IMP in HOLCF

Tobias Nipkow and Robert Sandner

November 22, 2007

Contents

1 Syntax of Commands

theory *Com* **imports** *Main* **begin**

typedecl *loc*
— an unspecified (arbitrary) type of locations (adresses/names) for variables

types
```
val   = nat — or anything else, nat used in examples
state = "loc ⇒ val"
aexp  = "state ⇒ val"
bexp  = "state ⇒ bool"
```
— arithmetic and boolean expressions are not modelled explicitly here,
— they are just functions on states

datatype
```
com = SKIP
    | Assign loc aexp        ("_ :== _ " 60)
    | Semi   com com         ("_; _"  [60, 60] 10)
    | Cond   bexp com com    ("IF _ THEN _ ELSE _"  60)
```

```
      | While  bexp com         ("WHILE _ DO _"  60)
```

syntax *(latex)*

```
  SKIP :: com   ("skip")
  Cond :: "bexp ⇒ com ⇒ com ⇒ com"  ("if _ then _ else _"  60)
  While :: "bexp ⇒ com ⇒ com" ("while _ do _"  60)
```

end

2 Natural Semantics of Commands

theory *Natural* **imports** *Com* **begin**

2.1 Execution of commands

We write ⟨c,s⟩ $\longrightarrow_c$ s' for *Statement* c*, started in state* s*, terminates in state* s'. Formally, ⟨c,s⟩ $\longrightarrow_c$ s' is just another form of saying *the tuple* (c,s,s') *is part of the relation* evalc:

constdefs

```
  update :: "('a ⇒ 'b) ⇒ 'a ⇒ 'b ⇒ ('a ⇒ 'b)" ("_/[_ ::= /_]" [900,0,0] 900)
  "update == fun_upd"
```

syntax *(xsymbols)*

```
  update :: "('a ⇒ 'b) ⇒ 'a ⇒ 'b ⇒ ('a ⇒ 'b)" ("_/[_ ↦ /_]" [900,0,0] 900)
```

The big-step execution relation evalc is defined inductively:

inductive

```
  evalc :: "[com,state,state] ⇒ bool" ("⟨_,_⟩/ ⟶c _" [0,0,60] 60)
```

where

```
  Skip:    "⟨skip,s⟩ ⟶c s"
| Assign:  "⟨x :== a,s⟩ ⟶c s[x↦a s]"

| Semi:    "⟨c0,s⟩ ⟶c s'' ⟹ ⟨c1,s''⟩ ⟶c s' ⟹ ⟨c0; c1, s⟩ ⟶c s'"

| IfTrue:  "b s ⟹ ⟨c0,s⟩ ⟶c s' ⟹ ⟨if b then c0 else c1, s⟩ ⟶c s'"
| IfFalse: "¬b s ⟹ ⟨c1,s⟩ ⟶c s' ⟹ ⟨if b then c0 else c1, s⟩ ⟶c s'"

| WhileFalse: "¬b s ⟹ ⟨while b do c,s⟩ ⟶c s"
| WhileTrue:  "b s ⟹ ⟨c,s⟩ ⟶c s'' ⟹ ⟨while b do c, s''⟩ ⟶c s'
              ⟹ ⟨while b do c, s⟩ ⟶c s'"
```

lemmas *evalc.intros [intro]* — use those rules in automatic proofs

The induction principle induced by this definition looks like this:

```
⟦⟨x1,x2⟩ ⟶c x3; ⋀s. P skip s s; ⋀x a s. P (x :== a ) s (s[x ↦ a s]);
 ⋀c0 s s'' c1 s'.
```

$\llbracket\langle c0,s\rangle \longrightarrow_c s''$; $P\ c0\ s\ s''$; $\langle c1,s''\rangle \longrightarrow_c s'$; $P\ c1\ s''\ s'\rrbracket$
$\Longrightarrow P\ (c0;\ c1)\ s\ s'$;
$\bigwedge b\ s\ c0\ s'\ c1.\ \llbracket b\ s;\ \langle c0,s\rangle \longrightarrow_c s';\ P\ c0\ s\ s'\rrbracket \Longrightarrow P$ (if b then $c0$ else $c1$) $s\ s'$;
$\bigwedge b\ s\ c1\ s'\ c0.\ \llbracket\neg\ b\ s;\ \langle c1,s\rangle \longrightarrow_c s';\ P\ c1\ s\ s'\rrbracket \Longrightarrow P$ (if b then $c0$ else $c1$) s s';
$\bigwedge b\ s\ c.\ \neg\ b\ s \Longrightarrow P$ (while b do c) $s\ s$;
$\bigwedge b\ s\ c\ s''\ s'.$
$\llbracket b\ s;\ \langle c,s\rangle \longrightarrow_c s'';\ P\ c\ s\ s'';$ $\langle$while b do $c,s''\rangle \longrightarrow_c s'$;
P (while b do c) $s''\ s'\rrbracket$
$\Longrightarrow P$ (while b do c) $s\ s'\rrbracket$
$\Longrightarrow P\ x1\ x2\ x3$

($\bigwedge$ and $\Longrightarrow$ are Isabelle's meta symbols for $\forall$ and $\longrightarrow$)

The rules of *evalc* are syntax directed, i.e. for each syntactic category there is always only one rule applicable. That means we can use the rules in both directions. The proofs for this are all the same: one direction is trivial, the other one is shown by using the *evalc* rules backwards:

lemma *skip:*
"$\langle$skip,$s\rangle \longrightarrow_c s' = (s' = s)$"
⟨*proof*⟩

lemma *assign:*
"$\langle x$:== $a,s\rangle \longrightarrow_c s' = (s' = s[x\mapsto a\ s])$"
⟨*proof*⟩

lemma *semi:*
"$\langle c0;\ c1,\ s\rangle \longrightarrow_c s' = (\exists s''.\ \langle c0,s\rangle \longrightarrow_c s'' \wedge \langle c1,s''\rangle \longrightarrow_c s')$"
⟨*proof*⟩

lemma *ifTrue:*
"$b\ s \Longrightarrow \langle$if b then $c0$ else $c1,\ s\rangle \longrightarrow_c s' = \langle c0,s\rangle \longrightarrow_c s'$"
⟨*proof*⟩

lemma *ifFalse:*
"$\neg b\ s \Longrightarrow \langle$if b then $c0$ else $c1,\ s\rangle \longrightarrow_c s' = \langle c1,s\rangle \longrightarrow_c s'$"
⟨*proof*⟩

lemma *whileFalse:*
"$\neg\ b\ s \Longrightarrow \langle$while b do $c,s\rangle \longrightarrow_c s' = (s' = s)$"
⟨*proof*⟩

lemma *whileTrue:*
"$b\ s \Longrightarrow$
$\langle$while b do $c,\ s\rangle \longrightarrow_c s'$ =
$(\exists s''.\ \langle c,s\rangle \longrightarrow_c s'' \wedge \langle$while b do $c,\ s''\rangle \longrightarrow_c s')$"
⟨*proof*⟩

Again, Isabelle may use these rules in automatic proofs:

lemmas *evalc_cases [simp] = skip assign ifTrue ifFalse whileFalse semi whileTrue*

2.2 Equivalence of statements

We call two statements *c* and *c'* equivalent wrt. the big-step semantics when *c started in s terminates in s' iff c' started in the same s also terminates in the same s'*. Formally:

```
constdefs
  equiv_c :: "com ⇒ com ⇒ bool" ("_ ∼ _")
  "c ∼ c' ≡ ∀s s'. ⟨c, s⟩ →c s' = ⟨c', s⟩ →c s'"
```

Proof rules telling Isabelle to unfold the definition if there is something to be proved about equivalent statements:

```
lemma equivI [intro!]:
  "(⋀s s'. ⟨c, s⟩ →c s' = ⟨c', s⟩ →c s') ⟹ c ∼ c'"
  ⟨proof⟩

lemma equivD1:
  "c ∼ c' ⟹ ⟨c, s⟩ →c s' ⟹ ⟨c', s⟩ →c s'"
  ⟨proof⟩

lemma equivD2:
  "c ∼ c' ⟹ ⟨c', s⟩ →c s' ⟹ ⟨c, s⟩ →c s'"
  ⟨proof⟩
```

As an example, we show that loop unfolding is an equivalence transformation on programs:

```
lemma unfold_while:
  "(while b do c) ∼ (if b then c; while b do c else skip)" (is "?w ∼ ?if")
⟨proof⟩
```

2.3 Execution is deterministic

The following proof presents all the details:

```
theorem com_det:
  assumes "⟨c,s⟩ →c t" and "⟨c,s⟩ →c u"
  shows "u = t"
  ⟨proof⟩
```

This is the proof as you might present it in a lecture. The remaining cases are simple enough to be proved automatically:

```
theorem
  assumes "⟨c,s⟩ →c t" and "⟨c,s⟩ →c u"
  shows "u = t"
  ⟨proof⟩

end
```

3 Denotational Semantics of Commands in HOLCF

theory *Denotational* **imports** *HOLCF Natural* **begin**

3.1 Definition

definition

```
  dlift :: "(('a::type) discr -> 'b::pcpo) => ('a lift -> 'b)" where
  "dlift f = (LAM x. case x of UU => UU | Def y => f·(Discr y))"
```

consts `D :: "com => state discr -> state lift"`

primrec

```
  "D(skip) = (LAM s. Def(undiscr s))"
  "D(X :== a) = (LAM s. Def((undiscr s)[X ↦ a(undiscr s)]))"
  "D(c0 ; c1) = (dlift(D c1) oo (D c0))"
  "D(if b then c1 else c2) =
        (LAM s. if b (undiscr s) then (D c1)·s else (D c2)·s)"
  "D(while b do c) =
        fix·(LAM w s. if b (undiscr s) then (dlift w)·((D c)·s)
                      else Def(undiscr s))"
```

3.2 Equivalence of Denotational Semantics in HOLCF and Evaluation Semantics in HOL

lemma `dlift_Def [simp]: "dlift f·(Def x) = f·(Discr x)"`
  ⟨*proof*⟩

lemma `cont_dlift [iff]: "cont (%f. dlift f)"`
  ⟨*proof*⟩

lemma `dlift_is_Def [simp]:`
  `"(dlift f·l = Def y) = (∃x. l = Def x ∧ f·(Discr x) = Def y)"`
  ⟨*proof*⟩

lemma `eval_implies_D: "⟨c,s⟩` $\longrightarrow_c$ `t ==> D c·(Discr s) = (Def t)"`
  ⟨*proof*⟩

lemma `D_implies_eval: "!s t. D c·(Discr s) = (Def t) --> ⟨c,s⟩` $\longrightarrow_c$ `t"`
  ⟨*proof*⟩

theorem `D_is_eval: "(D c·(Discr s) = (Def t)) = (⟨c,s⟩` $\longrightarrow_c$ `t)"`
  ⟨*proof*⟩

end

4 Correctness of Hoare by Fixpoint Reasoning

theory *HoareEx* **imports** *Denotational* **begin**

An example from the HOLCF paper by Müller, Nipkow, Oheimb, Slotosch [1]. It demonstrates fixpoint reasoning by showing the correctness of the Hoare rule for while-loops.

```
types assn = "state => bool"

definition
  hoare_valid :: "[assn, com, assn] => bool"  ("|= {(1_)}/ (_)/ {(1_)}" 50) where
  "|= {A} c {B} = (∀s t. A s ∧ D c $(Discr s) = Def t --> B t)"

lemma WHILE_rule_sound:
    "|= {A} c {A} ==> |= {A} while b do c {λs. A s ∧ ¬ b s}"
  ⟨proof⟩

end
```

References

[1] O. Müller, T. Nipkow, D. v. Oheimb, and O. Slotosch. HOLCF = HOL + LCF. *J. Functional Programming*, 9:191–223, 1999.