IMP in HOLCF

Tobias Nipkow and Robert Sandner

June 21, 2010

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)
```

notation *(latex)*
```
  SKIP  ("skip") and
  Cond  ("if _ then _ else _"  60) and
  While  ("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`:

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

notation *(xsymbols)*
```
  update  ("_/[_ ↦ /_]" [900,0,0] 900)
```

Disable conflicting syntax from HOL Map theory.

no_syntax
```
  "_maplet"  :: "['a, 'a] => maplet"             ("_ /|->/ _")
  "_maplets" :: "['a, 'a] => maplet"             ("_ /[|->]/ _")
  ""         :: "maplet => maplets"             ("_")
  "_Maplets" :: "[maplet, maplets] => maplets" ("_,/ _")
  "_MapUpd"  :: "['a ~=> 'b, maplets] => 'a ~=> 'b" ("_/'(_')" [900,0]900)
  "_Map"     :: "maplets => 'a ~=> 'b"            ("(1[_])")
```

The big-step execution relation `evalc` is defined inductively:

inductive
```
  evalc :: "[com,state,state] ⇒ bool" ("⟨_,_⟩/ ⟶_ _" [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. This property is called rule inversion.

```
inductive_cases skipE [elim!]:   "⟨skip,s⟩ ⟶c s'"
inductive_cases semiE [elim!]:   "⟨c0; c1, s⟩ ⟶c s'"
inductive_cases assignE [elim!]: "⟨x :== a,s⟩ ⟶c s'"
inductive_cases ifE [elim!]:     "⟨if b then c0 else c1, s⟩ ⟶c s'"
inductive_cases whileE [elim]:  "⟨while b do c,s⟩ ⟶c s'"
```

The next proofs are all trivial by rule inversion.

lemma *skip:*
"⟨skip,s⟩ ⟶c s' = (s' = s)"
⟨*proof*⟩

lemma *assign:*
"⟨x :== a,s⟩ ⟶c s' = (s' = s[x↦a s])"
⟨*proof*⟩

lemma *semi:*
"⟨c0; c1, s⟩ ⟶c s' = (∃s''. ⟨c0,s⟩ ⟶c s'' ∧ ⟨c1,s''⟩ ⟶c s')"
⟨*proof*⟩

lemma *ifTrue:*
"b s ⟹ ⟨if b then c0 else c1, s⟩ ⟶c s' = ⟨c0,s⟩ ⟶c s'"

⟨*proof*⟩

lemma *ifFalse:*
"¬b s ⟹ ⟨if b then c0 else c1, s⟩ ⟶$_c$ s' = ⟨c1,s⟩ ⟶$_c$ s'"
⟨*proof*⟩

lemma *whileFalse:*
"¬ b s ⟹ ⟨while b do c,s⟩ ⟶$_c$ s' = (s' = s)"
⟨*proof*⟩

lemma *whileTrue:*
"b s ⟹
⟨while b do c, s⟩ ⟶$_c$ s' =
(∃s''. ⟨c,s⟩ ⟶$_c$ s'' ∧ ⟨while b do c, s''⟩ ⟶$_c$ 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:

definition
equiv_c :: "com ⇒ com ⇒ bool" ("_ ∼ _" [56, 56] 55) **where**
"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*⟩

Happily, such lengthy proofs are seldom necessary. Isabelle can prove many such facts automatically.

lemma
"(while b do c) ∼ (if b then c; while b do c else skip)"
⟨*proof*⟩

lemma *triv_if:*
"(if b then c else c) ∼ c"
⟨*proof*⟩

lemma *commute_if:*
"(if b1 then (if b2 then c11 else c12) else c2)
∼
(if b2 then (if b1 then c11 else c2) else (if b1 then c12 else c2))"
⟨*proof*⟩

lemma *while_equiv:*
"⟨c0, s⟩ −→$_c$ u ⟹ c ∼ c' ⟹ (c0 = while b do c) ⟹ ⟨while b do c', s⟩ −→$_c$ u"
⟨*proof*⟩

lemma *equiv_while:*
"c ∼ c' ⟹ (while b do c) ∼ (while b do c')"
⟨*proof*⟩

Program equivalence is an equivalence relation.

lemma *equiv_refl:*
"c ∼ c"
⟨*proof*⟩

lemma *equiv_sym:*
"c1 ∼ c2 ⟹ c2 ∼ c1"
⟨*proof*⟩

lemma *equiv_trans:*
"c1 ∼ c2 ⟹ c2 ∼ c3 ⟹ c1 ∼ c3"
⟨*proof*⟩

Program constructions preserve equivalence.

lemma *equiv_semi:*
"c1 ∼ c1' ⟹ c2 ∼ c2' ⟹ (c1; c2) ∼ (c1'; c2')"
⟨*proof*⟩

lemma *equiv_if:*
"c1 ∼ c1' ⟹ c2 ∼ c2' ⟹ (if b then c1 else c2) ∼ (if b then c1' else c2')"
⟨*proof*⟩

lemma *while_never:* "⟨c, s⟩ −→$_c$ u ⟹ c ≠ while (λs. True) do c1"
⟨*proof*⟩

lemma *equiv_while_True:*

```
  "(while (λs. True) do c1) ∼ (while (λs. True) do c2)"
```
⟨*proof*⟩

2.3 Execution is deterministic

This proof is automatic.

theorem `"⟨c,s⟩ ⟶c t ⟹ ⟨c,s⟩ ⟶c u ⟹ u = t"`
⟨*proof*⟩

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 "../../HOL/IMP/Natural"* **begin**

Disable conflicting syntax from HOL Map theory.

```
no_syntax
  "_maplet"  :: "['a, 'a] => maplet"             ("_ /|->/ _")
  "_maplets" :: "['a, 'a] => maplet"             ("_ /[|->]/ _")
  ""         :: "maplet => maplets"             ("_")
  "_Maplets" :: "[maplet, maplets] => maplets" ("_,/ _")
  "_MapUpd"  :: "['a ~=> 'b, maplets] => 'a ~=> 'b" ("_/'(_')" [900,0]900)
  "_Map"     :: "maplets => 'a ~=> 'b"            ("(1[_])")
```

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))"

primrec D :: "com => state discr -> state lift"
where
  "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} = (∀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.