Imperative Insertion Sort

Christian Sternagel

May 23, 2019

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

partial-function (*heap*) *while* :: *bool Heap* ⇒ *'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 $\Longrightarrow$ ¬ *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 $\Longrightarrow$ ¬ *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* $\Longrightarrow$ ¬ *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 $\Longrightarrow$ ¬ *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′* ∧ ¬ *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*:
(∃ *h'*. *effect c h h'* () ∧ *I h'*) ⟷ *success c h* ∧ *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*: ⋀*h*. *I h* ⟹ *success p h* — the loop-condition terminates
and *success-f*: ⋀*h*. *I h* ⟹ *success p h* ⟹ *cond p h* ⟹ *success f h* — the loop-body terminates
and *I h* — the invariant holds before the loop is entered
and *step*: ⋀*h h' r*. *I h* ⟹ *success p h* ⟹ *cond p h* ⟹ *effect f h h' r* ⟹ (*h'*, *h*) ∈ *R* ∧ *I h'*
— the invariant is preserved by iterating the loop
shows ∃ *h'*. *effect* (*while p f*) *h h'* () ∧ *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* ⇒ (*'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 $I\ j\ h'$
⟨*proof*⟩

A rule for proving total correctness of for loops.

lemma *for-induct* [*consumes 1*, *case-names succeed base step*]:
assumes $i \leq j$
and $\bigwedge k\ h.\ I\ k\ h \Longrightarrow i \leq k \Longrightarrow k < j \Longrightarrow \mathit{success}\ (f\ k)\ h$
and $I\ i\ h$
and $\bigwedge k\ h\ h'\ r.\ I\ k\ h \Longrightarrow i \leq k \Longrightarrow k < j \Longrightarrow \mathit{effect}\ (f\ k)\ h\ h'\ r \Longrightarrow I\ (\mathit{Suc}\ k)\ h'$
shows $\exists h'.\ \mathit{effect}\ (\mathit{for}\ [i\ ..<\ j]\ f)\ h\ h'\ () \wedge I\ j\ h'$ (**is** *?P i h*)
⟨*proof*⟩

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

abbreviation *array-nth* :: *'a*::*heap array* ⇒ *nat* ⇒ *'a Heap* (-.*'*(-*'*) [*1000*, *0*] *14*)
where
$a.(i) \equiv \mathit{Array.nth}\ a\ i$

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 *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* ∧
sorted (*take l* (*Array.get h′ a*)) ∧
mset (*Array.get h a*) = *mset* (*Array.get h′ a*)
⟨*proof*⟩

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*)
⟨*proof*⟩

2.3 Total Correctness

lemma *success-shiftr-f*:
assumes *Ref.get h i* < *Array.length h a*
shows *success* (*shiftr-f a i*) *h*
⟨*proof*⟩

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*⟩

lemma *effect-shiftr-index*:
assumes *effect* (*shiftr a key i*) *h h′ u*
shows *Ref.get h′ i* ≤ *Ref.get h i*
⟨*proof*⟩

lemma *effect-shiftr-length*:
assumes *effect* (*shiftr a key i*) *h h′ u*
shows *Array.length h′ a* = *Array.length h a*
⟨*proof*⟩

lemma *success-insert-elt*:
assumes *k* < *Array.length h a*
shows *success* (*insert-elt a k*) *h*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *insertion-sort-correct*:
∃ *h′*. *effect* (*insertion-sort a*) *h h′ u* ∧
mset (*Array.get h a*) = *mset* (*Array.get h′ a*) ∧

sorted (*Array.get* h' *a*)
⟨*proof*⟩

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

end