4 A Simple Compiler

```
theory Compiler = Natural:
```

4.1 An abstract, simplistic machine

There are only three instructions:

```
datatype instr = ASIN loc aexp | JMPF bexp nat | JMPB nat
```

We describe execution of programs in the machine by an operational (small step) semantics:

```
consts  stepa1 :: instr list ⇒ ((state×nat) × (state×nat))set

syntax
  @stepa1 :: [instr list,state,nat,state,nat] ⇒ bool
             (_ ⊢ ⟨_,_⟩/ -1→ ⟨_,_⟩ [50,0,0,0,0] 50)
  @stepa :: [instr list,state,nat,state,nat] ⇒ bool
             (_ ⊢/ ⟨_,_⟩/ -*→ ⟨_,_⟩ [50,0,0,0,0] 50)

translations
  P ⊢ ⟨s,m⟩ -1→ ⟨t,n⟩ == ((s,m),t,n) : stepa1 P
  P ⊢ ⟨s,m⟩ -*→ ⟨t,n⟩ == ((s,m),t,n) : ((stepa1 P)^*)

inductive stepa1 P
intros
ASIN[simp]:
  ⟦ n<size P; P!n = ASIN x a ⟧ ⟹ P ⊢ ⟨s,n⟩ -1→ ⟨s[x↦ a s],Suc n⟩
JMPFT[simp,intro]:
  ⟦ n<size P; P!n = JMPF b i;  b s ⟧ ⟹ P ⊢ ⟨s,n⟩ -1→ ⟨s,Suc n⟩
JMPFF[simp,intro]:
  ⟦ n<size P; P!n = JMPF b i; ~b s; m=n+i ⟧ ⟹ P ⊢ ⟨s,n⟩ -1→ ⟨s,m⟩
JMPB[simp]:
  ⟦ n<size P; P!n = JMPB i; i <= n; j = n-i ⟧ ⟹ P ⊢ ⟨s,n⟩ -1→ ⟨s,j⟩
```

4.2 The compiler

```
consts compile :: com ⇒ instr list
primrec
compile skip = []
compile (x:==a) = [ASIN x a]
compile (c1;c2) = compile c1 @ compile c2
compile (if b then c1 else c2) =
 [JMPF b (length(compile c1) + 2)] @ compile c1 @
 [JMPF (%x. False) (length(compile c2)+1)] @ compile c2
compile (while b do c) = [JMPF b (length(compile c) + 2)] @ compile c @
 [JMPB (length(compile c)+1)]

declare nth_append[simp]
```

4.3 Context lifting lemmas

Some lemmas for lifting an execution into a prefix and suffix of instructions; only needed for the first proof.

```
lemma app_right_1:
  is1 ⊢ ⟨s1,i1⟩ -1→ ⟨s2,i2⟩ ⟹ is1 @ is2 ⊢ ⟨s1,i1⟩ -1→ ⟨s2,i2⟩
  (is ?P ⟹ _)
proof -
 assume ?P
 then show ?thesis
 by induct force+
qed

lemma app_left_1:
  is2 ⊢ ⟨s1,i1⟩ -1→ ⟨s2,i2⟩ ⟹
   is1 @ is2 ⊢ ⟨s1,size is1+i1⟩ -1→ ⟨s2,size is1+i2⟩
  (is ?P ⟹ _)
proof -
 assume ?P
 then show ?thesis
 by induct force+
qed

declare rtrancl_induct2 [induct set: rtrancl]

lemma app_right:
  is1 ⊢ ⟨s1,i1⟩ -*→ ⟨s2,i2⟩ ⟹ is1 @ is2 ⊢ ⟨s1,i1⟩ -*→ ⟨s2,i2⟩
  (is ?P ⟹ _)
proof -
 assume ?P
 then show ?thesis
 proof induct
   show is1 @ is2 ⊢ ⟨s1,i1⟩ -*→ ⟨s1,i1⟩ by simp
 next
   fix s1' i1' s2 i2
   assume is1 @ is2 ⊢ ⟨s1,i1⟩ -*→ ⟨s1',i1'⟩
          is1 ⊢ ⟨s1',i1'⟩ -1→ ⟨s2,i2⟩
   thus is1 @ is2 ⊢ ⟨s1,i1⟩ -*→ ⟨s2,i2⟩
     by(blast intro:app_right_1 rtrancl_trans)
 qed
qed

lemma app_left:
  is2 ⊢ ⟨s1,i1⟩ -*→ ⟨s2,i2⟩ ⟹
   is1 @ is2 ⊢ ⟨s1,size is1+i1⟩ -*→ ⟨s2,size is1+i2⟩
  (is ?P ⟹ _)
proof -
 assume ?P
 then show ?thesis
```

```
 proof induct
   show is1 @ is2 ⊢ ⟨s1,length is1 + i1⟩ -*→ ⟨s1,length is1 + i1⟩ by simp
 next
   fix s1' i1' s2 i2
   assume is1 @ is2 ⊢ ⟨s1,length is1 + i1⟩ -*→ ⟨s1',length is1 + i1'⟩
          is2 ⊢ ⟨s1',i1'⟩ -1→ ⟨s2,i2⟩
   thus is1 @ is2 ⊢ ⟨s1,length is1 + i1⟩ -*→ ⟨s2,length is1 + i2⟩
     by(blast intro:app_left_1 rtrancl_trans)
 qed
qed

lemma app_left2:
  ⟦ is2 ⊢ ⟨s1,i1⟩ -*→ ⟨s2,i2⟩; j1 = size is1+i1; j2 = size is1+i2 ⟧ ⟹
   is1 @ is2 ⊢ ⟨s1,j1⟩ -*→ ⟨s2,j2⟩
  by (simp add:app_left)

lemma app1_left:
  is ⊢ ⟨s1,i1⟩ -*→ ⟨s2,i2⟩ ⟹
   instr # is ⊢ ⟨s1,Suc i1⟩ -*→ ⟨s2,Suc i2⟩
  by(erule app_left[of _ _ _ _ _ [instr],simplified])
```

4.4 Compiler correctness

```
declare rtrancl_into_rtrancl[trans]
        rtrancl_into_rtrancl2[trans]
        rtrancl_trans[trans]
```

The first proof; The statement is very intuitive, but application of induction hypothesis requires the above lifting lemmas

```
theorem ⟨c,s⟩ -c→ t ⟹ compile c ⊢ ⟨s,0⟩ -*→ ⟨t,length(compile c)⟩
        (is ?P ⟹ ?Q c s t)
proof -
  assume ?P
  then show ?thesis
  proof induct
    show ⋀s. ?Q skip s s by simp
  next
    show ⋀a s x. ?Q (x :== a) s (s[x↦ a s]) by force
  next
    fix c0 c1 s0 s1 s2
    assume ?Q c0 s0 s1
    hence compile c0 @ compile c1 ⊢ ⟨s0,0⟩ -*→ ⟨s1,length(compile c0)⟩
      by(rule app_right)
    moreover assume ?Q c1 s1 s2
    hence compile c0 @ compile c1 ⊢ ⟨s1,length(compile c0)⟩ -*→
          ⟨s2,length(compile c0)+length(compile c1)⟩
    proof -
      note app_left[of _ 0]
```

```
    thus
            ⋀is1 is2 s1 s2 i2.
            is2 ⊢ ⟨s1,0⟩ -*→ ⟨s2,i2⟩ ⟹
            is1 @ is2 ⊢ ⟨s1,size is1⟩ -*→ ⟨s2,size is1+i2⟩
            by simp
  qed
  ultimately have compile c0 @ compile c1 ⊢ ⟨s0,0⟩ -*→
                    ⟨s2,length(compile c0)+length(compile c1)⟩
    by (rule rtrancl_trans)
  thus ?Q (c0; c1) s0 s2 by simp
next
  fix b c0 c1 s0 s1
  let ?comp = compile(if b then c0 else c1)
  assume b s0 and IH: ?Q c0 s0 s1
  hence ?comp ⊢ ⟨s0,0⟩ -1→ ⟨s0,1⟩ by auto
  also from IH
  have ?comp ⊢ ⟨s0,1⟩ -*→ ⟨s1,length(compile c0)+1⟩
    by(auto intro:app1_left app_right)
  also have ?comp ⊢ ⟨s1,length(compile c0)+1⟩ -1→ ⟨s1,length ?comp⟩
    by(auto)
  finally show ?Q (if b then c0 else c1) s0 s1 .
next
  fix b c0 c1 s0 s1
  let ?comp = compile(if b then c0 else c1)
  assume ¬b s0 and IH: ?Q c1 s0 s1
  hence ?comp ⊢ ⟨s0,0⟩ -1→ ⟨s0,length(compile c0) + 2⟩ by auto
  also from IH
  have ?comp ⊢ ⟨s0,length(compile c0)+2⟩ -*→ ⟨s1,length ?comp⟩
    by(force intro!:app_left2 app1_left)
  finally show ?Q (if b then c0 else c1) s0 s1 .
next
  fix b c and s::state
  assume ¬b s
  thus ?Q (while b do c) s s by force
next
  fix b c and s0::state and s1 s2
  let ?comp = compile(while b do c)
  assume b s0 and
    IHc: ?Q c s0 s1 and IHw: ?Q (while b do c) s1 s2
  hence ?comp ⊢ ⟨s0,0⟩ -1→ ⟨s0,1⟩ by auto
  also from IHc
  have ?comp ⊢ ⟨s0,1⟩ -*→ ⟨s1,length(compile c)+1⟩
    by(auto intro:app1_left app_right)
  also have ?comp ⊢ ⟨s1,length(compile c)+1⟩ -1→ ⟨s1,0⟩ by simp
  also note IHw
  finally show ?Q (while b do c) s0 s2.
```

qed
qed

Second proof; statement is generalized to cater for prefixes and suffixes; needs none of the lifting lemmas, but instantiations of pre/suffix.

```
theorem ⟨c,s⟩ -c→ t ⟹
  !a z. a@compile c@z ⊢ ⟨s,length a⟩ -*→ ⟨t,length a + length(compile c)⟩
apply(erule evalc.induct)
      apply simp
     apply(force intro!: ASIN)
    apply(intro strip)
    apply(erule_tac x = a in allE)
    apply(erule_tac x = a@compile c0 in allE)
    apply(erule_tac x = compile c1@z in allE)
    apply(erule_tac x = z in allE)
    apply(simp add:add_assoc[THEN sym])
    apply(blast intro:rtrancl_trans)

   apply(intro strip)

   apply(erule_tac x = a@[?I] in allE)
   apply(simp)

   apply(rule rtrancl_into_rtrancl2)
    apply(force intro!: JMPFT)

   apply(rule rtrancl_trans)
    apply(erule allE)
    apply assumption

   apply(rule r_into_rtrancl)
   apply(force intro!: JMPFF)

  apply(intro strip)
  apply(erule_tac x = a@[?I]@compile c0@[?J] in allE)
  apply(simp add:add_assoc)
  apply(rule rtrancl_into_rtrancl2)
   apply(force intro!: JMPFF)
  apply(blast)
 apply(force intro: JMPFF)
apply(intro strip)
apply(erule_tac x = a@[?I] in allE)
apply(erule_tac x = a in allE)
apply(simp)
apply(rule rtrancl_into_rtrancl2)
 apply(force intro!: JMPFT)
```

```
apply(rule rtrancl_trans)
 apply(erule allE)
 apply assumption
apply(rule rtrancl_into_rtrancl2)
 apply(force intro!: JMPB)
apply(simp)
done
```

Missing: the other direction!

end