Imperative Insertion Sort

Christian Sternagel

March 17, 2025

Contents

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

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

definition *cond p h* ⟷ *fst* (*the* (*execute p h*))

A locale that restricts to read-only loop conditions.

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

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

lemma *ro-cond*: *ro-cond p*
⟨*proof*⟩

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

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

lemma
success-while-cond: $success\ p\ h \Longrightarrow cond\ p\ h \Longrightarrow effect\ f\ h\ h'\ r \Longrightarrow success\ (while\ p\ f)\ h' \Longrightarrow$
$success\ (while\ p\ f)\ h$ **and**
success-while-not-cond: $success\ p\ h \Longrightarrow \neg\ cond\ p\ h \Longrightarrow success\ (while\ p\ f)\ h$
⟨*proof*⟩

lemma *success-cond-effect*:
$success\ p\ h \Longrightarrow cond\ p\ h \Longrightarrow effect\ p\ h\ h\ True$
⟨*proof*⟩

lemma *success-not-cond-effect*:
$success\ p\ h \Longrightarrow \neg\ cond\ p\ h \Longrightarrow effect\ p\ h\ h\ False$
⟨*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 $\bigwedge h\ h'\ u.\ I\ h \Longrightarrow success\ p\ h \Longrightarrow cond\ p\ h \Longrightarrow effect\ f\ h\ h'\ u \Longrightarrow I\ h'$
shows $I\ h'$
⟨*proof*⟩

lemma *effect-success-conv*:
$(\exists h'.\ effect\ c\ h\ h'\ ()\ \wedge\ I\ h') \longleftrightarrow success\ c\ h\ \wedge\ I\ (snd\ (the\ (execute\ c\ h)))$
⟨*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*: $\bigwedge h.\ I\ h \Longrightarrow success\ p\ h$ — the loop-condition terminates
and *success-f*: $\bigwedge h.\ I\ h \Longrightarrow success\ p\ h \Longrightarrow cond\ p\ h \Longrightarrow success\ f\ h$ — the loop-body terminates
and $I\ h$ — the invariant holds before the loop is entered
and *step*: $\bigwedge h\ h'\ r.\ I\ h \Longrightarrow success\ p\ h \Longrightarrow cond\ p\ h \Longrightarrow effect\ f\ h\ h'\ r \Longrightarrow (h', h) \in R \wedge I\ h'$
— the invariant is preserved by iterating the loop
shows $\exists h'.\ effect\ (while\ p\ f)\ h\ h'\ ()\ \wedge\ I\ h'$
⟨*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 $i \le j$
and $effect\ (for\ [i\ ..{<}\ j]\ f)\ h\ h'\ u$
and $I\ i\ h$
and $\bigwedge k\ h\ h'\ r.\ i \le k \Longrightarrow k < j \Longrightarrow I\ k\ h \Longrightarrow effect\ (f\ k)\ h\ h'\ r \Longrightarrow I\ (Suc\ k)\ h'$

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 $i \le j$
and $\bigwedge k\ h.\ I\ k\ h \Longrightarrow i \le k \Longrightarrow k < j \Longrightarrow success\ (f\ k)\ h$
and $I\ i\ h$
and $\bigwedge k\ h\ h'\ r.\ I\ k\ h \Longrightarrow i \le k \Longrightarrow k < j \Longrightarrow effect\ (f\ k)\ h\ h'\ r \Longrightarrow I\ (Suc\ k)\ h'$
shows $\exists h'.\ effect\ (for\ [i\ ..< j]\ f)\ h\ h'\ ()\ \wedge\ I\ j\ h'$ (**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* ⇒ *nat* ⇒ *'a* ⇒ *'a array Heap* (‹(-.'(-') ←/ -)› [*1000*, *0*, *13*] *14*)
where
$a.(i) \leftarrow x \equiv Array.upd\ i\ x\ a$

abbreviation *array-nth* :: *'a::heap array* ⇒ *nat* ⇒ *'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* {

$l \leftarrow Array.len\ a;$
$for\ [1\ ..< l]\ (\lambda j.\ do\ \{$
— Insert $a[j]$ into the sorted subarray $a[1\ ..\ j - 1]$.
$key \leftarrow a.(j);$
$i \leftarrow ref\ j;$
$while\ (do\ \{$
$i' \leftarrow !\ i;$
$if\ i' > 0\ then\ do\ \{x \leftarrow a.(i' - 1);\ return\ (x > key)\}$
$else\ return\ False\})$
$(do\ \{$
$i' \leftarrow !\ i;$
$x \leftarrow a.(i' - 1);$
$a.(i') \leftarrow x;$
$i := i' - 1$
$\});$
$i' \leftarrow !\ i;$
$a.(i') \leftarrow key$
$\})$
$\}$

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

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

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

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

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

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

lemma $insertion\text{-}sort\text{-}alt\text{-}def$:
$insertion\text{-}sort\ a = (Array.len\ a \gg= sort\text{-}upto\ a)$
$\langle proof \rangle$

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* ∧
    Array.get h′ a = *list-update* (*Array.get h a*) (*Ref.get h i*) (*Array.get h a* ! (*Ref.get h i* − *1*))
  ⟨*proof*⟩

lemma *success-shiftr-p*:
  Ref.get h i < *Array.length h a* ⟹ *success* (*shiftr-p a key i*) *h*
  ⟨*proof*⟩

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

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* ∧
    ?i h′ ≤ *j* ∧
    mset (*list-update* (*Array.get h a*) *j key*) =
      mset (*list-update* (*Array.get h′ a*) (*?i h′*) *key*) ∧
    ini h a j = *both h′ a j i* ∧
    sorted (*both h′ a j i*) ∧
    (∀ *x* ∈ *set* (*right h′ a j i*). *x* > *key*)
⟨*proof*⟩

lemma *sorted-take-nth*:
  assumes *0* < *i* **and** *i* < *length xs* **and** *xs* ! (*i* − *1*) ≤ *y*
    and *sorted* (*take i xs*)
  shows ∀ *x* ∈ *set* (*take i xs*). *x* ≤ *y*
⟨*proof*⟩

lemma *effect-for-insert-elt*:
  assumes *l* ≤ *Array.length h a*
    and *1* ≤ *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 $effect\ (insertion\text{-}sort\ a)\ h\ h'\ u$
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 $success\ (shiftr\text{-}f\ a\ i)\ h$
$\langle proof \rangle$

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

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 $success\ (insert\text{-}elt\ a\ k)\ h$
$\langle proof \rangle$

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

lemma *insertion-sort-correct*:
$\exists h'.\ effect\ (insertion\text{-}sort\ a)\ h\ h'\ u\ \wedge$
$mset\ (Array.get\ h\ a) = mset\ (Array.get\ h'\ a)\ \wedge$

sorted (*Array.get* h' *a*)
⟨*proof*⟩

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

end