \text{multiset-of } (\text{Array.get } h'' \ a)\)
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 \[\text{OF } \text{shiftr}' [\text{unfolded shiftr-def}]\]
  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 add: shiftr-p-def)
  then show \(?\text{thesis}\)
proof
  assume [simp]: \(i' = 0\)
  have \(*\): \(\text{take } (\text{Suc } j) (\text{list-update } (\text{Array.get } h_2 a) \ 0 \ \text{key}) =\)
    \(\text{key} \# \text{take } j \ (\text{drop } 1 \ (\text{Array.get } h_2 a))\)
  by (simp) (metis \(i' = 0\) append-Nil take-Suc-j diff-zero take-0)
  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-Cons)
  then show \(?\text{thesis}\) 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 \[\text{OF } - i' < \text{length } (\text{Array.get } h_2 a)\], of key]
  by (simp add: sorted-append)
  then show \(?\text{thesis}\)
    using shiftr by (auto simp: take-Suc-j sorted-append) (metis less-imp-le sorted.Cons)
qed

lemma effect-insertion-sort:
assumes \(\text{effect } (\text{insertion-sort } a) \ h \ h' \ u\)
shows \(\text{multiset-of } (\text{Array.get } h \ a) = \text{multiset-of } (\text{Array.get } h' \ a) \land \text{sorted}\)
(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
qed

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.success-while-induct) (auto dest: effect-shiftr-f)

lemma effect-shiftr-length:
  assumes effect (shiftr a key i) h h' u
shows $\text{Array}.\length h' a = \text{Array}.\length h a$

using assms unfolding shiftr-def
  by (induct h' rule: ro-shiftr-p.eff-effect-while-induct) (auto simp: length-def dest: effect-shiftr-f)

lemma success-insert-elt:
  assumes $k < \text{Array}.\length h a$
  shows success ($\text{insert-elt} a k$) h

proof –
  obtain key where effect (a.(k)) h h key
    using assms by (auto intro: effect-intros)
  moreover
  obtain $i$ and $h_1$ where effect (ref k) h $h_1$ $i$
    and [simp]: Ref.get $h_1$ $i$ = $k$
    and [simp]: Array.length $h_1$ a = Array.length h a
    by (auto simp: ref-def length-def) (metis Ref.get-alloc array-get-alloc effect-heapI)
  moreover
  obtain $h_2$ where \(*\): effect (shiftr $a$ key $i$) $h_1$ $h_2$ ()
    using success-shiftr [of $h_1$ $i$ $a$ key] and assms
    by (auto simp: effect-def execute-simps)
  moreover
  have effect (!$i$) $h_2$ $h_2$ (Ref.get $h_2$ $i$)
    and Ref.get $h_2$ $i$ $\leq$ Ref.get $h_1$ $i$
    and Ref.get $h_2$ $i$ $\leq$ Array.length $h_2$ a
    using effect-shiftr-index !$*\$ and effect-shiftr-length !$*\$ and assms
    by (auto intro!: effect-intros)
  moreover
  then obtain $h_3$ and $r$ where effect (a.(Ref.get $h_2$ $i$) $\leftarrow$ key) $h_2$ $h_3$ $r$
    using assms by (auto simp: multiset-of)

ultimately
  have effect (insert-elt $a$ $k$) $h_3$ $r$
    by (auto simp: insert-elt-def intro: effect-intros)
  then show ?thesis by (metis effectE)

qed

lemma for-insert-elt-correct:
  assumes $l \leq \text{Array}.\length h a$
    and $1 \leq l$
  shows $\exists h'. \\text{effect} \ (\text{for} \ [1 ..< l] \ \text{(insert-elt} a\)) \ h \ h' () \land$
    Array.length $h$ $a$ = Array.length $h'$ $a$ \land
    sorted (take l (Array.get $h'$ $a$)) \land
    multiset-of (Array.get $h$ $a$) = multiset-of (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

qed
next 

\textbf{case (step \(j \ h' \ h'' \ a\))}

with \texttt{assms(1)} have \(j < \text{Array.length} \ h' \ a\) by \texttt{auto}

from \texttt{step} have \(\text{sorted: sorted} \ (\text{take} \ j \ (\text{Array.get} \ h' \ a))\) by \texttt{blast}

from \texttt{step(\(4\)) [unfolded insert-elt-def]}

obtain \(\text{key}\) and \(h_1\) and \(i\) and \(h_2\) and \(i'\)

where \(\text{key}: \text{key} = \text{Array.get} \ h' \ a \ ! \ j\)

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 \(\text{simp}[:\text{Ref.get} \ h_2 \ i = i'\)

and \(\text{simp}[:\text{h''} = \text{Array.update} \ a \ i' \ \text{key} \ h_2\)

and \(i' < \text{Array.length} \ h_2 \ a\)

by \(\text{elim effect-bindE effect-nthE effect-lookupE effect-updE}\)

(auto intro: effect-intros, \texttt{metis array-get-alloc effectE execute-ref option})

\textbb{from (effect (ref \(j\)) \(h' \ h_1 \ a\)) have [simp]: Array.get \(h_1 \ a = \text{Array.get} \ h' \ a\)

by (metis \texttt{array-get-alloc effectE execute-ref option.sel})

from \texttt{step and assms(1)}

have \(j < \text{Array.length} \ h_1 \ a \ \text{sorted} \ (\text{take} \ j \ (\text{Array.get} \ h_1 \ a))\) by \texttt{auto}

note shiftr = effect-shiftr [OF ref \(\text{this}\) shiftr' [unfolded shiftr-def], simplified]

have \(i' \leq j\) using shiftr by \texttt{simp}

have \(i' < \text{length} \ (\text{Array.get} \ h_2 \ a)\)

by (metis \(i' < \text{Array.length} \ h_2 \ a\) [length-def])

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

have take-Suc-j: take (Suc \(j\)) (list-update (Array.get \(h_2 \ a\)) \(i' \ \text{key}\)) =

take \(i' \ (\text{Array.get} \ h_2 \ a) @ \text{key} \ # \ \text{take} \ (j - i') \ (\text{drop} \ (\text{Suc} \ i') \ (\text{Array.get} \ h_2 \ a))\)

unfolding upd-cone-take-nth-drop [OF \(i' < \text{length} \ (\text{Array.get} \ h_2 \ a)\)]

by (auto) (metis Suc-diff-denom i' \leq j) take-Suc-Cons)

have \(\text{Array.length} \ h \ a = \text{Array.length} \ h'' \ a\)

using shiftr by (auto) (metis \texttt{step.hyps(1)})

moreover

have multiset-of (Array.get \(h \ a\)) = multiset-of (Array.get \(h'' \ a\))

using shiftr and \texttt{step} by (simp add: key)

moreover

have \(\text{sorted} \ (\text{take} \ (\text{Suc} \ j) \ (\text{Array.get} \ h'' \ a))\)

proof =

from \(\text{ro-shiftr-p.effect-while-post} \ [\text{OF} \ \text{shiftr'} \ \text{[unfolded shiftr-def]]}\)

have \(i' = 0 \lor (0 < i' \ \land \ \text{key} \geq \text{Array.get} \ h_2 \ a \ ! \ (i' - 1))\)

by (auto dest!: \(\text{ro-shiftr-p.success-not-cond-effect}\)

(auto elim!: effect-elims simp: shiftr-p-def)

then show \(\text{thesis}\)
proof
assume \([\text{simp}]\): \(i' = 0\)
have \(*\): take \((\text{Suc} \ j)\) (list-update (Array.get \(h_2\) \(a\)) \(0\) key) =
key \# take \(j\) (drop \(1\) (Array.get \(h_2\) \(a\)))
by \((\text{simp})\) (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 \(\forall x \in \text{set} (\text{take} \ j (\text{drop} \ 1 (\text{Array.get} \ h_2 \ a)))\). key < \(x\) by simp-all
then have sorted (key \# take \(j\) (drop \(1\) (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 < \(i'\) \&\& key ≥ Array.get \(h_2\) \(a\)! (\(i' - 1\))
moreover
have sorted (take \(i'\) (Array.get \(h_2\) \(a\)) \&\& take \((j - i')\) (drop \(\text{Suc} \ i'\) (Array.get \(h_2\) \(a\))))
and \(\forall x \in \text{set} (\text{take} \ i' (\text{Array.get} \ h_2 \ a))\). key < \(x\) by auto
ultimately have \(\forall x \in \text{set} (\text{take} \ i' (\text{Array.get} \ h_2 \ a))\). \(x\) ≤ key
using sorted-take-nth \([\text{OF} - i' < \text{length} (\text{Array.get} \ h_2 \ a)], \ \text{of key}\]
by \((\text{simp add: sorted-append})\)
then show \(?\text{thesis}\) using shiftr by \((\text{auto simp: take-Suc-j sorted-append})\) (metis less-imp-le
sorted.Cons)
qed
qed ultimately
show \(?\text{case}\) by blast
qed

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

export-code insertion-sort in Haskell

end