1 Looping Constructs for Imperative HOL

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.

```plaintext
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.

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

lemma \text{ro-cond: ro-cond p}
using read-only by (simp add: ro-cond-def)

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

lemma \text{execute-while-unfolds [execute-simps]}:
\begin{align*}
\text{success p h} & \implies \text{cond p h} \implies \text{execute (while p f) h} = \text{execute (f >\& while p f) h} \\
\text{success p h} & \implies \neg \text{cond p h} \implies \text{execute (while p f) h} = \text{execute (return (\)) h}
\end{align*}
by (auto simp: while-simps execute-simps)

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

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

lemma \text{success-not-cond-effect: success p h} \implies \neg \text{cond p h} \implies \text{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 \text{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: cond-def)
apply (metis effectE ro-cond.read-only)+
done

A rule for proving partial correctness of while loops.

lemma \text{ro-cond-effect-while-induct:}
assumes ro-cond p
assumes effect (while p f) h h' u 
and I h
and $h, h'$. $I h \implies \text{success } p \implies \text{cond } p \implies \text{effect } f h' u \implies I h'$

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 $\text{effect } p h b$

and $*: \text{effect } (\text{if } b \text{ then } f \gg w p f \text{ else return } ()) h h' u$

using $\text{1.hyps and (ro-cond } p)$

by (auto elim!: $\text{effect-elims intro: effect-intros} \ (\text{metis effectE ro-cond.read-only})$

then have $\text{cond: success } p \text{ cond } p h = b$ by (auto simp: $\text{cond-def elim!}$:

$\text{effect-elims effectE}$)

show $?\text{case}$

proof (cases $b$)

assume $\neg b$

then show $?\text{thesis using } *$ and $\langle I h \rangle$ by (auto elim: $\text{effect-elims}$)

next

assume $b$

moreover

with $*$ obtain $h''$ and $r$

where $\text{effect } f h h'' r$ and $\text{effect } (w p f) h'' h' u$ by (auto elim: $\text{effect-elims}$)

moreover

ultimately

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

qed

qed $\text{fact}$

lemma $\text{effect-success-conv:}$

$(\exists h'. \text{ effect } c h h' (\land I h')) \iff \text{success } c h \land I (\text{snd (the (execute c h)))}$

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

context $\text{ro-cond}$

begin

lemmas $\text{effect-while-post} = \text{ro-cond-effect-while-post} \ [\text{OF ro-cond} \text{ and}\n$

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

[\text{OF ro-cond}]

A rule for proving total correctness of while loops.

lemma $\text{wf-while-induct} \ [\text{consumes 1, case-names success-cond success-body base}\n$

step]:

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

and $\text{success-p: } \bigwedge h. I h \implies \text{success } p h$ — the loop-condition terminates

and $\text{success-f: } \bigwedge h. I h \implies \text{success } p h \implies \text{cond } p h \implies \text{success } f h$ — the loop-body terminates

and $I h$ — the invariant holds before the loop is entered

and $\text{step: } \bigwedge h h' r. I h \implies \text{success } p h \implies \text{cond } p h \implies \text{effect } f h h' r \implies (h', r)$,
\( h \) \( \in \, R \land I h' \)

— the invariant is preserved by iterating the loop

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

using \( \psi f \, R \) and \( I h \)

proof (induction \( h \))

\begin{verbatim}
    case (less \( h \))
    show \( ? \) case
        proof (cases cond \( p \, h \))
            assume \( \neg \) cond \( p \, h \)
            then show \( ? \) thesis
                using \( I \, h' \) and \( \text{success-p (of } h \) \)
                by (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 (auto simp: success-def effect-def)
            with \( \text{less.IH (of } h' \) \)
            show \( ? \) thesis
                using \( \text{cond } p \, h \) by (auto simp: execute-simps effect-def)
        qed
    qed
\end{verbatim}

A rule for proving termination of while loops.

lemmas

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

    \text{=}

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

end

1.2 For Loops

fun for :: 'a list \( \Rightarrow \) ('a \( \Rightarrow \) 'b Heap) \( \Rightarrow \) unit Heap
where

\begin{verbatim}
    for [] \( f = \text{return } () \) |
    for (x # xs) \( f = f \, x \Rightarrow \) for xs \( f \)
\end{verbatim}

A rule for proving partial correctness of for loops.

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

\begin{verbatim}
    assumes \( i \leq j \)
    and \( \text{effect (for } [i ..< j] \, f \) \, h \, h' \) \, u
    and \( I \, i \, h \)
    and \( \{k \, h \, h' \, r. \, i \leq k \Rightarrow k < j \Rightarrow I \, k \, h \Rightarrow \text{effect } (f \, k) \, h \, h' \, r \Rightarrow I \, (\text{Suc } k) \) \)
    \( h' \)
    shows \( I \, j \, h' \)

    using \( \text{asms} \)

    proof (induction \( j - i \) arbitrary: \( i \, h \))
        case 0
        then show \( ? \) case by (auto elim: effect-elims)
    next
\end{verbatim}

4
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⟩
  obtain h" r
  where *: effect (f i) h h" r
  and **: effect (f (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 \( k \in \mathbb{N} \land I k h \implies i \leq k \implies k < j \implies \text{success (f k) h} \)
  and I i h
  and h' r
  shows \( \exists h'. \text{effect (f i ..< j) h h'} (\text{is } ?P i h) \)
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'
    by (auto elim: success-effectE)
    moreover
    then have I (Suc i) h' using Suc by auto
    moreover
    qed
have ?P (Suc i) h'
by (rule Suc(1) [OF \k. k = j - Suc i \Suc i \Suc i \leq j; Suc(4) \I (Suc i) h' Suc(6)])
auto
ultimately
show ?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 ≡ Array.apd 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] (\j. do {
     — Insert a[j] into the sorted subarray a[1 .. j - l].
     key ← a.(j);
     i ← ref j;
     while (do {


\[ \begin{align*}
i' & \leftarrow ! i; \\
\text{if } i' > 0 \text{ then do } \{x \leftarrow a.i' - 1; \text{return } (x > \text{key})\} \\
\text{else return False} \\
\} \\
\end{align*} \]

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

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

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

definition shiftr-f a i = do 
\begin{align*}
i' & \leftarrow ! i; \\
x & \leftarrow a.i' - 1; \\
a.i' & \leftarrow x; \\
i & := i' - 1 \\
\} \\
\end{align*} 

definition shiftr a key i = while (shiftr-p a key i) (shiftr-f a i) 

definition insert-elt a = (\lambda j. do 
\begin{align*}
\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. for \[1..<l \] (insert-elt a)) 

lemma insertion-sort-alt-def: 
insertion-sort a = (Array.len a \gg= 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 \(\text{Ref.get } h' i = \text{Ref.get } h i - 1 \wedge\)
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 j i = take (Ref.get h i) (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'
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) ∧
in i 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'' and ⟨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 ⟨cond (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)
1))

by auto

from step have \( ?i h' < \text{length} (\text{Array}.\text{get} h' a) \)
and **: \( ?i h' - (Suc 0) \leq ?i h' \leq \text{length} (\text{Array}.\text{get} h' a) \)
and \( ?i h' \leq j \)
and \( ?i h' < Suc j \)
and IH: \( \text{init} h a j = \text{both} h' a j i \)
by (auto simp add: \text{Array}.length-def)

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

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

moreover have mset (\text{list-update} (\text{Array}.\text{get} h a) j key) =
  mset (\text{list-update} (\text{Array}.\text{get} h'' a) (?i h'') 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 \( \text{init} h a j = \text{both} h'' a j i \)
using \( \langle 0 < ?i h' \rangle \) and \( \langle ?i h' \leq j \rangle \) and \( \langle ?i h' < \text{length} (\text{Array}.\text{get} h' a) \rangle \) and **
and IH
by (auto simp: IH upd-conv-take-nth-drop Suc-diff-le min-absorb1)
  (metis Suc-lessD Suc-pred 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 \( \langle ?i h' \leq j \rangle \) and \( \langle ?i h' < \text{length} (\text{Array}.\text{get} h' a) \rangle \) 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 (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 \( \langle 0 < i \rangle \) and \( \langle i < \text{length} xs \rangle \)
by (metis Suc-diff-1 less-imp-diff-less take-Suc-conv-app-nth)
then show \( \exists \)thesis
  using \( \text{sorted} \ (\text{take } i \ 	ext{xs}) \) and \( \text{xs} ! (i - 1) \leq y \)
  by (auto simp: \text{sorted-append})

qed

lemma effect-for-insert-elt:
assumes \( l \leq \text{Array.length } h \ a \)
and \( l \leq l' \)
and effect \( (\text{for } [1..< l] \ (\text{insert-elt } a)) \) \( h \ h' u \)
shows \( \text{Array.length } h \ a = \text{Array.length } h' \ a \wedge \)
\( \text{sorted} \ (\text{take } l \ (\text{Array.get } h' \ a)) \wedge \)
\( \text{mset} \ (\text{Array.get } h \ a) = \text{mset} \ (\text{Array.get } h' \ a) \)
using \text{assms}(2–)

proof (induction \( l \ h' \) rule: effect-for-induct)
  case base
  show \( \exists \text{case} \) by (cases \text{Array.get } h \ a) simp-all

next
  case (step \( j \ h' \ h'' u \))
  with \text{assms}(\( i \)) have \( j < \text{Array.length } h \ a \) by auto
  from step have \text{sorted}: \text{sorted}(\text{take } j \ (\text{Array.get } h' \ a)) \text{ by blast}
  from step(\( 3 \)) [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 \ 1 \ j
    and effect \( \text{ref } j \) \( h' \ h_1 \ i \)
    and \( \text{ref } j \) = \text{Ref.get } h_1 \ i = j
    and \( \text{shiftr}' \) = effect(\text{shiftr a key i}) \( h_1 \ h_2 \)
    and \( \text{simp} \): \text{Ref.get } h_2 \ i = i'
    and \( \text{simp} \): \( h'' = \text{Array.update a i' key } h_2 \)
    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 \( \text{ref } j \) \( h' \ h_1 \ i \)) have \( \text{simp}: \text{Array.get } h_1 \ a = \text{Array.get } h' \ a \)
    by (metis \text{array-get-alloc} effectE execute-ref option.sel)

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

  from step and \text{assms}(\( 1 \))
  have \( j < \text{Array.length } h_1 \ a \) \text{sorted}: \text{sorted}(\text{take } j \ (\text{Array.get } h_1 \ a)) \text{ by auto}
  note shiftr = effect-shiftr \( \text{OF } \text{ref } j \) \( \text{this shiftr}' \) [unfolded shiftr-def, simplified]
  have \( i' < j \) using \text{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 \( \text{simp}: \text{min} \ (\text{Suc } j) \ i' = i' \) using \( i' < j \) by simp
  have \( \text{simp}: \text{min} \ (\text{length} \ (\text{Array.get } h_2 \ a)) \ i' = i' \)
    using \( i' < \text{length} \ (\text{Array.get } h_2 \ a) \) by (simp)
  have take-Suc-j: take(\text{Suc } j) \ (\text{list-update} \ (\text{Array.get } h_2 \ a) \ i' \ key) =
    take \( i' \) \ (\text{Array.get } h_2 \ a) \oplus \text{key} # \text{take} \ (j - i') \ (\text{drop} \ (\text{Suc } i') \ (\text{Array.get } h_2 \ a)) \)

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

have \ \texttt{Array.length} \ h \ a = \texttt{Array.length} \ h'' \ a 
using \ \texttt{shiftr} \ by \ (\texttt{auto}) \ (\texttt{metis} \ \texttt{step.IH})

moreover
have \ \texttt{mset} \ (\texttt{Array.get} \ h \ a) = \texttt{mset} \ (\texttt{Array.get} \ h'' \ a) 
using \ \texttt{shiftr} \ and \ \texttt{step} \ by \ (\texttt{simp add: key})
moreover
have \ \texttt{sorted} \ (\texttt{take} \ (\texttt{Suc} \ j) \ (\texttt{Array.get} \ h'' \ a))

proof
given \ \texttt{ro-shiftr-p} \ from \ \texttt{shiftr} \ \texttt{success-not-cond-effect}
\texttt{(auto elim!: effect-elims \ simp: shiftr-p-def)}
then show \ \texttt{?thesis}

proof
assume \ [\texttt{simp}]: \ i' = 0 

have \ \texttt{*}: \ \texttt{take} \ (\texttt{Suc} \ j) \ (\texttt{list-update} \ (\texttt{Array.get} \ h_2 \ a) \ 0 \ \texttt{key}) = 
kev' \# \ \texttt{take} \ j \ (\texttt{drop} \ 1 \ (\texttt{Array.get} \ h_2 \ a))
by \ (\texttt{simp}) \ (\texttt{metis} \ i' = 0: \ \texttt{append-Nil} \ \texttt{take-Suc-j} \ \texttt{diff-zero} \ \texttt{take-0})

from \ \texttt{sorted} \ and \ \texttt{shiftr} 
\texttt{have} \ \texttt{sorted} \ (\texttt{take} \ j \ (\texttt{drop} \ 1 \ (\texttt{Array.get} \ h_2 \ a)))
and \ \forall \ x \in \texttt{set} \ (\texttt{take} \ j \ (\texttt{drop} \ 1 \ (\texttt{Array.get} \ h_2 \ a))). \ \texttt{key} < \texttt{x} \ \texttt{by} \ \texttt{simp-all}
then have \ \texttt{sorted} \ (\texttt{key} \# \ \texttt{take} \ j \ (\texttt{drop} \ 1 \ (\texttt{Array.get} \ h_2 \ a)))
by \ (\texttt{metis} \ \texttt{less-imp-le} \ \texttt{sorted.simps}(2))
then show \ \texttt{?thesis} \ by \ (\texttt{simp add: \ *} )

next
assume \ 0 < i' \land \texttt{key} \geq \texttt{Array.get} \ h_2 \ a \ ! \ (i' - 1)

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

qed

qed

ultimately
show \ \texttt{?case} \ by \ \texttt{blast}

lemma \ \texttt{effect-insertion-sort}:
assumes \ \texttt{effect} \ (\texttt{insertion-sort} \ a) \ h \ h' \ u 
shows \ \texttt{mset} \ (\texttt{Array.get} \ h \ a) = \texttt{mset} \ (\texttt{Array.get} \ h' \ a) \land \texttt{sorted} \ (\texttt{Array.get} \ h' \ a)
using \ \texttt{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
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 h_1 where effect (ref k) h h_1
 and [sdep]: Ref.get h_1 i = k
 and [sdep]: Array.length h_1 a = Array.length h a
 by (auto simp: ref-def length-def)
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 shiftr-def)
moreover
have effect (! i) h_3 h_2 (Ref.get h_2 i)
 and Ref.get h_3 i ≤ Ref.get h_1 i
 and Ref.get h_2 i < Array.length h_2 a
 using effect-shiftr-index [OF →] and effect-shiftr-length [OF →] and assms
by (auto intro!: effect-intros)
moreover
then obtain h_4 and r where effect (a.(Ref.get h_2 i) ← key) h_2 h_3 r
 using assms by (auto simp: execute-simps)
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 ≤ 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(4) [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 [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))
  using shiftr and step by (simp add: key)

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 elim 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 \( \forall x \in set \ (take j (drop 1 (Array.get h2 a))).\ key < x \) by simp-all
then have sorted (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 sorted (take $i'$ (Array.get h2 a) @ take $(j - i')$ (drop (Suc $i'$) (Array.get h2 a)))
and \( \forall x \in set \ (take (j - i') (drop (Suc i') (Array.get h2 a))).\ key < x \)
using shiftr by auto
ultimately have \( \forall x \in set \ (take i' (Array.get h2 a)).\ x < key \)
using sorted-take-nth [OF - $i'$ < length (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

lemma insertion-sort-correct:
\( \exists h'.\ effect \ (insertion-sort a) h h' a \land \)
\( mset \ (Array.get h a) = mset \ (Array.get h' a) \land \)
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-elt-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

end