2 Natural Semantics of Commands

theory Natural = Com:

2.1 Execution of commands

consts evalc :: (com × state × state) set
  @evalc :: [com, state, state] ⇒ bool

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

translations \((c,s) \xrightarrow{-c} s' == (c,s,s') \in evalc\)

constdefs update :: ('a ⇒ 'b) ⇒ 'a ⇒ 'b ⇒ ('a ⇒ 'b)
update == fun_upd

The big-step execution relation \(evalc\) is defined inductively:

inductive evalc

intros

Skip: \(\langle \text{skip}, s \rangle \xrightarrow{-c} s\)
Assign: \(\langle x := a, s \rangle \xrightarrow{-c} s[x := a]\)

Semi: \[
\langle c0, s \rangle \xrightarrow{-c} s''; \langle c1, s'' \rangle \xrightarrow{-c} s' \Rightarrow \langle c0 \circ c1, s \rangle \xrightarrow{-c} s'
\]

IfTrue: \[
\langle b s; \langle c0, s \rangle \xrightarrow{-c} s' \rangle \Rightarrow \langle \text{if } b \text{ then } c0 \text{ else } c1, s \rangle \xrightarrow{-c} s'
\]

IfFalse: \[
\langle \neg b s; \langle c1, s \rangle \xrightarrow{-c} s' \rangle \Rightarrow \langle \text{if } b \text{ then } c0 \text{ else } c1, s \rangle \xrightarrow{-c} s'
\]

WhileFalse: \[\neg b s \Rightarrow \langle \text{while } b \text{ do } c, s \rangle \xrightarrow{-c} s\]

WhileTrue: \[
\langle b s; \langle c, s \rangle \xrightarrow{-c} s''; \langle \text{while } b \text{ do } c, s'' \rangle \xrightarrow{-c} s'\rangle
\Rightarrow \langle \text{while } b \text{ do } c, s \rangle \xrightarrow{-c} s'
\]

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

The induction principle induced by this definition looks like this:

\[
\begin{align*}
\langle xc, xb \rangle \xrightarrow{-c} xa; \quad \land s. \ P \ (\text{skip} \ s \ s); \quad \land a \ s \ x. \ P \ (x := a) \ s \ (s[x := a] s); \\
\land c0 \ c1 \ s \ s' \ s''; \\
\langle c0, s \rangle \xrightarrow{-c} s''; \quad P \ c0 \ s \ s''; \quad \langle c1, s'' \rangle \xrightarrow{-c} s'\; P \ c1 \ s'' \ s'
\Rightarrow P \ (c0 \ c1) \ s \ s'; \\
\land b \ c0 \ c1 \ s \ s'. \ \langle b \ s; \langle c0, s \rangle \xrightarrow{-c} s'; \ P \ c0 \ s \ s' \rangle \Rightarrow P \ (\text{if } b \text{ then } c0 \text{ else } c1) \ s \ s'; \\
\land b \ c \ s. \ \neg b \ s \Rightarrow P \ (\text{while } b \text{ do } c) \ s \ s; \\
\land b \ c \ s \ s' \ s''.
\end{align*}
\]

1
⇒ P (while b do c) s s'
⇒ P xc xb xa

(∧ 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:
  \((\langle \text{skip}, s \rangle -c \rightarrow s') = (s' = s)\)
  by (rule, erule evalc.elims) auto

lemma assign:
  \((\langle x := a, s \rangle -c \rightarrow s') = (s' = s[x\mapsto a])\)
  by (rule, erule evalc.elims) auto

lemma semi:
  \((\langle c_0; c_1, s \rangle -c \rightarrow s') = (\exists s''. (\langle c_0, s \rangle -c \rightarrow s'' \land (c_1,s'') -c \rightarrow s'))\)
  by (rule, erule evalc.elims) auto

lemma ifTrue:
  \(b s \Rightarrow (\langle \text{if } b \text{ then } c_0 \text{ else } c_1, s \rangle -c \rightarrow s') = (\langle c_0, s \rangle -c \rightarrow s')\)
  by (rule, erule evalc.elims) auto

lemma ifFalse:
  \(\neg b s \Rightarrow (\langle \text{if } b \text{ then } c_0 \text{ else } c_1, s \rangle -c \rightarrow s') = (\langle c_1, s \rangle -c \rightarrow s')\)
  by (rule, erule evalc.elims) auto

lemma whileFalse:
  \(\neg b s \Rightarrow (\langle \text{while } b \text{ do } c, s \rangle -c \rightarrow s') = (s' = s)\)
  by (rule, erule evalc.elims) auto

lemma whileTrue:
  \(b s \Rightarrow (\langle \text{while } b \text{ do } c, s \rangle -c \rightarrow s') =\)
  \(\exists s''. (c, s) -c \rightarrow s'' \land (\langle \text{while } b \text{ do } c, s'' \rangle -c \rightarrow s')\)
  by (rule, erule evalc.elims) 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')

A small proof rule, telling Isabelle to unfold the definition automatically if there is something
to be proved about equivalent states:

lemma equivI [intro!]:
  (∀ s s'. (⟨c, s⟩ -c→ s') = (⟨c', s⟩ -c→ s')) ⇒ c ∼ c'
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 been 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
\[
\langle c, s \rangle \to_c s' \text{ and } \langle \text{?w, s''} \rangle \to_c s' \text{ by (cases set: evalc) auto}
\]
— with this information, we can build a derivation tree for the while
with \( b \)
have \( \langle \text{?w, s} \rangle \to_c s' \) by (rule WhileTrue)
}
ultimately
— both cases together again give us what we want:
have \( \langle \text{?w, s} \rangle \to_c s' \) by blast
}
ultimately
show \( \text{?thesis} \) by blast
qed

2.3 Execution is deterministic

The following proof presents all the details:

**theorem** com_det: \( \langle c,s \rangle \to_c t \land \langle c,s \rangle \to_c u \implies u=t \)
**proof** clarify — transform the goal into canonical form
assume \( \langle c,s \rangle \to_c t \)
thus \( \forall u. \langle c,s \rangle \to_c u \implies u=t \)
**proof** (induct set: evalc)
fix \( s u \) assume \( \langle \text{skip},s \rangle \to_c u \)
thus \( u = s \) by simp
next
fix \( a s x u \) assume \( \langle x := a,s \rangle \to_c u \)
thus \( u = s[x := a \ s] \) by simp
next
fix \( c0 c s1 s2 u \)
assume \( \text{IH0: } \forall u. \langle c0, s \rangle \to_c u \implies u = s2 \)
assume \( \text{IH1: } \forall u. \langle c1, s2 \rangle \to_c u \implies u = s1 \)
assume \( \langle c0; c1, s \rangle \to_c u \)
then obtain \( s' \) where
\( c0: \langle c0, s \rangle \to_c s' \) and 
\( c1: \langle c1, s' \rangle \to_c u \)
by auto
from \( c0 \text{ IH0} \) have \( s'=s2 \) by blast
with \( c1 \text{ IH1} \) show \( u=s1 \) by blast
next
fix \( b c0 c1 s s1 u \)
assume \( \text{IH: } \forall u. \langle c0, s \rangle \to_c u \implies u = s1 \)
assume \( b s \) and \( \langle \text{if } b \text{ then } c0 \text{ else } c1, s \rangle \to_c u \)
hence \(\langle c0, s \rangle \rightarrow c \rightarrow u\) by simp
with IH show \(u = s1\) by blast

next
fix \(b, c, c0, c1, s, s1, s2, u\)
assume IH: \(\bigwedge u. \langle c1, s \rangle \rightarrow c \rightarrow u \implies u = s1\)
assume \(\neg b\) and \(\langle\text{if } b \text{ then } c0 \text{ else } c1, s \rangle \rightarrow c \rightarrow u\)
hence \(\langle c1, s \rangle \rightarrow c \rightarrow u\) by simp
with IH show \(u = s1\) by blast

next
fix \(b, c, s, s1, s2, u\)
assume \(\langle\text{while } b \text{ do } c, s \rangle \rightarrow c \rightarrow u\)
thus \(u = s\) by simp

next
fix \(b, c, s, s1, s2, u\)
assume IH\(c\): \(\bigwedge u. \langle c, s \rangle \rightarrow c \rightarrow u \implies u = s2\)
assume IH\(w\): \(\bigwedge u. \langle\text{while } b \text{ do } c, s2 \rangle \rightarrow c \rightarrow u \implies u = s1\)
assume \(b\) and \(\langle\text{while } b \text{ do } c, s \rangle \rightarrow c \rightarrow u\)
then obtain \(s'\) where
\(c\): \(\langle c, s \rangle \rightarrow c \rightarrow s'\) and
\(w\): \(\langle\text{while } b \text{ do } c, s' \rangle \rightarrow c \rightarrow u\)
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:

\(\text{theorem } \langle c, s \rangle \rightarrow c \rightarrow t \land \langle c, s \rangle \rightarrow c \rightarrow u \implies u = t\)

\(\text{proof clarify}\)
assume \(\langle c, s \rangle \rightarrow c \rightarrow t\)
thus \(\bigwedge u. \langle c, s \rangle \rightarrow c \rightarrow u \implies u = t\)

\(\text{proof (induct set: evalc)}\)
— the simple skip case for demonstration:
fix \(s, u\) assume \(\langle\text{skip, } s \rangle \rightarrow c \rightarrow u\)
thus \(u = s\) by simp

next
— and the only really interesting case, while:
fix \(b, c, s, s1, s2, u\)
assume IH\(c\): \(\bigwedge u. \langle c, s \rangle \rightarrow c \rightarrow u \implies u = s2\)
assume IH\(w\): \(\bigwedge u. \langle\text{while } b \text{ do } c, s2 \rangle \rightarrow c \rightarrow u \implies u = s1\)

assume \(b, s\) and \(\langle\text{while } b \text{ do } c, s \rangle \rightarrow c \rightarrow u\)
then obtain \(s'\) where
c: \langle c, s \rangle -c \to s' and
w: \langle \text{while } b \text{ do } c, s' \rangle -c \to u
by auto

from c \text{ IH}, have } s' = s_2 \text{ by blast}
with w \text{ IH} show } u = s_1 \text{ by blast}
qed (\text{best dest: evalc_cases [THEN iffD1]+} — prove the rest automatically
qed

end