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'.
```

```
    ⟦⟨c0,s⟩ ⟶c s''; P c0 s s''; ⟨c1,s''⟩ ⟶c s'; P c1 s'' s'⟧
    ⟹ P (c0; c1) s s';
 ⋀b s c0 s' c1. ⟦b s; ⟨c0,s⟩ ⟶c s'; P c0 s s'⟧ ⟹ P (if b then c0 else c1) s s';
 ⋀b s c1 s' c0. ⟦¬ b s; ⟨c1,s⟩ ⟶c s'; P c1 s s'⟧ ⟹ P (if b then c0 else c1) s
s';
 ⋀b s c. ¬ b s ⟹ P (while b do c) s s;
 ⋀b s c s'' s'.
    ⟦b s; ⟨c,s⟩ ⟶c s''; P c s s''; ⟨while b do c,s''⟩ ⟶c s';
     P (while b do c) s'' s'⟧
    ⟹ P (while b do c) s s'⟧
⟹ P x1 x2 x3
```

(⋀ and ⟹ are Isabelle's meta symbols for ∀ and ⟶)

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:
  "⟨skip,s⟩ ⟶c s' = (s' = s)"
  by (rule, erule evalc.cases) auto

lemma assign:
  "⟨x :== a,s⟩ ⟶c s' = (s' = s[x↦a s])"
  by (rule, erule evalc.cases) auto

lemma semi:
  "⟨c0; c1, s⟩ ⟶c s' = (∃s''. ⟨c0,s⟩ ⟶c s'' ∧ ⟨c1,s''⟩ ⟶c s')"
  by (rule, erule evalc.cases) auto

lemma ifTrue:
  "b s ⟹ ⟨if b then c0 else c1, s⟩ ⟶c s' = ⟨c0,s⟩ ⟶c s'"
  by (rule, erule evalc.cases) auto

lemma ifFalse:
  "¬b s ⟹ ⟨if b then c0 else c1, s⟩ ⟶c s' = ⟨c1,s⟩ ⟶c s'"
  by (rule, erule evalc.cases) auto

lemma whileFalse:
  "¬ b s ⟹ ⟨while b do c,s⟩ ⟶c s' = (s' = s)"
  by (rule, erule evalc.cases) auto

lemma whileTrue:
  "b s ⟹
  ⟨while b do c, s⟩ ⟶c s' =
  (∃s''. ⟨c,s⟩ ⟶c s'' ∧ ⟨while b do c, s''⟩ ⟶c s')"
  by (rule, erule evalc.cases) auto
```

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'"
  by (unfold equiv_c_def) blast

lemma equivD1:
  "c ∼ c' ⟹ ⟨c, s⟩ ⟶c s' ⟹ ⟨c', s⟩ ⟶c s'"
  by (unfold equiv_c_def) blast

lemma equivD2:
  "c ∼ c' ⟹ ⟨c', s⟩ ⟶c s' ⟹ ⟨c, s⟩ ⟶c s'"
  by (unfold equiv_c_def) blast
```

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 -
  — to show the equivalence, we look at the derivation tree for
  — each side and from that construct a derivation tree for the other side
  { fix s s' assume w: "⟨?w, s⟩ ⟶c s'"
    — as a first thing we note that, if b is False in state s,
    — then both statements do nothing:
    hence "¬b s ⟹ s = s'" by simp
    hence "¬b s ⟹ ⟨?if, s⟩ ⟶c s'" by simp
    moreover
    — on the other hand, if b is True in state s,
    — then only the WhileTrue rule can have been used to derive ⟨?w, s⟩ ⟶c s'
    { assume b: "b s"
      with w obtain s'' where
        "⟨c, s⟩ ⟶c s''" and "⟨?w, s''⟩ ⟶c s'" by (cases set: evalc) auto
      — now we can build a derivation tree for the if
      — first, the body of the True-branch:
      hence "⟨c; ?w, s⟩ ⟶c s'" by (rule Semi)
      — then the whole if
      with b have "⟨?if, s⟩ ⟶c s'" by (rule IfTrue)
    }
    ultimately
    — both cases together give us what we want:
    have "⟨?if, s⟩ ⟶c s'" by blast
  }
  moreover
  — now the other direction:
  { fix s s' assume "if": "⟨?if, s⟩ ⟶c s'"
    — again, if b is False in state s, then the False-branch
    — of the if is executed, and both statements do nothing:
    hence "¬b s ⟹ s = s'" by simp
    hence "¬b s ⟹ ⟨?w, s⟩ ⟶c s'" by simp
    moreover
    — on the other hand, if b is True in state s,
    — then this time only the IfTrue rule can have be used
    { assume b: "b s"
      with "if" have "⟨c; ?w, s⟩ ⟶c s'" by (cases set: evalc) auto
      — and for this, only the Semi-rule is applicable:
      then obtain s'' where
        "⟨c, s⟩ ⟶c s''" and "⟨?w, s''⟩ ⟶c s'" by (cases set: evalc) auto
      — with this information, we can build a derivation tree for the while
      with b
      have "⟨?w, s⟩ ⟶c s'" by (rule WhileTrue)
    }
    ultimately
    — both cases together again give us what we want:
    have "⟨?w, s⟩ ⟶c s'" by blast
  }
  ultimately
  show ?thesis by blast
qed
```

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"
  using prems
proof (induct arbitrary: u set: evalc)
  fix s u assume "⟨skip,s⟩ ⟶c u"
  thus "u = s" by simp
next
  fix a s x u assume "⟨x :== a,s⟩ ⟶c u"
  thus "u = s[x ↦ a s]" by simp
next
  fix c0 c1 s s1 s2 u
  assume IH0: "⋀u. ⟨c0,s⟩ ⟶c u ⟹ u = s2"
  assume IH1: "⋀u. ⟨c1,s2⟩ ⟶c u ⟹ u = s1"
  assume "⟨c0;c1, s⟩ ⟶_c u"
  then obtain s' where
      c0: "⟨c0,s⟩ ⟶_c s'" and
      c1: "⟨c1,s'⟩ ⟶_c u"
    by auto

  from c0 IH0 have "s'=s2" by blast
  with c1 IH1 show "u=s1" by blast
next
  fix b c0 c1 s s1 u
  assume IH: "⋀u. ⟨c0,s⟩ ⟶_c u ⟹ u = s1"

  assume "b s" and "⟨if b then c0 else c1,s⟩ ⟶_c u"
  hence "⟨c0, s⟩ ⟶_c u" by simp
  with IH show "u = s1" by blast
next
  fix b c0 c1 s s1 u
  assume IH: "⋀u. ⟨c1,s⟩ ⟶_c u ⟹ u = s1"

  assume "¬b s" and "⟨if b then c0 else c1,s⟩ ⟶_c u"
  hence "⟨c1, s⟩ ⟶_c u" by simp
  with IH show "u = s1" by blast
next
  fix b c s u
  assume "¬b s" and "⟨while b do c,s⟩ ⟶_c u"
  thus "u = s" by simp
next
  fix b c s s1 s2 u
  assume "IH_c": "⋀u. ⟨c,s⟩ ⟶_c u ⟹ u = s2"
  assume "IH_w": "⋀u. ⟨while b do c,s2⟩ ⟶_c u ⟹ u = s1"

  assume "b s" and "⟨while b do c,s⟩ ⟶_c u"
  then obtain s' where
      c: "⟨c,s⟩ ⟶_c s'" and
      w: "⟨while b do c,s'⟩ ⟶_c u"
    by auto

  from c "IH_c" have "s' = s2" by blast
  with w "IH_w" show "u = s1" by blast
qed
```

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"
  using prems
proof (induct arbitrary: u)
  — the simple skip case for demonstration:
```

```
  fix s u assume "⟨skip,s⟩ ⟶c u"
  thus "u = s" by simp
next
  — and the only really interesting case, while:
  fix b c s s1 s2 u
  assume "IHc": "⋀u. ⟨c,s⟩ ⟶c u ⟹ u = s2"
  assume "IHw": "⋀u. ⟨while b do c,s2⟩ ⟶c u ⟹ u = s1"

  assume "b s" and "⟨while b do c,s⟩ ⟶c u"
  then obtain s' where
      c: "⟨c,s⟩ ⟶c s'" and
      w: "⟨while b do c,s'⟩ ⟶c u"
    by auto

  from c "IHc" have "s' = s2" by blast
  with w "IHw" show "u = s1" by blast
qed (best dest: evalc_cases [THEN iffD1])+ — prove the rest automatically

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)"
  by (simp add: dlift_def)
```

```
lemma cont_dlift [iff]: "cont (%f. dlift f)"
  by (simp add: dlift_def)

lemma dlift_is_Def [simp]:
    "(dlift f·l = Def y) = (∃x. l = Def x ∧ f·(Discr x) = Def y)"
  by (simp add: dlift_def split: lift.split)

lemma eval_implies_D: "⟨c,s⟩ −→c t ==> D c·(Discr s) = (Def t)"
  apply (induct set: evalc)
        apply simp_all
   apply (subst fix_eq)
   apply simp
  apply (subst fix_eq)
  apply simp
  done

lemma D_implies_eval: "!s t. D c·(Discr s) = (Def t) --> ⟨c,s⟩ −→c t"
  apply (induct c)
      apply simp
     apply simp
    apply force
   apply (simp (no_asm))
   apply force
  apply (simp (no_asm))
  apply (rule fix_ind)
    apply (fast intro!: adm_lemmas adm_chfindom ax_flat)
   apply (simp (no_asm))
  apply (simp (no_asm))
  apply safe
  apply fast
  done

theorem D_is_eval: "(D c·(Discr s) = (Def t)) = (⟨c,s⟩ −→c t)"
  by (fast elim!: D_implies_eval [rule_format] eval_implies_D)

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}"
  apply (unfold hoare_valid_def)
  apply (simp (no_asm))
  apply (rule fix_ind)
    apply (simp (no_asm)) — simplifier with enhanced adm-tactic
   apply (simp (no_asm))
  apply (simp (no_asm))
  apply blast
  done

end
```

References

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