

# Imperative Insertion Sort

Christian Sternagel

February 23, 2021

## Contents

<b>1</b>	<b>Looping Constructs for Imperative HOL</b>	<b>1</b>
1.1	While Loops . . . . .	1
1.2	For Loops . . . . .	3
<b>2</b>	<b>Insertion Sort</b>	<b>4</b>
2.1	The Algorithm . . . . .	4
2.2	Partial Correctness . . . . .	6
2.3	Total Correctness . . . . .	7

## 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*:  $ro\ cond\ p$   
*<proof>*

**lemma** *cond-cases* [*execute-simps*]:  
 $success\ p\ h \implies cond\ p\ h \implies execute\ p\ h = Some\ (True,\ h)$   
 $success\ p\ h \implies \neg\ cond\ p\ h \implies execute\ p\ h = Some\ (False,\ h)$   
*<proof>*

**lemma** *execute-while-unfolds* [*execute-simps*]:  
 $success\ p\ h \implies cond\ p\ h \implies execute\ (while\ p\ f)\ h = execute\ (f\ \gg\ while\ p\ f)\ h$   
 $success\ p\ h \implies \neg\ cond\ p\ h \implies execute\ (while\ p\ f)\ h = execute\ (return\ ())\ h$   
*<proof>*

**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*:  $success\ p\ h \implies \neg\ cond\ p\ h \implies success\ (while\ p\ f)\ h$   
*<proof>*

**lemma** *success-cond-effect*:  
 $success\ p\ h \implies cond\ p\ h \implies effect\ p\ h\ h\ True$   
*<proof>*

**lemma** *success-not-cond-effect*:  
 $success\ p\ h \implies \neg\ cond\ p\ h \implies effect\ p\ h\ h\ False$   
*<proof>*

**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**  $success\ p\ h' \wedge \neg\ cond\ p\ h'$   
*<proof>*

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**  $\bigwedge h\ h'\ u.\ I\ h \implies success\ p\ h \implies cond\ p\ h \implies effect\ f\ h\ h'\ u \implies I\ h'$   
**shows**  $I\ h'$   
*<proof>*

**lemma** *effect-success-conv*:  
 $(\exists h'. \text{effect } c \ h \ h' \ () \wedge I \ h') \longleftrightarrow \text{success } c \ h \wedge I \ (\text{snd } (\text{the } (\text{execute } c \ h)))$   
*<proof>*

**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*:  $\bigwedge h. I \ h \Longrightarrow \text{success } p \ h$  — the loop-condition terminates  
**and** *success-f*:  $\bigwedge h. I \ h \Longrightarrow \text{success } p \ h \Longrightarrow \text{cond } p \ h \Longrightarrow \text{success } f \ h$  — the loop-body terminates  
**and** *I h* — the invariant holds before the loop is entered  
**and** *step*:  $\bigwedge h \ h' \ r. I \ h \Longrightarrow \text{success } p \ h \Longrightarrow \text{cond } p \ h \Longrightarrow \text{effect } f \ h \ h' \ r \Longrightarrow (h', h) \in R \wedge I \ h'$   
— the invariant is preserved by iterating the loop  
**shows**  $\exists h'. \text{effect } (\text{while } p \ f) \ h \ h' \ () \wedge I \ h'$   
*<proof>*

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*  $\Rightarrow$  (*'a*  $\Rightarrow$  *'b Heap*)  $\Rightarrow$  *unit Heap*  
**where**  
*for* [] *f* = *return ()* |  
*for* (*x # xs*) *f* = *f x*  $\gg$  *for xs f*

A rule for proving partial correctness of for loops.

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

**assumes**  $i \leq j$   
**and** *effect* (*for* [*i ..< j*] *f*) *h h' u*  
**and** *I i h*  
**and**  $\bigwedge k \ h \ h' \ r. i \leq k \Longrightarrow k < j \Longrightarrow I \ k \ h \Longrightarrow \text{effect } (f \ k) \ h \ h' \ r \Longrightarrow I \ (\text{Suc } k) \ h'$

**shows**  $I j h'$   
 $\langle proof \rangle$

A rule for proving total correctness of for loops.

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

**assumes**  $i \leq j$   
**and**  $\bigwedge k h. I k h \implies i \leq k \implies k < j \implies success (f k) h$   
**and**  $I i h$   
**and**  $\bigwedge k h h' r. I k h \implies i \leq k \implies k < j \implies effect (f k) h h' r \implies I (Suc k) h'$   
**shows**  $\exists h'. effect (for [i ..< j] f) h h' () \wedge I j h' (\mathbf{is} \ ?P i h)$   
 $\langle proof \rangle$

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* ::  $'a::heap\ array \Rightarrow nat \Rightarrow 'a \Rightarrow 'a\ array\ Heap\ ((-)'(-) \leftarrow / -) [1000, 0, 13] 14)$

**where**

$a.(i) \leftarrow x \equiv Array.upd\ i\ x\ a$

**abbreviation** *array-nth* ::  $'a::heap\ array \Rightarrow nat \Rightarrow 'a\ Heap\ (-)'(-) [1000, 0] 14)$

**where**

$a.(i) \equiv Array.nth\ a\ i$

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' ← ! i; if i' > 0 then do {x ← a.(i' - 1); return (x > key)} else return False})*

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

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

**definition** *insert-elt a = (λj. do {*  
*key ← a*.(*j*);  
*i* ← ref *j*;  
*shiftr a key i*;  
*i'* ← ! *i*;  
*a*.(*i'*) ← *key*  
*})*

**definition** *sort-upto a = (λl. for [1 ..< l] (insert-elt a))*

**lemma** *insertion-sort-alt-def:*  
*insertion-sort a = (Array.len a ≫= sort-upto a)*  
*⟨proof⟩*

## 2.2 Partial Correctness

**lemma** *effect-shiftr-f*:

**assumes** *effect* (*shiftr-f a i*) *h h' u*

**shows**  $\text{Ref.get } h' \ i = \text{Ref.get } h \ i - 1 \wedge$

$\text{Array.get } h' \ a = \text{list-update } (\text{Array.get } h \ a) \ (\text{Ref.get } h \ i) \ (\text{Array.get } h \ a ! (\text{Ref.get } h \ i - 1))$

*<proof>*

**lemma** *success-shiftr-p*:

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

*<proof>*

**interpretation** *ro-shiftr-p*: *ro-cond shiftr-p a key i for a key i*

*<proof>*

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

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

**definition** [*simp*]:  $\text{right } h \ a \ j \ i = \text{take } (j - \text{Ref.get } h \ i) \ (\text{drop } (\text{Ref.get } h \ i + 1) \ (\text{Array.get } h \ a))$

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

**lemma** *effect-shiftr*:

**assumes**  $\text{Ref.get } h \ i = j$  (**is**  $?i \ h = -$ )

**and**  $j < \text{Array.length } h \ a$

**and** *sorted* ( $\text{take } j \ (\text{Array.get } h \ a)$ )

**and** *effect* (*while* (*shiftr-p a key i*) (*shiftr-f a i*)) *h h' u*

**shows**  $\text{Array.length } h \ a = \text{Array.length } h' \ a \wedge$

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

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

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

$\text{ini } h \ a \ j = \text{both } h' \ a \ j \ i \wedge$

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

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

*<proof>*

**lemma** *sorted-take-nth*:

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

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

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

*<proof>*

**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**  $Array.length\ h\ a = Array.length\ h'\ a \wedge$   
 $sorted\ (take\ l\ (Array.get\ h'\ a)) \wedge$   
 $mset\ (Array.get\ h\ a) = mset\ (Array.get\ h'\ a)$   
 $\langle proof \rangle$

**lemma** *effect-insertion-sort*:

**assumes**  $effect\ (insertion-sort\ a)\ h\ h'\ u$   
**shows**  $mset\ (Array.get\ h\ a) = mset\ (Array.get\ h'\ a) \wedge sorted\ (Array.get\ h'\ a)$   
 $\langle proof \rangle$

## 2.3 Total Correctness

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

**assumes**  $Ref.get\ h\ i < Array.length\ h\ a$   
**shows**  $success\ (shiftr-f\ a\ i)\ h$   
 $\langle proof \rangle$

**lemma** *success-shiftr*:

**assumes**  $Ref.get\ h\ i < Array.length\ h\ a$   
**shows**  $success\ (while\ (shiftr-p\ a\ key\ i)\ (shiftr-f\ a\ i))\ h$   
 $\langle proof \rangle$

**lemma** *effect-shiftr-index*:

**assumes**  $effect\ (shiftr\ a\ key\ i)\ h\ h'\ u$   
**shows**  $Ref.get\ h'\ i \leq Ref.get\ h\ i$   
 $\langle proof \rangle$

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

**assumes**  $effect\ (shiftr\ a\ key\ i)\ h\ h'\ u$   
**shows**  $Array.length\ h'\ a = Array.length\ h\ a$   
 $\langle proof \rangle$

**lemma** *success-insert-elt*:

**assumes**  $k < Array.length\ h\ a$   
**shows**  $success\ (insert-elt\ a\ k)\ h$   
 $\langle proof \rangle$

**lemma** *for-insert-elt-correct*:

**assumes**  $l \leq Array.length\ h\ a$   
**and**  $1 \leq l$   
**shows**  $\exists h'. effect\ (for\ [1..<l]\ (insert-elt\ a))\ h\ h'\ () \wedge$   
 $Array.length\ h\ a = Array.length\ h'\ a \wedge$   
 $sorted\ (take\ l\ (Array.get\ h'\ a)) \wedge$   
 $mset\ (Array.get\ h\ a) = mset\ (Array.get\ h'\ a)$   
 $\langle proof \rangle$

**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⟩

export-code insertion-sort in Haskell

end
```