3 Transition Semantics of Commands

theory *Transition = Natural:*

3.1 The transition relation

We formalize the transition semantics as in [1]. This makes some of the rules a bit more intuitive, but also requires some more (internal) formal overhead.
Since configurations that have terminated are written without a statement, the transition relation is not *((com × state) × com × state) set* but instead:

consts *evalc1 :: ((com option × state) × (com option × state)) set*

Some syntactic sugar that we will use to hide the *option* part in configurations:

syntax
@angle :: [com, state] ⇒ com option × state (⟨_,_⟩)
@angle2 :: state ⇒ com option × state (⟨_⟩)

translations
⟨c,s⟩ == (Some c, s)
⟨s⟩ == (None, s)

More syntactic sugar for the transition relation, and its iteration.

syntax
@evalc1 :: [(com option×state),(com option×state)] ⇒ bool
(_ ⟶₁ _ [81,81] 100)
@evalcn :: [(com option×state),nat,(com option×state)] ⇒ bool
(_ -_→₁ _ [81,81] 100)
@evalc :: [(com option×state),(com option×state)] ⇒ bool*
(_ ⟶₁ _ [81,81] 100)*

translations
cs ⟶₁ cs' == (cs,cs') ∈ evalc1
cs -n→₁ cs' == (cs,cs') ∈ evalc1^n
cs ⟶₁ cs' == (cs,cs') ∈ evalc1^**

— Isabelle converts *(cs0,(c1,s1))* to *(cs0, c1, s1)*, so we also include:
cs0 -n→₁ (c1,s1) == (cs0,c1,s1) ∈ evalc1^n
cs0 ⟶₁ (c1,s1) == (cs0,c1,s1) ∈ evalc1
cs0 ⟶₁ (c1,s1) == (cs0,c1,s1) ∈ evalc1^**

Now, finally, we are set to write down the rules for our small step semantics:

inductive *evalc1*
intros
Skip: ⟨skip, s⟩ ⟶₁ ⟨s⟩
Assign: ⟨x :== a, s⟩ ⟶₁ ⟨s[x ↦ a s]⟩

Semi1: $\langle c0,s\rangle \longrightarrow_1 \langle s'\rangle \Longrightarrow \langle c0;c1,s\rangle \longrightarrow_1 \langle c1,s'\rangle$
Semi2: $\langle c0,s\rangle \longrightarrow_1 \langle c0',s'\rangle \Longrightarrow \langle c0;c1,s\rangle \longrightarrow_1 \langle c0';c1,s'\rangle$

IfTrue: $b\ s \Longrightarrow \langle \text{if } b \text{ then } c1 \text{ else } c2,s\rangle \longrightarrow_1 \langle c1,s\rangle$
IfFalse: $\neg b\ s \Longrightarrow \langle \text{if } b \text{ then } c1 \text{ else } c2,s\rangle \longrightarrow_1 \langle c2,s\rangle$

While: $\langle \text{while } b \text{ do } c,s\rangle \longrightarrow_1 \langle \text{if } b \text{ then } c; \text{ while } b \text{ do } c \text{ else skip},s\rangle$

lemmas *[intro] = evalc1.intros* — again, use these rules in automatic proofs

As for the big step semantics you can read these rules in a syntax directed way:

lemma *SKIP_1:* $\langle \text{skip}, s\rangle \longrightarrow_1 y = (y = \langle s\rangle)$
by *(rule, cases set: evalc1, auto)*

lemma *Assign_1:* $\langle x :== a, s\rangle \longrightarrow_1 y = (y = \langle s[x \mapsto a\ s]\rangle)$
by *(rule, cases set: evalc1, auto)*

lemma *Cond_1:*
$\langle \text{if } b \text{ then } c1 \text{ else } c2, s\rangle \longrightarrow_1 y = ((b\ s \longrightarrow y = \langle c1, s\rangle) \wedge (\neg b\ s \longrightarrow y = \langle c2, s\rangle))$
by *(rule, cases set: evalc1, auto)*

lemma *While_1:* $\langle \text{while } b \text{ do } c, s\rangle \longrightarrow_1 y = (y = \langle \text{if } b \text{ then } c; \text{ while } b \text{ do } c \text{ else skip}, s\rangle)$
by *(rule, cases set: evalc1, auto)*

lemmas *[simp] = SKIP_1 Assign_1 Cond_1 While_1*

3.2 Examples

lemma
$s\ x = 0 \Longrightarrow \langle \text{while } \lambda s.\ s\ x \neq 1 \text{ do } x:== \lambda s.\ s\ x+1, s\rangle \longrightarrow_1^* \langle s[x \mapsto 1]\rangle$
*(***is** _ $\Longrightarrow \langle ?w, \_\rangle \longrightarrow_1^* \_$*)*
proof -
let *?if* = if $\lambda s.\ s\ x \neq 1$ then *x:==* $\lambda s.\ s\ x+1$; *?w* else skip
assume *[simp]: s x = 0*
have $\langle ?w, s\rangle \longrightarrow_1 \langle ?if, s\rangle$ **..**
also have $\langle ?if, s\rangle \longrightarrow_1 \langle x:== \lambda s.\ s\ x+1; ?w, s\rangle$ **by** *simp*
also have $\langle x:== \lambda s.\ s\ x+1; ?w, s\rangle \longrightarrow_1 \langle ?w, s[x \mapsto 1]\rangle$ **by** *(rule Semi1, simp)*
also have $\langle ?w, s[x \mapsto 1]\rangle \longrightarrow_1 \langle ?if, s[x \mapsto 1]\rangle$ **..**
also have $\langle ?if, s[x \mapsto 1]\rangle \longrightarrow_1 \langle \text{skip}, s[x \mapsto 1]\rangle$ **by** *(simp add: update_def)*
also have $\langle \text{skip}, s[x \mapsto 1]\rangle \longrightarrow_1 \langle s[x \mapsto 1]\rangle$ **..**
finally show *?thesis* **.**
qed

lemma
$s\ x = 2 \Longrightarrow \langle \text{while } \lambda s.\ s\ x \neq 1 \text{ do } x:== \lambda s.\ s\ x+1, s\rangle \longrightarrow_1^* s'$
*(***is** _ $\Longrightarrow \langle ?w, \_\rangle \longrightarrow_1^* s'$*)*
proof -

let *?c = x:== λs. s x+1*
let *?if* = if *λs. s x ≠ 1* then *?c; ?w* else skip
assume *[simp]: s x = 2*
note *update_def [simp]*
have ⟨*?w, s*⟩ ⟶₁ ⟨*?if, s*⟩ **..**
also have ⟨*?if, s*⟩ ⟶₁ ⟨*?c; ?w, s*⟩ **by** *simp*
also have ⟨*?c; ?w, s*⟩ ⟶₁ ⟨*?w, s[x ↦ 3]*⟩ **by** *(rule Semi1, simp)*
also have ⟨*?w, s[x ↦ 3]*⟩ ⟶₁ ⟨*?if, s[x ↦ 3]*⟩ **..**
also have ⟨*?if, s[x ↦ 3]*⟩ ⟶₁ ⟨*?c; ?w, s[x ↦ 3]*⟩ **by** *simp*
also have ⟨*?c; ?w, s[x ↦ 3]*⟩ ⟶₁ ⟨*?w, s[x ↦ 4]*⟩ **by** *(rule Semi1, simp)*
also have ⟨*?w, s[x ↦ 4]*⟩ ⟶₁ ⟨*?if, s[x ↦ 4]*⟩ **..**
also have ⟨*?if, s[x ↦ 4]*⟩ ⟶₁ ⟨*?c; ?w, s[x ↦ 4]*⟩ **by** *simp*
also have ⟨*?c; ?w, s[x ↦ 4]*⟩ ⟶₁ ⟨*?w, s[x ↦ 5]*⟩ **by** *(rule Semi1, simp)*
oops

3.3 Basic properties

There are no *stuck* programs:

lemma *no_stuck:* ∃*y.* ⟨*c,s*⟩ ⟶₁ *y*
proof *(induct c)*
— case Semi:
fix *c1 c2* **assume** ∃*y.* ⟨*c1,s*⟩ ⟶₁ *y*
then obtain *y* **where** ⟨*c1,s*⟩ ⟶₁ *y* **..**
then obtain *c1' s'* **where** ⟨*c1,s*⟩ ⟶₁ ⟨*s'*⟩ ∨ ⟨*c1,s*⟩ ⟶₁ ⟨*c1',s'*⟩
by *(cases y, cases fst y, auto)*
thus ∃*s'.* ⟨*c1;c2,s*⟩ ⟶₁ *s'* **by** *auto*
next
— case If:
fix *b c1 c2* **assume** ∃*y.* ⟨*c1,s*⟩ ⟶₁ *y* **and** ∃*y.* ⟨*c2,s*⟩ ⟶₁ *y*
thus ∃*y.* ⟨if *b* then *c1* else *c2, s*⟩ ⟶₁ *y* **by** *(cases b s) auto*
qed *auto* — the rest is trivial

If a configuration does not contain a statement, the program has terminated and there is no next configuration:

lemma *stuck [dest]: (None, s)* ⟶₁ *y* ⟹ *False* **by** *(auto elim: evalc1.elims)*

Therefore we also know:

lemma *evalc_None_0 [simp]:* ⟨*s*⟩ -*n*→₁ *y = (n = 0* ∧ *y =* ⟨*s*⟩*)*
by *(cases n) auto*

lemma *SKIP_n:* ⟨skip, *s*⟩ -*n*→₁ ⟨*s'*⟩ ⟹ *s = s'*
by *(cases n) auto*

3.4 Equivalence to natural semantics

We first need two lemmas about semicolon statements: decomposition and composition.

```
lemma semiD:
  ⋀c1 c2 s s''. ⟨c1; c2, s⟩ -n→₁ ⟨s''⟩ ⟹
  ∃i j s'. ⟨c1, s⟩ -i→₁ ⟨s'⟩ ∧ ⟨c2, s'⟩ -j→₁ ⟨s''⟩ ∧ n = i+j
  (is PROP ?P n)
proof (induct n)
  show PROP ?P 0 by simp
next
  fix n assume IH: PROP ?P n
  show PROP ?P (Suc n)
  proof -
    fix c1 c2 s s''
    assume ⟨c1; c2, s⟩ -Suc n→₁ ⟨s''⟩
    then obtain y where
      1: ⟨c1; c2, s⟩ ⟶₁ y and
      n: y -n→₁ ⟨s''⟩
      by blast

    from 1
    show ∃i j s'. ⟨c1, s⟩ -i→₁ ⟨s'⟩ ∧ ⟨c2, s'⟩ -j→₁ ⟨s''⟩ ∧ Suc n = i+j
      (is ∃i j s'. ?Q i j s')
    proof (cases set: evalc1)
      case Semi1
      then obtain s' where
        y = ⟨c2, s'⟩ and ⟨c1, s⟩ ⟶₁ ⟨s'⟩
        by auto
      with 1 n have ?Q 1 n s' by simp
      thus ?thesis by blast
    next
      case Semi2
      then obtain c1' s' where
        y:  y = ⟨c1'; c2, s'⟩ and
        c1: ⟨c1, s⟩ ⟶₁ ⟨c1', s'⟩
        by auto
      with n have ⟨c1'; c2, s'⟩ -n→₁ ⟨s''⟩ by simp
      with IH obtain i j s0 where
        c1': ⟨c1',s'⟩ -i→₁ ⟨s0⟩ and
        c2:  ⟨c2,s0⟩ -j→₁ ⟨s''⟩ and
        i:   n = i+j
        by blast

      from c1 c1'
      have ⟨c1,s⟩ -(i+1)→₁ ⟨s0⟩ by (auto simp del: relpow.simps intro: rel_pow_Suc_I2)
      with c2 i
      have ?Q (i+1) j s0 by simp
      thus ?thesis by blast
    qed auto — the remaining cases cannot occur
```

```
  qed
qed


lemma semiI:
  ⋀c0 s s''. ⟨c0,s⟩ -n→₁ ⟨s''⟩ ⟹ ⟨c1,s''⟩ ⟶₁* ⟨s'⟩ ⟹ ⟨c0; c1, s⟩ ⟶₁* ⟨s'⟩
proof (induct n)
  fix c0 s s'' assume ⟨c0,s⟩ -(0::nat)→₁ ⟨s''⟩
  hence False by simp
  thus ?thesis c0 s s'' ..
next
  fix c0 s s'' n
  assume c0: ⟨c0,s⟩ -Suc n→₁ ⟨s''⟩
  assume c1: ⟨c1,s''⟩ ⟶₁* ⟨s'⟩
  assume IH: ⋀c0 s s''.
    ⟨c0,s⟩ -n→₁ ⟨s''⟩ ⟹ ⟨c1,s''⟩ ⟶₁* ⟨s'⟩ ⟹ ⟨c0; c1,s⟩ ⟶₁* ⟨s'⟩
  from c0 obtain y where
    1: ⟨c0,s⟩ ⟶₁ y and n: y -n→₁ ⟨s''⟩ by blast
  from 1 obtain c0' s0' where
    y = ⟨s0'⟩ ∨ y = ⟨c0', s0'⟩
    by (cases y, cases fst y, auto)
  moreover
  { assume y: y = ⟨s0'⟩
    with n have s'' = s0' by simp
    with y 1 have ⟨c0; c1,s⟩ ⟶₁ ⟨c1, s''⟩ by blast
    with c1 have ⟨c0; c1,s⟩ ⟶₁* ⟨s'⟩ by (blast intro: rtrancl_trans)
  }
  moreover
  { assume y: y = ⟨c0', s0'⟩
    with n have ⟨c0', s0'⟩ -n→₁ ⟨s''⟩ by blast
    with IH c1 have ⟨c0'; c1,s0'⟩ ⟶₁* ⟨s'⟩ by blast
    moreover
    from y 1 have ⟨c0; c1,s⟩ ⟶₁ ⟨c0'; c1,s0'⟩ by blast
    hence ⟨c0; c1,s⟩ ⟶₁* ⟨c0'; c1,s0'⟩ by blast
    ultimately
    have ⟨c0; c1,s⟩ ⟶₁* ⟨s'⟩ by (blast intro: rtrancl_trans)
  }
  ultimately
  show ⟨c0; c1,s⟩ ⟶₁* ⟨s'⟩ by blast
qed
```

The easy direction of the equivalence proof:

```
lemma evalc_imp_evalc1:
  ⟨c,s⟩ -c→ s' ⟹ ⟨c, s⟩ ⟶₁* ⟨s'⟩
proof -
  assume ⟨c,s⟩ -c→ s'
```

thus ⟨*c*, *s*⟩ −→$_1$* ⟨*s'*⟩
proof *induct*
  fix *s* **show** ⟨skip,*s*⟩ −→$_1$* ⟨*s*⟩ **by** *auto*
next
  fix *x a s* **show** ⟨*x :== a* ,*s*⟩ −→$_1$* ⟨*s*[*x*↦*a s*]⟩ **by** *auto*
next
  fix *c0 c1 s s'' s'*
  assume ⟨*c0*,*s*⟩ −→$_1$* ⟨*s''*⟩
  then obtain *n* **where** ⟨*c0*,*s*⟩ -*n*→$_1$ ⟨*s''*⟩ **by** *(blast dest: rtrancl_pow)*
  moreover
  assume ⟨*c1*,*s''*⟩ −→$_1$* ⟨*s'*⟩
  ultimately
  show ⟨*c0; c1*,*s*⟩ −→$_1$* ⟨*s'*⟩ **by** *(rule semiI)*
next
  fix *s::state* **and** *b c0 c1 s'*
  assume *b s*
  hence ⟨if *b* then *c0* else *c1*,*s*⟩ −→$_1$ ⟨*c0*,*s*⟩ **by** *simp*
  also assume ⟨*c0*,*s*⟩ −→$_1$* ⟨*s'*⟩
  finally show ⟨if *b* then *c0* else *c1*,*s*⟩ −→$_1$* ⟨*s'*⟩ **.**
next
  fix *s::state* **and** *b c0 c1 s'*
  assume ¬*b s*
  hence ⟨if *b* then *c0* else *c1*,*s*⟩ −→$_1$ ⟨*c1*,*s*⟩ **by** *simp*
  also assume ⟨*c1*,*s*⟩ −→$_1$* ⟨*s'*⟩
  finally show ⟨if *b* then *c0* else *c1*,*s*⟩ −→$_1$* ⟨*s'*⟩ **.**
next
  fix *b c* **and** *s::state*
  assume *b:* ¬*b s*
  let *?if* = if *b* then *c;* while *b* do *c* else skip
  have ⟨while *b* do *c*,*s*⟩ −→$_1$ ⟨*?if, s*⟩ **by** *blast*
  also have ⟨*?if*,*s*⟩ −→$_1$ ⟨skip, *s*⟩ **by** *(simp add: b)*
  also have ⟨skip, *s*⟩ −→$_1$ ⟨*s*⟩ **by** *blast*
  finally show ⟨while *b* do *c*,*s*⟩ −→$_1$* ⟨*s*⟩ **.**
next
  fix *b c s s'' s'*
  let *?w* = while *b* do *c*
  let *?if* = if *b* then *c; ?w* else skip
  assume *w:* ⟨*?w*,*s''*⟩ −→$_1$* ⟨*s'*⟩
  assume *c:* ⟨*c*,*s*⟩ −→$_1$* ⟨*s''*⟩
  assume *b:* *b s*
  have ⟨*?w*,*s*⟩ −→$_1$ ⟨*?if, s*⟩ **by** *blast*
  also have ⟨*?if, s*⟩ −→$_1$ ⟨*c; ?w, s*⟩ **by** *(simp add: b)*
  also
  from *c* **obtain** *n* **where** ⟨*c*,*s*⟩ -*n*→$_1$ ⟨*s''*⟩ **by** *(blast dest: rtrancl_pow)*
  with *w* **have** ⟨*c; ?w*,*s*⟩ −→$_1$* ⟨*s'*⟩ **by** - *(rule semiI)*
  finally show ⟨while *b* do *c*,*s*⟩ −→$_1$* ⟨*s'*⟩ **.**

```
  qed
qed
```

Finally, the equivalence theorem:

```
theorem evalc_equiv_evalc1:
  ⟨c, s⟩ -c→ s' = ⟨c,s⟩ ⟶₁* ⟨s'⟩
proof
  assume ⟨c,s⟩ -c→ s'
  show ⟨c, s⟩ ⟶₁* ⟨s'⟩ by (rule evalc_imp_evalc1)
next
  assume ⟨c, s⟩ ⟶₁* ⟨s'⟩
  then obtain n where ⟨c, s⟩ -n→₁ ⟨s'⟩ by (blast dest: rtrancl_pow)
  moreover
  have ⋀c s s'. ⟨c, s⟩ -n→₁ ⟨s'⟩ ⟹ ⟨c,s⟩ -c→ s'
  proof (induct rule: nat_less_induct)
    fix n
    assume IH: ∀m. m < n ⟶ (∀c s s'. ⟨c, s⟩ -m→₁ ⟨s'⟩ ⟶ ⟨c,s⟩ -c→ s')
    fix c s s'
    assume c: ⟨c, s⟩ -n→₁ ⟨s'⟩
    then obtain m where n: n = Suc m by (cases n) auto
    with c obtain y where
      c': ⟨c, s⟩ ⟶₁ y and m: y -m→₁ ⟨s'⟩ by blast
    show ⟨c,s⟩ -c→ s'
    proof (cases c)
      case SKIP
      with c n show ?thesis by auto
    next
      case Assign
      with c n show ?thesis by auto
    next
      fix c1 c2 assume semi: c = (c1; c2)
      with c obtain i j s'' where
        c1: ⟨c1, s⟩ -i→₁ ⟨s''⟩ and
        c2: ⟨c2, s''⟩ -j→₁ ⟨s'⟩ and
        ij: n = i+j
        by (blast dest: semiD)
      from c1 c2 obtain
        0 < i and 0 < j by (cases i, auto, cases j, auto)
      with ij obtain
        i: i < n and j: j < n by simp
      from c1 i IH
      have ⟨c1,s⟩ -c→ s'' by blast
      moreover
      from c2 j IH
      have ⟨c2,s''⟩ -c→ s' by blast
      moreover
      note semi
      ultimately
      show ⟨c,s⟩ -c→ s' by blast
```

```
next
  fix b c1 c2 assume If: c = if b then c1 else c2
  { assume True: b s = True
    with If c n
    have ⟨c1,s⟩ -m→₁ ⟨s'⟩ by auto
    with n IH
    have ⟨c1,s⟩ -c→ s' by blast
    with If True
    have ⟨c,s⟩ -c→ s' by simp
  }
  moreover
  { assume False: b s = False
    with If c n
    have ⟨c2,s⟩ -m→₁ ⟨s'⟩ by auto
    with n IH
    have ⟨c2,s⟩ -c→ s' by blast
    with If False
    have ⟨c,s⟩ -c→ s' by simp
  }
  ultimately
  show ⟨c,s⟩ -c→ s' by (cases b s) auto
next
  fix b c' assume w: c = while b do c'
  with c n
  have if: ⟨if b then c'; while b do c' else skip,s⟩ -m→₁ ⟨s'⟩
    (is ⟨?if,_⟩ -m→₁ _) by auto
  { assume False: b s = False
    with if
    have s' = s by (cases m, auto dest: SKIP_n)
    with w False
    have ⟨c,s⟩ -c→ s' by simp
  }
  moreover
  { assume True: b s = True
    with if n obtain n' where
      ⟨c'; while b do c',s⟩ -n'→₁ ⟨s'⟩ and n': n' < n
      by (cases m) auto
    then obtain i j s'' where
      c': ⟨c',s⟩ -i→₁ ⟨s''⟩ and
      w': ⟨while b do c',s''⟩ -j→₁ ⟨s'⟩ and
      ij: n' = i+j
      by (blast dest: semiD)
    from ij n' obtain
      i: i < n and j: j < n by simp
    from i c' IH
    have ⟨c',s⟩ -c→ s'' by blast
    moreover
    from j w' IH
    have ⟨while b do c',s''⟩ -c→ s' by blast
    moreover
```

```
        note w True
        ultimately
        have ⟨c,s⟩ -c→ s' by blast
      }
      ultimately
      show ⟨c,s⟩ -c→ s' by (cases b s) auto
    qed
  qed
  ultimately
  show ⟨c,s⟩ -c→ s' by blast
qed


end
```

References

[1] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications*. Wiley, 1992.