

IMP in HOLCF

Tobias Nipkow and Robert Sandner

June 8, 2008

Contents

1	Syntax of Commands	1
2	Natural Semantics of Commands	2
2.1	Execution of commands	2
2.2	Equivalence of statements	4
2.3	Execution is deterministic	4
3	Denotational Semantics of Commands in HOLCF	5
3.1	Definition	5
3.2	Equivalence of Denotational Semantics in HOLCF and Evaluation Semantics in HOL	5
4	Correctness of Hoare by Fixpoint Reasoning	6

1 Syntax of Commands

`theory Com imports Main begin`

`typeddecl loc`

— an unspecified (arbitrary) type of locations (addresses/names) for variables

`types`

`val = nat` — or anything else, `nat` used in examples

`state = "loc \Rightarrow val"`

`aexp = "state \Rightarrow val"`

`bexp = "state \Rightarrow 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  $\Rightarrow$  com  $\Rightarrow$  com  $\Rightarrow$  com" ("if _ then _ else _" 60)
  While :: "bexp  $\Rightarrow$  com  $\Rightarrow$  com" ("while _ do _" 60)

end

```

2 Natural Semantics of Commands

theory Natural imports Com begin

2.1 Execution of commands

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

```

constdefs
  update :: "('a  $\Rightarrow$  'b)  $\Rightarrow$  'a  $\Rightarrow$  'b  $\Rightarrow$  ('a  $\Rightarrow$  'b)" ("_/_ ::= /_" [900,0,0] 900)
  "update == fun_upd"

```

```

syntax (xsymbols)
  update :: "('a  $\Rightarrow$  'b)  $\Rightarrow$  'a  $\Rightarrow$  'b  $\Rightarrow$  ('a  $\Rightarrow$  'b)" ("_/_  $\mapsto$  /_" [900,0,0] 900)

```

The big-step execution relation evalc is defined inductively:

```

inductive
  evalc :: "[com, state, state]  $\Rightarrow$  bool" ("<_,_>/  $\longrightarrow_c$  _" [0,0,60] 60)
where
  Skip:      "<skip, s>  $\longrightarrow_c$  s"
  | Assign:  "<x ::= a, s>  $\longrightarrow_c$  s[x  $\mapsto$  a s]"

  | Semi:    "<c0, s>  $\longrightarrow_c$  s''  $\Longrightarrow$  <c1, s''>  $\longrightarrow_c$  s'  $\Longrightarrow$  <c0; c1, s>  $\longrightarrow_c$  s'"

  | IfTrue:  "b s  $\Longrightarrow$  <c0, s>  $\longrightarrow_c$  s'  $\Longrightarrow$  <if b then c0 else c1, s>  $\longrightarrow_c$  s'"
  | IfFalse: " $\neg$ b s  $\Longrightarrow$  <c1, s>  $\longrightarrow_c$  s'  $\Longrightarrow$  <if b then c0 else c1, s>  $\longrightarrow_c$  s'"

  | WhileFalse: " $\neg$ b s  $\Longrightarrow$  <while b do c, s>  $\longrightarrow_c$  s"
  | WhileTrue:  "b s  $\Longrightarrow$  <c, s>  $\longrightarrow_c$  s''  $\Longrightarrow$  <while b do c, s''>  $\longrightarrow_c$  s'
                  $\Longrightarrow$  <while b do c, s>  $\longrightarrow_c$  s'"

```

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

The induction principle induced by this definition looks like this:

```

[[<x1, x2>  $\longrightarrow_c$  x3;  $\bigwedge s. P \text{ skip } s \ s$ ;  $\bigwedge x \ a \ s. P \ (x ::= a) \ s \ (s[x \mapsto a \ s])$ ;
 $\bigwedge c0 \ s \ s'' \ c1 \ s'.$ 
```

$$\begin{aligned}
& \llbracket \langle c0, s \rangle \longrightarrow_c s''; P \ c0 \ s \ s''; \langle c1, s'' \rangle \longrightarrow_c s'; P \ c1 \ s'' \ s' \rrbracket \\
& \implies P \ (c0; c1) \ s \ s'; \\
& \bigwedge b \ s \ c0 \ s' \ c1. \llbracket b \ s; \langle c0, s \rangle \longrightarrow_c s'; P \ c0 \ s \ s' \rrbracket \implies P \ (\text{if } b \text{ then } c0 \text{ else } c1) \ s \ s'; \\
& \bigwedge b \ s \ c1 \ s' \ c0. \llbracket \neg b \ s; \langle c1, s \rangle \longrightarrow_c s'; P \ c1 \ s \ s' \rrbracket \implies P \ (\text{if } b \text{ then } c0 \text{ else } c1) \ s \\
& s'; \\
& \bigwedge b \ s \ c. \neg b \ s \implies P \ (\text{while } b \text{ do } c) \ s \ s; \\
& \bigwedge b \ s \ c \ s'' \ s'. \\
& \quad \llbracket b \ s; \langle c, s \rangle \longrightarrow_c s''; P \ c \ s \ s''; \langle \text{while } b \text{ do } c, s'' \rangle \longrightarrow_c s'; \\
& \quad P \ (\text{while } b \text{ do } c) \ s'' \ s' \rrbracket \\
& \implies P \ (\text{while } b \text{ do } c) \ s \ s' \\
& \implies P \ x1 \ x2 \ x3
\end{aligned}$$

(\bigwedge and \implies 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:

" $\langle \text{skip}, s \rangle \longrightarrow_c s' = (s' = s)$ "
 $\langle \text{proof} \rangle$

lemma assign:

" $\langle x ::= a, s \rangle \longrightarrow_c s' = (s' = s[x \mapsto a])$ "
 $\langle \text{proof} \rangle$

lemma semi:

" $\langle c0; c1, s \rangle \longrightarrow_c s' = (\exists s''. \langle c0, s \rangle \longrightarrow_c s'' \wedge \langle c1, s'' \rangle \longrightarrow_c s')$ "
 $\langle \text{proof} \rangle$

lemma ifTrue:

" $b \ s \implies \langle \text{if } b \text{ then } c0 \text{ else } c1, s \rangle \longrightarrow_c s' = \langle c0, s \rangle \longrightarrow_c s'$ "
 $\langle \text{proof} \rangle$

lemma ifFalse:

" $\neg b \ s \implies \langle \text{if } b \text{ then } c0 \text{ else } c1, s \rangle \longrightarrow_c s' = \langle c1, s \rangle \longrightarrow_c s'$ "
 $\langle \text{proof} \rangle$

lemma whileFalse:

" $\neg b \ s \implies \langle \text{while } b \text{ do } c, s \rangle \longrightarrow_c s' = (s' = s)$ "
 $\langle \text{proof} \rangle$

lemma whileTrue:

" $b \ s \implies$
 $\langle \text{while } b \text{ do } c, s \rangle \longrightarrow_c s' =$
 $(\exists s''. \langle c, s \rangle \longrightarrow_c s'' \wedge \langle \text{while } b \text{ do } c, s'' \rangle \longrightarrow_c s')$ "
 $\langle \text{proof} \rangle$

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  $\Rightarrow$  com  $\Rightarrow$  bool" ("_  $\sim$  _")
" $c \sim c' \equiv \forall s s'. \langle c, s \rangle \longrightarrow_c s' = \langle c', s \rangle \longrightarrow_c s'$ "
```

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

lemma equivI [intro!]:

```
"( $\bigwedge s s'. \langle c, s \rangle \longrightarrow_c s' = \langle c', s \rangle \longrightarrow_c s'$ )  $\implies c \sim c'$ "
<proof>
```

lemma equivD1:

```
" $c \sim c' \implies \langle c, s \rangle \longrightarrow_c s' \implies \langle c', s \rangle \longrightarrow_c s'$ "
<proof>
```

lemma equivD2:

```
" $c \sim c' \implies \langle c', s \rangle \longrightarrow_c s' \implies \langle c, s \rangle \longrightarrow_c s'$ "
<proof>
```

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

lemma unfold_while:

```
"(while b do c)  $\sim$  (if b then c; while b do c else skip)" (is "?w  $\sim$  ?if")
<proof>
```

2.3 Execution is deterministic

The following proof presents all the details:

theorem com_det:

```
assumes "<math>\langle c, s \rangle \longrightarrow_c t</math>" and "<math>\langle c, s \rangle \longrightarrow_c u</math>"
shows "<math>u = t</math>"
<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 "<math>\langle c, s \rangle \longrightarrow_c t</math>" and "<math>\langle c, s \rangle \longrightarrow_c u</math>"
shows "<math>u = t</math>"
<proof>
```

end

3 Denotational Semantics of Commands in HOLCF

theory Denotational imports HOLCF "../../HOL/IMP/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⟩ ⟶c t ==> D c·(Discr s) = (Def t)"
  <proof>
```

```
lemma D_implies_eval: "!s t. D c·(Discr s) = (Def t) --> ⟨c,s⟩ ⟶c t"
  <proof>
```

```
theorem D_is_eval: "(D c·(Discr s) = (Def t)) = (⟨c,s⟩ ⟶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} = ($\forall s\ t. A\ s \wedge D\ c\ \$ (Discr\ s) = Def\ t \longrightarrow B\ t$)"

lemma WHILE_rule_sound:

"|= {A} c {A} ==> |= {A} while b do c { $\lambda s. A\ s \wedge \neg b\ s$ }"
<proof>

end

References

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