2 Natural Semantics of Commands

theory *Natural = Com:*

2.1 Execution of commands

```
consts  evalc  :: (com × state × state) set
        @evalc :: [com,state,state] ⇒ bool (⟨_,_⟩/ -c→ _ [0,0,50] 50)
```

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

```
translations  ⟨c,s⟩ -c→ s' == (c,s,s') ∈ evalc
```

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

The big-step execution relation evalc is defined inductively:

```
inductive evalc
  intros
  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:

```
⟦⟨xc,xb⟩ -c→ xa; ⋀s. P skip s s; ⋀a s x. P (x :== a ) s (s[x ↦ a s]);
   ⋀c0 c1 s s' s''.
      ⟦⟨c0,s⟩ -c→ s''; P c0 s s''; ⟨c1,s''⟩ -c→ s'; P c1 s'' s'⟧
      ⟹ P (c0; c1) s s';
   ⋀b c0 c1 s s'. ⟦b s; ⟨c0,s⟩ -c→ s'; P c0 s s'⟧ ⟹ P (if b then c0 else c1) s s';
   ⋀b c0 c1 s s'.
      ⟦¬ b s; ⟨c1,s⟩ -c→ s'; P c1 s s'⟧ ⟹ P (if b then c0 else c1) s s';
   ⋀b c s. ¬ b s ⟹ P (while b do c) s s;
   ⋀b c s 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 xc xb xa
```

($\bigwedge$ 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:
  (⟨skip,s⟩ -c→ s') = (s' = s)
  by (rule, erule evalc.elims) auto

lemma assign:
  (⟨x :== a,s⟩ -c→ s') = (s' = s[x↦a s])
  by (rule, erule evalc.elims) auto

lemma semi:
  (⟨c0; c1, s⟩ -c→ s') = (∃s''. ⟨c0,s⟩ -c→ s'' ∧ ⟨c1,s''⟩ -c→ s')
  by (rule, erule evalc.elims) auto

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

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

lemma whileFalse:
  ¬ b s ⟹ (⟨while b do c,s⟩ -c→ s') = (s' = s)
  by (rule, erule evalc.elims) 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.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 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: ⟨c,s⟩ -c→ t ∧ ⟨c,s⟩ -c→ u ⟶ u=t
proof clarify — transform the goal into canonical form
  assume ⟨c,s⟩ -c→ t
  thus ⋀u. ⟨c,s⟩ -c→ u ⟹ u=t
  proof (induct 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
qed
```

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

```
theorem ⟨c,s⟩ -c→ t ∧ ⟨c,s⟩ -c→ u ⟶ u=t
proof clarify
  assume ⟨c,s⟩ -c→ t
  thus ⋀u. ⟨c,s⟩ -c→ u ⟹ u=t
  proof (induct set: evalc)
    — 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 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 (best dest: evalc_cases [THEN iffD1])+ — prove the rest automatically
qed

end
```