

# Imperative Insertion Sort

Christian Sternagel

March 17, 2025

## Contents

<b>1</b>	<b>Looping Constructs for Imperative HOL</b>	<b>1</b>
1.1	While Loops . . . . .	1
1.2	For Loops . . . . .	4
<b>2</b>	<b>Insertion Sort</b>	<b>6</b>
2.1	The Algorithm . . . . .	6
2.2	Partial Correctness . . . . .	7
2.3	Total Correctness . . . . .	12

## 1 Looping Constructs for Imperative HOL

```
theory Imperative-Loops
imports HOL-Imperative-HOL.Imperative-HOL
begin
```

### 1.1 While Loops

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

```
partial-function (heap) while :: bool Heap  $\Rightarrow$  'b Heap  $\Rightarrow$  unit Heap
where
  [code]: while p f = do {
    b  $\leftarrow$  p;
    if b then f  $\gg$  while p f
    else return ()
  }
```

```
definition cond p h  $\longleftrightarrow$  fst (the (execute p h))
```

A locale that restricts to read-only loop conditions.

```
locale ro-cond =
  fixes p :: bool Heap
```

**assumes** *read-only*:  $success\ p\ h \implies snd\ (the\ (execute\ p\ h)) = h$   
**begin**

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

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

**lemma** *execute-while-unfolds* [*execute-simps*]:  
 $success\ p\ h \implies cond\ p\ h \implies execute\ (while\ p\ f)\ h = execute\ (f\ \gg\ while\ p\ f)\ h$   
 $success\ p\ h \implies \neg\ cond\ p\ h \implies execute\ (while\ p\ f)\ h = execute\ (return\ ())\ h$   
**by** (*auto* *simp*: *while.simps* *execute-simps*)

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

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

**lemma** *success-not-cond-effect*:  
 $success\ p\ h \implies \neg\ cond\ p\ h \implies effect\ p\ h\ h\ False$   
**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**  $success\ p\ h' \wedge \neg\ cond\ p\ h'$   
**using** *assms*(1)  
**apply** (*induct* *rule*: *while.raw-induct* [*OF* - *assms*(2)])  
**apply** (*auto* *elim!*: *effect-elim* *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**  $\bigwedge h h' u. I h \implies \text{success } p h \implies \text{cond } p h \implies \text{effect } f h h' u \implies I h'$   
**shows**  $I h'$   
**using**  $\text{assms}(1, 3-)$   
**proof** (*induction*  $p f h h' u$  *rule: while.raw-induct*)  
**case** ( $1 w p f h h' u$ )  
**obtain**  $b$   
**where**  $\text{effect } p h h b$   
**and**  $*$ :  $\text{effect } (\text{if } b \text{ then } f \gg w p f \text{ else return } ()) h h' u$   
**using**  $1.\text{hyps}$  **and**  $\langle \text{ro-cond } p \rangle$   
**by** (*auto elim!*: *effect-elim intro: effect-intros*) (*metis effectE ro-cond.read-only*)  
**then have**  $\text{cond}: \text{success } p h \text{ cond } p h = b$  **by** (*auto simp: cond-def elim!*: *effect-elim effectE*)  
**show**  $?case$   
**proof** (*cases*  $b$ )  
**assume**  $\neg b$   
**then show**  $?thesis$  **using**  $*$  **and**  $\langle I h \rangle$  **by** (*auto elim: effect-elim*)  
**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: effect-elim*)  
**moreover**  
**ultimately**  
**show**  $?thesis$  **using**  $1$  **and**  $\text{cond}$  **by** *blast*  
**qed**  
**qed fact**

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

**context** *ro-cond*  
**begin**

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

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**  $\text{success-p}: \bigwedge h. I h \implies \text{success } p h$  — 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**  $\text{step}: \bigwedge h h' r. I h \implies \text{success } p h \implies \text{cond } p h \implies \text{effect } f h h' r \implies (h',$

```

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-elim)
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-elim)
  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' () ∧ I j h' (is ?P i h)
using assms
proof (induction j - i arbitrary: i h)
  case 0
  then show ?case by (auto simp: effect-def execute-simps)
next
  case (Suc k)
  show ?case
  proof (cases j = i)
    assume j = i
    with Suc show ?thesis by auto
  next
    assume j ≠ i
    with ⟨i ≤ j⟩ and ⟨Suc k = j - i⟩
    have i < j and k = j - Suc i and Suc i ≤ j by auto
    then have [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 ?P (Suc i) h'
      by (rule Suc(1) [OF ‹k = j - Suc i› ‹Suc i ≤ j› Suc(4) ‹I (Suc i) h'›
        Suc(6)]) auto
    ultimately
    show ?case by (auto simp: effect-def execute-simps)
  qed
end

```

A rule for proving termination of for loops.

```

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

end

```

## 2 Insertion Sort

```

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

```

### 2.1 The Algorithm

```

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

```

```

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

```

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

To avoid ambiguities with existing syntax we use OCaml's notation for accessing ( $a.(i)$ ) and updating ( $a.(i) \leftarrow x$ ) 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' ← ! i;
    if i' > 0 then do {x ← a.(i' - 1); return (x > key)}
    else return False}
  (do {
    i' ← ! i;
    x ← a.(i' - 1);
    a.(i') ← x;
    i := i' - 1
  });
  i' ← ! i;
  a.(i') ← key
})
}

```

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

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

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

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

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

**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** *Ref.get*  $h'$   $i =$  *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*:

$Ref.get\ h\ i < Array.length\ h\ a \implies success\ (shiftr-p\ a\ key\ i)\ h$

**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*]:  $right\ h\ a\ j\ i = take\ (j - Ref.get\ h\ i)\ (drop\ (Ref.get\ h\ i + 1)\ (Array.get\ h\ a))$

**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**  $Array.length\ h\ a = Array.length\ h'\ a \wedge$

$?i\ h' \leq j \wedge$

$mset\ (list-update\ (Array.get\ h\ a)\ j\ key) =$

$mset\ (list-update\ (Array.get\ h'\ a)\ (?i\ h')\ key) \wedge$

$ini\ h\ a\ j = both\ h'\ a\ j\ i \wedge$

*sorted* ( $both\ h'\ a\ j\ i$ )  $\wedge$

$(\forall x \in set\ (right\ h'\ a\ j\ i). x > key)$

**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**  $\langle success\ (shiftr-p\ a\ key\ i)\ h' \rangle$  **and**  $\langle cond\ (shiftr-p\ a\ key\ i)\ h' \rangle$

**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*  $\langle effect\ (shiftr-f\ a\ i)\ h'\ h''\ u \rangle$ ]

**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))  
 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 ?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 sorted (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 ?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 ?thesis
  using  $\langle \text{sorted (take } i \text{ xs)} \rangle$  and  $\langle \text{xs ! (} i - 1 \text{)} \leq y \rangle$ 
  by (auto simp: sorted-append)
qed

lemma effect-for-insert-elt:
  assumes  $l \leq \text{Array.length } h \ a$ 
  and  $1 \leq l$ 
  and effect (for [1 ..< l] (insert-elt a)) h h' u
  shows  $\text{Array.length } h \ a = \text{Array.length } h' \ a \wedge$ 
   $\text{sorted (take } l \ (\text{Array.get } h' \ a)) \wedge$ 
   $\text{mset (Array.get } h \ a) = \text{mset (Array.get } h' \ a)$ 
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  $j < \text{Array.length } h' \ a$  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: key = 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  $\langle \text{effect (ref } j \text{) } h' \ h_1 \ i \rangle$  have [simp]:  $\text{Array.get } h_1 \ a = \text{Array.get } h' \ a$ 
  by (metis array-get-alloc effectE execute-ref option.sel)

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

from step and assms(1)
  have  $j < \text{Array.length } h_1 \ a$  sorted (take j (Array.get h_1 a)) by auto
  note shiftr = effect-shiftr [OF ref1 this shiftr' [unfolded shiftr-def], simplified]
  have  $i' \leq j$  using shiftr by simp

have  $i' < \text{length (Array.get } h_2 \ a)$ 
  by (metis i' < Array.length h_2 a length-def)
have [simp]:  $\min (\text{Suc } j) \ i' = i'$  using  $\langle i' \leq j \rangle$  by simp
have [simp]:  $\min (\text{length (Array.get } h_2 \ a)) \ i' = i'$ 
  using  $\langle i' < \text{length (Array.get } h_2 \ a) \rangle$  by (simp)
have take-Suc-j: take (Suc j) (list-update (Array.get h_2 a) i' key) =
  take i' (Array.get h_2 a) @ key # take (j - i') (drop (Suc i') (Array.get h_2 a))

```

```

unfolding upd-conv-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.IH)
moreover
have mset (Array.get h a) = mset (Array.get h'' a)
  using shiftr and step by (simp add: key)
moreover
have sorted (take (Suc j) (Array.get h'' a))
proof -
  from ro-shiftr-p.effect-while-post [OF shiftr' [unfolded shiftr-def]]
    have i' = 0 ∨ (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 [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 ∀ x ∈ 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 ≥ 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 ∀ x ∈ set (take (j - i') (drop (Suc i') (Array.get h2 a))). key < x
    using shiftr by auto
  ultimately have ∀ x ∈ 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
qed
ultimately
show ?case by blast
qed

lemma effect-insertion-sort:
assumes effect (insertion-sort a) h h' u
shows mset (Array.get h a) = mset (Array.get h' a) ∧ sorted (Array.get h' a)
using assms

```

```

apply (cases Array.length h a)
apply (auto elim!: effect-elims simp: insertion-sort-def Array.length-def)[1]
unfolding insertion-sort-def
unfolding shiftr-p-def [symmetric] shiftr-f-def [symmetric]
unfolding shiftr-def [symmetric] insert-elt-def [symmetric]
apply (elim effect-elims)
apply (simp only:)
apply (subgoal-tac Suc nat ≤ Array.length h a)
apply (drule effect-for-insert-elt)
apply (auto simp: Array.length-def)
done

```

## 2.3 Total Correctness

**lemma** *success-shiftr-f*:

```

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

```

**lemma** *success-shiftr*:

```

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

```

**lemma** *effect-shiftr-length*:

```

assumes effect (shiftr a key i) h h' u
shows Array.length h' a = Array.length h a
using assms unfolding shiftr-def

```

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

**lemma success-insert-elt:**

assumes  $k < \text{Array.length } h \ a$   
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<sub>1</sub> where effect (ref k) h h<sub>1</sub> i  
and [simp]: Ref.get h<sub>1</sub> i = k  
and [simp]: Array.length h<sub>1</sub> a = Array.length h a

by (auto simp: ref-def length-def) (metis Ref.get-alloc array-get-alloc effect-heapI)

moreover

obtain h<sub>2</sub> where \*: effect (shiftr a key i) h<sub>1</sub> h<sub>2</sub> ()  
using success-shiftr [of h<sub>1</sub> i a key] and assms  
by (auto simp: success-def effect-def shiftr-def)

moreover

have effect (! i) h<sub>2</sub> h<sub>2</sub> (Ref.get h<sub>2</sub> i)  
and Ref.get h<sub>2</sub> i ≤ Ref.get h<sub>1</sub> i  
and Ref.get h<sub>2</sub> i < Array.length h<sub>2</sub> a  
using effect-shiftr-index [OF \*] and effect-shiftr-length [OF \*] and assms  
by (auto intro!: effect-intros)

moreover

then obtain h<sub>3</sub> and r where effect (a.(Ref.get h<sub>2</sub> i) ← key) h<sub>2</sub> h<sub>3</sub> r  
using assms by (auto simp: effect-def execute-simps)

ultimately

have effect (insert-elt a k) h h<sub>3</sub> 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  $\exists h'. \text{effect (for } [1 \ ..< l] \text{ (insert-elt a)) } h \ h' \ () \wedge$   
 $\text{Array.length } h \ a = \text{Array.length } h' \ a \wedge$   
 $\text{sorted (take } l \text{ (Array.get } h' \ a)) \wedge$   
 $\text{mset (Array.get } h \ a) = \text{mset (Array.get } h' \ a)$

using assms(2)

**proof** (induction rule: for-induct)

case (succeed k h)

then show ?case using assms and success-insert-elt [of k h a] by auto

next

case base

show ?case by (cases Array.get h a) simp-all

next

case (step j h' h'' u)

**with** *assms*(1) **have**  $j < \text{Array.length } h' \ a$  **by** *auto*  
**from** *step* **have** *sorted*: *sorted* (take  $j$  (Array.get  $h' \ a$ )) **by** *blast*  
**from** *step*(4) [*unfolded insert-elt-def*]  
**obtain** *key* **and**  $h_1$  **and**  $i$  **and**  $h_2$  **and**  $i'$   
**where** *key*:  $key = \text{Array.get } h' \ a \ ! \ j$   
**and** *effect* (ref  $j$ )  $h' \ h_1 \ i$   
**and** *ref*<sub>1</sub>:  $\text{Ref.get } h_1 \ i = j$   
**and** *shiftr'*: *effect* (shiftr  $a \ key \ i$ )  $h_1 \ h_2 \ ()$   
**and** [*simp*]:  $\text{Ref.get } h_2 \ i = i'$   
**and** [*simp*]:  $h'' = \text{Array.update } a \ i' \ key \ h_2$   
**and**  $i' < \text{Array.length } h_2 \ a$   
**by** (*elim effect-bindE effect-nthE effect-lookupE effect-updE*)  
(*auto intro: effect-intros, metis effect-refE*)

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

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

**from** *step* **and** *assms*(1)  
**have**  $j < \text{Array.length } h_1 \ a$  *sorted* (take  $j$  (Array.get  $h_1 \ a$ )) **by** *auto*  
**note** *shiftr* = *effect-shiftr* [*OF ref*<sub>1</sub> *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*  $\langle i' < \text{Array.length } h_2 \ a \rangle$  *length-def*)  
**have** [*simp*]:  $\min (\text{Suc } j) \ i' = i'$  **using**  $\langle i' \leq j \rangle$  **by** *simp*  
**have** [*simp*]:  $\min (\text{length } (\text{Array.get } h_2 \ a)) \ i' = i'$   
**using**  $\langle i' < \text{length } (\text{Array.get } h_2 \ a) \rangle$  **by** (*simp*)  
**have** *take-Suc-j*:  $\text{take } (\text{Suc } j) \ (\text{list-update } (\text{Array.get } h_2 \ a) \ i' \ key) =$   
 $\text{take } i' \ (\text{Array.get } h_2 \ a) \ @ \ key \ \# \ \text{take } (j - i') \ (\text{drop } (\text{Suc } i') \ (\text{Array.get } h_2 \ a))$   
**unfolding** *upd-conv-take-nth-drop* [*OF*  $\langle i' < \text{length } (\text{Array.get } h_2 \ a) \rangle$ ]  
**by** (*auto*) (*metis Suc-diff-le*  $\langle i' \leq j \rangle$  *take-Suc-Cons*)

**have**  $\text{Array.length } h \ a = \text{Array.length } h'' \ a$   
**using** *shiftr* **by** (*auto*) (*metis step.hyps*(1))  
**moreover**  
**have** *mset* (Array.get  $h \ a$ ) = *mset* (Array.get  $h'' \ a$ )  
**using** *shiftr* **and** *step* **by** (*simp add: key*)  
**moreover**  
**have** *sorted* (take (Suc  $j$ ) (Array.get  $h'' \ a$ ))  
**proof** –  
**from** *ro-shiftr-p.effect-while-post* [*OF shiftr'* [*unfolded shiftr-def*]]  
**have**  $i' = 0 \vee (0 < i' \wedge key \geq \text{Array.get } h_2 \ a \ ! \ (i' - 1))$   
**by** (*auto dest!*: *ro-shiftr-p.success-not-cond-effect*)  
(*auto elim!*: *effect-elims simp: shiftr-p-def*)  
**then show** *?thesis*  
**proof**  
**assume** [*simp*]:  $i' = 0$

```

have *: take (Suc j) (list-update (Array.get h2 a) 0 key) =
  key # take j (drop 1 (Array.get h2 a))
  by (simp) (metis ‹i' = 0› append-Nil take-Suc-j diff-zero take-0)
from sorted and shiftr
  have sorted (take j (drop 1 (Array.get h2 a)))
  and  $\forall x \in \text{set } (\text{take } j \text{ (drop } 1 \text{ (Array.get } h_2 \text{ a)}))$ . 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'  $\wedge$  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  $\forall x \in \text{set } (\text{take } (j - i') \text{ (drop } (\text{Suc } i') \text{ (Array.get } h_2 \text{ a)}))$ . key < x
  using shiftr by auto
ultimately have  $\forall x \in \text{set } (\text{take } i' \text{ (Array.get } h_2 \text{ a)})$ . x  $\leq$  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
qed
ultimately
show ?case by blast
qed

```

```

lemma insertion-sort-correct:
   $\exists h'$ . effect (insertion-sort a) h h' u  $\wedge$ 
    mset (Array.get h a) = mset (Array.get h' a)  $\wedge$ 
    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 inser-
tion-sort-alt-def sort-upto-def)
qed

```

**export-code** insertion-sort **in** Haskell

end