

IMP — A WHILE-language and its Semantics

Gerwin Klein, Heiko Loetzbeyer, Tobias Nipkow, Robert Sandner

June 8, 2008

Abstract

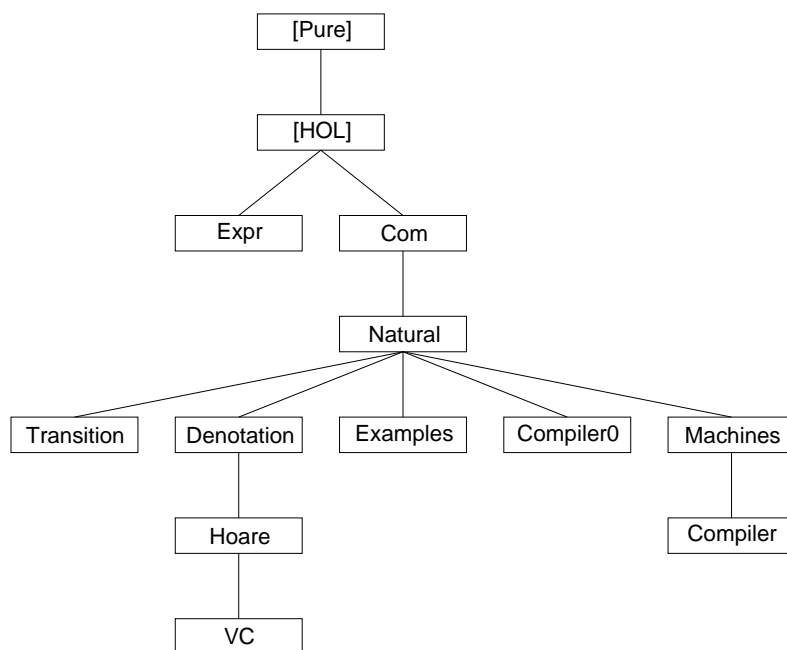
The denotational, operational, and axiomatic semantics, a verification condition generator, and all the necessary soundness, completeness and equivalence proofs. Essentially a formalization of the first 100 pages of [3].

An eminently readable description of this theory is found in [2]. See also HOLCF/IMP for a denotational semantics.

Contents

1	Expressions	3
1.1	Arithmetic expressions	3
1.2	Evaluation of arithmetic expressions	3
1.3	Boolean expressions	3
1.4	Evaluation of boolean expressions	3
1.5	Denotational semantics of arithmetic and boolean expressions	4
2	Syntax of Commands	5
3	Natural Semantics of Commands	6
3.1	Execution of commands	6
3.2	Equivalence of statements	8
3.3	Execution is deterministic	8
4	Transition Semantics of Commands	9
4.1	The transition relation	9
4.2	Examples	10
4.3	Basic properties	10
4.4	Equivalence to natural semantics (after Nielson and Nielson)	11
4.5	Winskel's Proof	11
4.6	A proof without n	12
5	Denotational Semantics of Commands	13

6	Inductive Definition of Hoare Logic	15
7	Verification Conditions	16
8	Examples	18
8.1	An example due to Tony Hoare	18
8.2	Factorial	18
9	A Simple Compiler	19
9.1	An abstract, simplistic machine	19
9.2	The compiler	19
9.3	Context lifting lemmas	20
9.4	Compiler correctness	20
9.5	Instructions	21
9.6	M0 with PC	22
9.7	M0 with lists	22
9.8	The compiler	24
9.9	Compiler correctness	24



1 Expressions

theory *Expr* **imports** *Main* **begin**

Arithmetic expressions and Boolean expressions. Not used in the rest of the language, but included for completeness.

1.1 Arithmetic expressions

typeddecl *loc*

types

state = "*loc* => *nat*"

datatype

aexp = *N nat*
| *X loc*
| *Op1 "nat => nat" aexp*
| *Op2 "nat => nat => nat" aexp aexp*

1.2 Evaluation of arithmetic expressions

inductive

evala :: "[*aexp***state*,*nat*] => *bool*" (infixl "-a->" 50)

where

N: "*N*(*n*),*s*) -a-> *n*"
| *X*: "*X*(*x*),*s*) -a-> *s*(*x*)"
| *Op1*: "*e*,*s*) -a-> *n* ==> (*Op1 f e*,*s*) -a-> *f*(*n*)"
| *Op2*: "[| (*e0*,*s*) -a-> *n0*; (*e1*,*s*) -a-> *n1* |]
==> (*Op2 f e0 e1*,*s*) -a-> *f n0 n1*"

lemmas [*intro*] = *N X Op1 Op2*

1.3 Boolean expressions

datatype

bexp = *true*
| *false*
| *ROp "nat => nat => bool" aexp aexp*
| *noti bexp*
| *andi bexp bexp* (infixl "*andi*" 60)
| *ori bexp bexp* (infixl "*ori*" 60)

1.4 Evaluation of boolean expressions

inductive

evalb :: "[*bexp***state*,*bool*] => *bool*" (infixl "-b->" 50)

— avoid clash with ML constructors *true*, *false*

where

tru: "*(true,s)* -b-> *True*"

```

/ fls:    "(false,s) -b-> False"
/ ROp:    "[| (a0,s) -a-> n0; (a1,s) -a-> n1 |]
          ==> (ROp f a0 a1,s) -b-> f n0 n1"
/ noti:   "(b,s) -b-> w ==> (noti(b),s) -b-> (~w)"
/ andi:   "[| (b0,s) -b-> w0; (b1,s) -b-> w1 |]
          ==> (b0 andi b1,s) -b-> (w0 & w1)"
/ ori:    "[| (b0,s) -b-> w0; (b1,s) -b-> w1 |]
          ==> (b0 ori b1,s) -b-> (w0 | w1)"

```

```
lemmas [intro] = tru fls ROp noti andi ori
```

1.5 Denotational semantics of arithmetic and boolean expressions

consts

```

A    :: "aexp => state => nat"
B    :: "bexp => state => bool"

```

primrec

```

"A(N(n)) = (%s. n)"
"A(X(x)) = (%s. s(x))"
"A(Op1 f a) = (%s. f(A a s))"
"A(Op2 f a0 a1) = (%s. f (A a0 s) (A a1 s))"

```

primrec

```

"B(true) = (%s. True)"
"B(false) = (%s. False)"
"B(ROp f a0 a1) = (%s. f (A a0 s) (A a1 s))"
"B(noti(b)) = (%s. ~(B b s))"
"B(b0 andi b1) = (%s. (B b0 s) & (B b1 s))"
"B(b0 ori b1) = (%s. (B b0 s) | (B b1 s))"

```

```

lemma [simp]: "(N(n),s) -a-> n' = (n = n')"
<proof>

```

```

lemma [simp]: "(X(x),sigma) -a-> i = (i = sigma x)"
<proof>

```

```

lemma [simp]:
  "(Op1 f e,sigma) -a-> i = (∃n. i = f n ∧ (e,sigma) -a-> n)"
<proof>

```

```

lemma [simp]:
  "(Op2 f a1 a2,sigma) -a-> i =
    (∃n0 n1. i = f n0 n1 ∧ (a1, sigma) -a-> n0 ∧ (a2, sigma) -a-> n1)"
<proof>

```

```

lemma [simp]: "((true,sigma) -b-> w) = (w=True)"
<proof>

```

```

lemma [simp]:

```

```

"((false,sigma) -b-> w) = (w=False)"
⟨proof⟩

lemma [simp]:
  "((ROP f a0 a1,sigma) -b-> w) =
  (? m. (a0,sigma) -a-> m & (? n. (a1,sigma) -a-> n & w = f m n))"
  ⟨proof⟩

lemma [simp]:
  "((noti(b),sigma) -b-> w) = (? x. (b,sigma) -b-> x & w = (~x))"
  ⟨proof⟩

lemma [simp]:
  "((b0 andi b1,sigma) -b-> w) =
  (? x. (b0,sigma) -b-> x & (? y. (b1,sigma) -b-> y & w = (x&y)))"
  ⟨proof⟩

lemma [simp]:
  "((b0 ori b1,sigma) -b-> w) =
  (? x. (b0,sigma) -b-> x & (? y. (b1,sigma) -b-> y & w = (x|y)))"
  ⟨proof⟩

lemma aexp_iff: "((a,s) -a-> n) = (A a s = n)"
  ⟨proof⟩

lemma bexp_iff:
  "((b,s) -b-> w) = (B b s = w)"
  ⟨proof⟩

end

```

2 Syntax of Commands

theory *Com* imports *Main* begin

typedecl *loc*

— an unspecified (arbitrary) type of locations (addresses/names) for variables

types

val = *nat* — or anything else, *nat* used in examples

state = "*loc* \Rightarrow *val*"

aexp = "*state* \Rightarrow *val*"

bexp = "*state* \Rightarrow *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)

syntax (latex)
  SKIP :: com    ("skip")
  Cond :: "bexp  $\Rightarrow$  com  $\Rightarrow$  com  $\Rightarrow$  com" ("if _ then _ else _" 60)
  While :: "bexp  $\Rightarrow$  com  $\Rightarrow$  com" ("while _ do _" 60)

end

```

3 Natural Semantics of Commands

theory Natural imports Com begin

3.1 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* :

```

constdefs
  update :: "('a  $\Rightarrow$  'b)  $\Rightarrow$  'a  $\Rightarrow$  'b  $\Rightarrow$  ('a  $\Rightarrow$  'b)" ("_/_ ::= /_" [900,0,0] 900)
  "update == fun_upd"

```

```

syntax (xsymbols)
  update :: "('a  $\Rightarrow$  'b)  $\Rightarrow$  'a  $\Rightarrow$  'b  $\Rightarrow$  ('a  $\Rightarrow$  'b)" ("_/_  $\mapsto$  /_" [900,0,0] 900)

```

The big-step execution relation evalc is defined inductively:

```

inductive
  evalc :: "[com, state, state]  $\Rightarrow$  bool" ("<_,_>/_  $\longrightarrow_c$  _" [0,0,60] 60)
where
  Skip:      "<skip, s>  $\longrightarrow_c$  s"
  | Assign:  "<x ::= a, s>  $\longrightarrow_c$  s[x  $\mapsto$  a s]"

  | Semi:    "<c0, s>  $\longrightarrow_c$  s''  $\Rightarrow$  <c1, s''>  $\longrightarrow_c$  s'  $\Rightarrow$  <c0; c1, s>  $\longrightarrow_c$  s'"

  | IfTrue:  "b s  $\Rightarrow$  <c0, s>  $\longrightarrow_c$  s'  $\Rightarrow$  <if b then c0 else c1, s>  $\longrightarrow_c$  s'"
  | IfFalse: "~b s  $\Rightarrow$  <c1, s>  $\longrightarrow_c$  s'  $\Rightarrow$  <if b then c0 else c1, s>  $\longrightarrow_c$  s'"

  | WhileFalse: "~b s  $\Rightarrow$  <while b do c, s>  $\longrightarrow_c$  s"
  | WhileTrue:  "b s  $\Rightarrow$  <c, s>  $\longrightarrow_c$  s''  $\Rightarrow$  <while b do c, s''>  $\longrightarrow_c$  s'
                  $\Rightarrow$  <while b do c, s>  $\longrightarrow_c$  s'"

```

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

The induction principle induced by this definition looks like this:

$$\begin{aligned}
& \llbracket \langle x1, x2 \rangle \longrightarrow_c x3; \bigwedge s. P \text{ skip } s \ s; \bigwedge x \ a \ s. P \ (x ::= a) \ s \ (s[x \mapsto a] \ s) \rrbracket; \\
& \bigwedge c0 \ s \ s'' \ c1 \ s'. \\
& \quad \llbracket \langle c0, s \rangle \longrightarrow_c s''; P \ c0 \ s \ s''; \langle c1, s'' \rangle \longrightarrow_c s'; P \ c1 \ s'' \ s' \rrbracket \\
& \quad \implies P \ (c0; c1) \ s \ s'; \\
& \bigwedge b \ s \ c0 \ s' \ c1. \llbracket b \ s; \langle c0, s \rangle \longrightarrow_c s'; P \ c0 \ s \ s' \rrbracket \implies P \ (\text{if } b \text{ then } c0 \text{ else } c1) \ s \ s'; \\
& \bigwedge b \ s \ c1 \ s' \ c0. \llbracket \neg b \ s; \langle c1, s \rangle \longrightarrow_c s'; P \ c1 \ s \ s' \rrbracket \implies P \ (\text{if } b \text{ then } c0 \text{ else } c1) \ s \ s'; \\
& \bigwedge b \ s \ c. \neg b \ s \implies P \ (\text{while } b \text{ do } c) \ s \ s; \\
& \bigwedge b \ s \ c \ s'' \ s'. \\
& \quad \llbracket b \ s; \langle c, s \rangle \longrightarrow_c s''; P \ c \ s \ s''; \langle \text{while } b \text{ do } c, s'' \rangle \longrightarrow_c s'; \\
& \quad P \ (\text{while } b \text{ do } c) \ s'' \ s' \rrbracket \\
& \quad \implies P \ (\text{while } b \text{ do } c) \ s \ s' \\
& \implies P \ x1 \ x2 \ x3
\end{aligned}$$

(\bigwedge and \implies 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:

" $\langle \text{skip}, s \rangle \longrightarrow_c s' = (s' = s)$ "
 $\langle \text{proof} \rangle$

lemma assign:

" $\langle x ::= a, s \rangle \longrightarrow_c s' = (s' = s[x \mapsto a])$ "
 $\langle \text{proof} \rangle$

lemma semi:

" $\langle c0; c1, s \rangle \longrightarrow_c s' = (\exists s''. \langle c0, s \rangle \longrightarrow_c s'' \wedge \langle c1, s'' \rangle \longrightarrow_c s')$ "
 $\langle \text{proof} \rangle$

lemma ifTrue:

" $b \ s \implies \langle \text{if } b \text{ then } c0 \text{ else } c1, s \rangle \longrightarrow_c s' = \langle c0, s \rangle \longrightarrow_c s'$ "
 $\langle \text{proof} \rangle$

lemma ifFalse:

" $\neg b \ s \implies \langle \text{if } b \text{ then } c0 \text{ else } c1, s \rangle \longrightarrow_c s' = \langle c1, s \rangle \longrightarrow_c s'$ "
 $\langle \text{proof} \rangle$

lemma whileFalse:

" $\neg b \ s \implies \langle \text{while } b \text{ do } c, s \rangle \longrightarrow_c s' = (s' = s)$ "
 $\langle \text{proof} \rangle$

lemma whileTrue:

" $b \ s \implies$
 $\langle \text{while } b \text{ do } c, s \rangle \longrightarrow_c s' =$
 $(\exists s''. \langle c, s \rangle \longrightarrow_c s'' \wedge \langle \text{while } b \text{ do } c, s'' \rangle \longrightarrow_c s')$ "
 $\langle \text{proof} \rangle$

Again, Isabelle may use these rules in automatic proofs:

```
lemmas evalc_cases [simp] = skip assign ifTrue ifFalse whileFalse semi whileTrue
```

3.2 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  $\Rightarrow$  com  $\Rightarrow$  bool" ("_  $\sim$  _")
  "c  $\sim$  c'  $\equiv \forall s s'. \langle c, s \rangle \longrightarrow_c s' = \langle c', s \rangle \longrightarrow_c s'"$ 
```

Proof rules telling Isabelle to unfold the definition if there is something to be proved about equivalent statements:

```
lemma equivI [intro!]:
  "( $\bigwedge s s'. \langle c, s \rangle \longrightarrow_c s' = \langle c', s \rangle \longrightarrow_c s'$ )  $\implies c \sim c'$ "
  <proof>
```

```
lemma equivD1:
  "c  $\sim$  c'  $\implies \langle c, s \rangle \longrightarrow_c s' \implies \langle c', s \rangle \longrightarrow_c s'"$ 
  <proof>
```

```
lemma equivD2:
  "c  $\sim$  c'  $\implies \langle c', s \rangle \longrightarrow_c s' \implies \langle c, s \rangle \longrightarrow_c s'"$ 
  <proof>
```

As an example, we show that loop unfolding is an equivalence transformation on programs:

```
lemma unfold_while:
  "(while b do c)  $\sim$  (if b then c; while b do c else skip)" (is "?w  $\sim$  ?if")
  <proof>
```

3.3 Execution is deterministic

The following proof presents all the details:

```
theorem com_det:
  assumes " $\langle c, s \rangle \longrightarrow_c t$ " and " $\langle c, s \rangle \longrightarrow_c u$ "
  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 " $\langle c, s \rangle \longrightarrow_c t$ " and " $\langle c, s \rangle \longrightarrow_c u$ "
  shows "u = t"
  <proof>
```

```
end
```


4 Transition Semantics of Commands

`theory Transition imports Natural begin`

4.1 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 $((com \times state) \times com \times state) \text{ set}$ but instead: $((com \text{ option} \times state) \times com \text{ option} \times state) \text{ set}$

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" ("<_>")
```

```
syntax (xsymbols)
  "_angle" :: "[com, state] ⇒ com option × state" ("⟨_,_⟩")
  "_angle2" :: "state ⇒ com option × state" ("⟨_⟩")
```

```
syntax (HTML output)
  "_angle" :: "[com, state] ⇒ com option × state" ("⟨_,_⟩")
  "_angle2" :: "state ⇒ com option × state" ("⟨_⟩")
```

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

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

```
inductive_set
  evalc1 :: "((com option × state) × (com option × state)) set"
  and evalc1' :: "[com option × state, com option × state] ⇒ bool"
  ("_ →1 _" [60,60] 61)
where
  "cs →1 cs' == (cs,cs') ∈ evalc1"
| Skip:    "⟨skip, s⟩ →1 ⟨s⟩"
| Assign:  "⟨x ::= a, s⟩ →1 ⟨s[x ↦ a]⟩"

| Semi1:   "⟨c0,s⟩ →1 ⟨s'⟩ ⇒ ⟨c0;c1,s⟩ →1 ⟨c1,s'⟩"
| Semi2:   "⟨c0,s⟩ →1 ⟨c0',s'⟩ ⇒ ⟨c0;c1,s⟩ →1 ⟨c0';c1,s'⟩"

| IfTrue:  "b s ⇒ ⟨if b then c1 else c2,s⟩ →1 ⟨c1,s⟩"
| IfFalse: "¬b s ⇒ ⟨if b then c1 else c2,s⟩ →1 ⟨c2,s⟩"

| While:   "⟨while b do c,s⟩ →1 ⟨if b then c; while b do c else skip,s⟩"
```

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

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

abbreviation

```
evalcn :: "[com option × state), nat, (com option × state)] ⇒ bool"
  ("_ ->_1 _" [60,60,60] 60) where
  "cs ->_1 cs' == (cs, cs') ∈ evalc1^n"
```

abbreviation

```
evalc' :: "[com option × state), (com option × state)] ⇒ bool"
  ("_ ->_1^* _" [60,60] 60) where
  "cs ->_1^* cs' == (cs, cs') ∈ evalc1^*"
⟨proof⟩⟨proof⟩
```

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

```
lemma SKIP_1: "⟨skip, s⟩ ->_1 y = (y = ⟨s⟩)"
  ⟨proof⟩
```

```
lemma Assign_1: "⟨x := a, s⟩ ->_1 y = (y = ⟨s[x ↦ a s]⟩)"
  ⟨proof⟩
```

```
lemma Cond_1:
  "⟨if b then c1 else c2, s⟩ ->_1 y = ((b s -> y = ⟨c1, s⟩) ∧ (¬b s -> y = ⟨c2, s⟩))"
  ⟨proof⟩
```

```
lemma While_1:
  "⟨while b do c, s⟩ ->_1 y = (y = ⟨if b then c; while b do c else skip, s⟩)"
  ⟨proof⟩
```

```
lemmas [simp] = SKIP_1 Assign_1 Cond_1 While_1
```

4.2 Examples

```
lemma
  "s x = 0 ⇒ ⟨while λs. s x ≠ 1 do (x := λs. s x + 1), s⟩ ->_1^* ⟨s[x ↦ 1]⟩"
  (is "_ ⇒ ⟨?w, _⟩ ->_1^* _")
  ⟨proof⟩
```

```
lemma
  "s x = 2 ⇒ ⟨while λs. s x ≠ 1 do (x := λs. s x + 1), s⟩ ->_1^* s'"
  (is "_ ⇒ ⟨?w, _⟩ ->_1^* s'")
  ⟨proof⟩
```

4.3 Basic properties

There are no *stuck* programs:

```
lemma no_stuck: "∃ y. ⟨c, s⟩ ->_1 y"
  ⟨proof⟩
```

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

```
lemma stuck [elim!]: "⟨s⟩ ->_1 y ⇒ P"
```

$\langle proof \rangle$

lemma *evalc_None_retranc1* [*simp*, *dest!*]: " $\langle s \rangle \rightarrow_1^* s' \implies s' = \langle s \rangle$ "
 $\langle proof \rangle \langle proof \rangle \langle proof \rangle$ **lemma** *evalc1_None_0* [*simp*]: " $\langle s \rangle \rightarrow_{n=1} y = (n = 0 \wedge y = \langle s \rangle)$ "
 $\langle proof \rangle$

lemma *SKIP_n*: " $\langle skip, s \rangle \rightarrow_{n=1} \langle s' \rangle \implies s' = s \wedge n=1$ "
 $\langle proof \rangle$

4.4 Equivalence to natural semantics (after Nielson and Nielson)

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

lemma *semiD*:
" $\langle c1; c2, s \rangle \rightarrow_{n=1} \langle s'' \rangle \implies$
 $\exists i j s'. \langle c1, s \rangle \rightarrow_{i=1} \langle s' \rangle \wedge \langle c2, s' \rangle \rightarrow_{j=1} \langle s'' \rangle \wedge n = i+j$ "
 $\langle proof \rangle$

lemma *semiI*:
" $\langle c0, s \rangle \rightarrow_{n=1} \langle s'' \rangle \implies \langle c1, s'' \rangle \rightarrow_1^* \langle s' \rangle \implies \langle c0; c1, s \rangle \rightarrow_1^* \langle s' \rangle$ "
 $\langle proof \rangle$

The easy direction of the equivalence proof:

lemma *evalc_imp_evalc1*:
assumes " $\langle c, s \rangle \rightarrow_c s'$ "
shows " $\langle c, s \rangle \rightarrow_1^* \langle s' \rangle$ "
 $\langle proof \rangle$

Finally, the equivalence theorem:

theorem *evalc_equiv_evalc1*:
" $\langle c, s \rangle \rightarrow_c s' = \langle c, s \rangle \rightarrow_1^* \langle s' \rangle$ "
 $\langle proof \rangle$

4.5 Winskel's Proof

declare *rel_pow_0_E* [*elim!*]

Winskel's small step rules are a bit different [3]; we introduce their equivalents as derived rules:

lemma *whileFalse1* [*intro*]:
" $\neg b \ s \implies \langle while \ b \ do \ c, s \rangle \rightarrow_1^* \langle s \rangle$ " (**is** " $_ \implies \langle ?w, s \rangle \rightarrow_1^* \langle s \rangle$ ")
 $\langle proof \rangle$

lemma *whileTrue1* [*intro*]:
" $b \ s \implies \langle while \ b \ do \ c, s \rangle \rightarrow_1^* \langle c; while \ b \ do \ c, s \rangle$ "
(**is** " $_ \implies \langle ?w, s \rangle \rightarrow_1^* \langle c; ?w, s \rangle$ ")
 $\langle proof \rangle$

```

inductive_cases evalc1_SEs:
  "<skip,s> →1 (co, s')"
  "<x:=a,s> →1 (co, s')"
  "<c1;c2, s> →1 (co, s')"
  "<if b then c1 else c2, s> →1 (co, s')"
  "<while b do c, s> →1 (co, s')"

inductive_cases evalc1_E: "<while b do c, s> →1 (co, s')"

declare evalc1_SEs [elim!]

lemma evalc_impl_evalc1: "<c,s> →c s1 ⇒ <c,s> →1* <s1>"
  <proof>

lemma lemma2:
  "<c;d,s> -n→1 <u> ⇒ ∃ t m. <c,s> →1* <t> ∧ <d,t> -m→1 <u> ∧ m ≤ n"
  <proof>

lemma evalc1_impl_evalc:
  "<c,s> →1* <t> ⇒ <c,s> →c t"
  <proof>

proof of the equivalence of evalc and evalc1

lemma evalc1_eq_evalc: "<<c, s> →1* <t>> = (<c,s> →c t)"
  <proof>

```

4.6 A proof without n

The inductions are a bit awkward to write in this section, because *None* as result statement in the small step semantics doesn't have a direct counterpart in the big step semantics. Winskel's small step rule set (using the *skip* statement to indicate termination) is better suited for this proof.

```

lemma my_lemma1:
  assumes "<c1,s1> →1* <s2>"
  and "<c2,s2> →1* cs3"
  shows "<c1;c2,s1> →1* cs3"
  <proof>

lemma evalc_impl_evalc1': "<c,s> →c s1 ⇒ <c,s> →1* <s1>"
  <proof>

```

The opposite direction is based on a Coq proof done by Ranan Fraer and Yves Bertot. The following sketch is from an email by Ranan Fraer.

First we've broke it into 2 lemmas:

Lemma 1

$((c,s) \dashrightarrow (\text{SKIP},t)) \Rightarrow (\langle c,s \rangle \dashrightarrow_c t)$

This is a quick one, dealing with the cases skip, assignment and while_false.

Lemma 2

$((c,s) \dashv\vdash^* (c',s')) \wedge \langle c',s' \rangle \dashrightarrow_{c'} t \Rightarrow \langle c,s \rangle \dashrightarrow_c t$

This is proved by rule induction on the $\dashv\vdash^*$ relation and the induction step makes use of a third lemma:

Lemma 3

$((c,s) \dashrightarrow (c',s')) \wedge \langle c',s' \rangle \dashrightarrow_{c'} t \Rightarrow \langle c,s \rangle \dashrightarrow_c t$

This captures the essence of the proof, as it shows that $\langle c',s' \rangle$ behaves as the continuation of $\langle c,s \rangle$ with respect to the natural semantics.

The proof of Lemma 3 goes by rule induction on the \dashrightarrow relation, dealing with the cases sequence1, sequence2, if_true, if_false and while_true. In particular in the case (sequence1) we make use again of Lemma 1.

`inductive_cases evalc1_term_cases: " $\langle c,s \rangle \longrightarrow_1 \langle s' \rangle$ "`

`lemma FB_lemma3:`

`" $\langle c,s \rangle \longrightarrow_1 \langle c',s' \rangle \implies c \neq \text{None} \implies \langle \text{if } c' = \text{None then skip else the } c',s' \rangle \longrightarrow_c t \implies \langle \text{the } c,s \rangle \longrightarrow_c t$ "`
 `$\langle \text{proof} \rangle$`

`lemma FB_lemma2:`

`" $\langle c,s \rangle \longrightarrow_1^* \langle c',s' \rangle \implies c \neq \text{None} \implies \langle \text{if } c' = \text{None then skip else the } c',s' \rangle \longrightarrow_c t \implies \langle \text{the } c,s \rangle \longrightarrow_c t$ "`
 `$\langle \text{proof} \rangle$`

`lemma evalc1_impl_evalc': " $\langle c,s \rangle \longrightarrow_1^* \langle t \rangle \implies \langle c,s \rangle \longrightarrow_c t$ "`
 `$\langle \text{proof} \rangle$`

`end`

5 Denotational Semantics of Commands

`theory Denotation imports Natural begin`

```

types com_den = "(state × state) set"

constdefs
  Gamma :: "[bexp, com_den] => (com_den => com_den)"
  "Gamma b cd == (λphi. {(s,t). (s,t) ∈ (phi 0 cd) ∧ b s} ∪
    {(s,t). s=t ∧ ¬b s})"

consts
  C :: "com => com_den"

primrec
  C_skip:    "C skip    = Id"
  C_assign:  "C (x ::= a) = {(s,t). t = s[x↦a(s)]}"
  C_comp:    "C (c0;c1)  = C(c1) 0 C(c0)"
  C_if:      "C (if b then c1 else c2) = {(s,t). (s,t) ∈ C c1 ∧ b s} ∪
    {(s,t). (s,t) ∈ C c2 ∧ ¬b s}"
  C_while:   "C(while b do c) = lfp (Gamma b (C c))"

lemma Gamma_mono: "mono (Gamma b c)"
  ⟨proof⟩

lemma C_While_If: "C(while b do c) = C(if b then c; while b do c else skip)"
  ⟨proof⟩

lemma com1: "⟨c,s⟩ →c t ⇒ (s,t) ∈ C(c)"
  ⟨proof⟩

lemma com2: "(s,t) ∈ C(c) ⇒ ⟨c,s⟩ →c t"
  ⟨proof⟩

lemma denotational_is_natural: "(s,t) ∈ C(c) = (⟨c,s⟩ →c t)"
  ⟨proof⟩

end

```

6 Inductive Definition of Hoare Logic

theory Hoare imports Denotation begin

types assn = "state => bool"

constdefs hoare_valid :: "[assn,com,assn] => bool" ("|= {(1_)} / (_)/ {(1_)}" 50)
 "|= {P}c{Q} == !s t. (s,t) : C(c) --> P s --> Q t"

inductive

hoare :: "assn => com => assn => bool" ("|- {(1_)} / (_)/ {(1_)}" 50)

where

skip: "|- {P}skip{P}"
 | ass: "|- {%s. P(s[x↦a s])} x:=a {P}"
 | semi: "[| |- {P}c{Q}; |- {Q}d{R} |] ==> |- {P} c;d {R}"
 | If: "[| |- {%s. P s & b s}c{Q}; |- {%s. P s & ~b s}d{Q} |] ==>
 |- {P} if b then c else d {Q}"
 | While: "|- {%s. P s & b s} c {P} ==>
 |- {P} while b do c {%s. P s & ~b s}"
 | conseq: "[| !s. P' s --> P s; |- {P}c{Q}; !s. Q s --> Q' s |] ==>
 |- {P'}c{Q'}"

constdefs wp :: "com => assn => assn"
 "wp c Q = (%s. !t. (s,t) : C(c) --> Q t)"

lemma hoare_conseq1: "[| !s. P' s --> P s; |- {P}c{Q} |] ==> |- {P'}c{Q}"
 <proof>

lemma hoare_conseq2: "[| |- {P}c{Q}; !s. Q s --> Q' s |] ==> |- {P}c{Q'}"
 <proof>

lemma hoare_sound: "|- {P}c{Q} ==> |= {P}c{Q}"
 <proof>

lemma wp_SKIP: "wp skip Q = Q"
 <proof>

lemma wp_Ass: "wp (x:=a) Q = (%s. Q(s[x↦a s]))"
 <proof>

lemma wp_Semi: "wp (c;d) Q = wp c (wp d Q)"
 <proof>

lemma wp_If:
 "wp (if b then c else d) Q = (%s. (b s --> wp c Q s) & (~b s --> wp d Q s))"
 <proof>

lemma wp_While_True:

```

    "b s ==> wp (while b do c) Q s = wp (c; while b do c) Q s"
  <proof>

lemma wp_While_False: "~b s ==> wp (while b do c) Q s = Q s"
  <proof>

lemmas [simp] = wp_SKIP wp_Ass wp_Semi wp_If wp_While_True wp_While_False

lemma wp_While_if:
  "wp (while b do c) Q s = (if b s then wp (c; while b do c) Q s else Q s)"
  <proof>

lemma wp_While: "wp (while b do c) Q s =
  (s : gfp(%S.{s. if b s then wp c (%s. s:S) s else Q s}))"
  <proof>

declare C_while [simp del]

lemmas [intro!] = hoare.skip hoare.ass hoare.semi hoare.If

lemma wp_is_pre: "|- {wp c Q} c {Q}"
  <proof>

lemma hoare_relative_complete: "|= {P}c{Q} ==> |- {P}c{Q}"
  <proof>

end

```

7 Verification Conditions

```

theory VC imports Hoare begin

datatype acom = Askip
              | Aass   loc aexp
              | Asemi  acom acom
              | Aif     bexp acom acom
              | Awhile  bexp assn acom

consts
  vc :: "acom => assn => assn"
  awp :: "acom => assn => assn"
  vcawp :: "acom => assn => assn × assn"
  astrip :: "acom => com"

primrec
  "awp Askip Q = Q"
  "awp (Aass x a) Q = (λs. Q(s[x ↦ a s]))"

```



```

"awp (Asemi c d) Q = awp c (awp d Q)"
"awp (Aif b c d) Q = (λs. (b s-->awp c Q s) & (~b s-->awp d Q s))"
"awp (Awhile b I c) Q = I"

```

primrec

```

"vc Askip Q = (λs. True)"
"vc (Aass x a) Q = (λs. True)"
"vc (Asemi c d) Q = (λs. vc c (awp d Q) s & vc d Q s)"
"vc (Aif b c d) Q = (λs. vc c Q s & vc d Q s)"
"vc (Awhile b I c) Q = (λs. (I s & ~b s --> Q s) &
                             (I s & b s --> awp c I s) & vc c I s)"

```

primrec

```

"astrip Askip = SKIP"
"astrip (Aass x a) = (x==a)"
"astrip (Asemi c d) = (astrip c;astrip d)"
"astrip (Aif b c d) = (if b then astrip c else astrip d)"
"astrip (Awhile b I c) = (while b do astrip c)"

```

primrec

```

"vcawp Askip Q = (λs. True, Q)"
"vcawp (Aass x a) Q = (λs. True, λs. Q(s[x↦a s]))"
"vcawp (Asemi c d) Q = (let (vcd,wpd) = vcawp d Q;
                             (vcc,wpc) = vcawp c wpd
                             in (λs. vcc s & vcd s, wpc))"
"vcawp (Aif b c d) Q = (let (vcd,wpd) = vcawp d Q;
                             (vcc,wpc) = vcawp c Q
                             in (λs. vcc s & vcd s,
                                 λs.(b s --> wpc s) & (~b s --> wpd s)))"
"vcawp (Awhile b I c) Q = (let (vcc,wpc) = vcawp c I
                             in (λs. (I s & ~b s --> Q s) &
                                 (I s & b s --> wpc s) & vcc s, I))"

```

declare hoare.intros [intro]

lemma 1: " $!s. P s \rightarrow P s$ " *<proof>*

lemma vc_sound: " $(!s. vc c Q s) \rightarrow \vdash \{awp c Q\} \text{astrip } c \{Q\}$ " *<proof>*

lemma awp_mono [rule_format (no_asm)]:
" $!P Q. (!s. P s \rightarrow Q s) \rightarrow (!s. awp c P s \rightarrow awp c Q s)$ " *<proof>*

lemma vc_mono [rule_format (no_asm)]:
" $!P Q. (!s. P s \rightarrow Q s) \rightarrow (!s. vc c P s \rightarrow vc c Q s)$ " *<proof>*

```

lemma vc_complete: assumes der: "|- {P}c{Q}"
  shows "( $\exists$  ac. astrip ac = c & ( $\forall$  s. vc ac Q s) & ( $\forall$  s. P s  $\rightarrow$  awp ac Q s))"
  (is "? ac. ?Eq P c Q ac")
  <proof>

lemma vcawp_vc_awp: "vcawp c Q = (vc c Q, awp c Q)"
  <proof>

end

```

8 Examples

theory Examples imports Natural begin

```

constdefs
  factorial :: "loc => loc => com"
  "factorial a b == b := (%s. 1);
    while (%s. s a ~= 0) do
      (b := (%s. s b * s a); a := (%s. s a - 1))"

declare update_def [simp]

```

8.1 An example due to Tony Hoare

```

lemma lemma1:
  assumes 1: "!x. P x  $\rightarrow$  Q x"
  and 2: " $\langle w, s \rangle \rightarrow_c t$ "
  shows "w = While P c  $\Rightarrow \langle$ While Q c, t $\rangle \rightarrow_c u \Rightarrow \langle$ While Q c, s $\rangle \rightarrow_c u$ "
  <proof>

lemma lemma2 [rule_format (no_asm)]:
  "[| !x. P x  $\rightarrow$  Q x;  $\langle w, s \rangle \rightarrow_c u$  |] ==>
  !c. w = While Q c  $\rightarrow \langle$ While P c; While Q c, s $\rangle \rightarrow_c u$ "
  <proof>

lemma Hoare_example: "!x. P x  $\rightarrow$  Q x ==>
  ( $\langle$ While P c; While Q c, s $\rangle \rightarrow_c t$ ) = ( $\langle$ While Q c, s $\rangle \rightarrow_c t$ )"
  <proof>

```

8.2 Factorial

```

lemma factorial_3: "a~=b ==>
   $\langle$ factorial a b, Mem(a:=3) $\rangle \rightarrow_c$  Mem(b:=6, a:=0)"
  <proof>

```

the same in single step mode:

```

lemmas [simp del] = evalc_cases

```

```

lemma "a~b  $\implies$   $\langle$ factorial a b, Mem(a:=3) $\rangle \longrightarrow_c$  Mem(b:=6, a:=0)"
 $\langle$ proof $\rangle$ 

end

```

9 A Simple Compiler

```
theory Compiler0 imports Natural begin
```

9.1 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:

```

inductive_set
  step1 :: "instr list  $\Rightarrow$  ((state $\times$ nat)  $\times$  (state $\times$ nat))set"
  and step1' :: "[instr list, state, nat, state, nat]  $\Rightarrow$  bool"
  ("_  $\vdash$  (3<_,_>/ -1 $\rightarrow$  <_,_>)" [50,0,0,0,0] 50)
  for P :: "instr list"
where
  "P  $\vdash$  <s,m> -1 $\rightarrow$  <t,n> == ((s,m),t,n) : step1 P"
/ ASIN[simp]:
  "[[ n<size P; P!n = ASIN x a ]  $\implies$  P  $\vdash$  <s,n> -1 $\rightarrow$  <s[x $\mapsto$  a s],Suc n>"
/ JMPFT[simp,intro]:
  "[[ n<size P; P!n = JMPF b i; b s ]  $\implies$  P  $\vdash$  <s,n> -1 $\rightarrow$  <s,Suc n>"
/ JMPFF[simp,intro]:
  "[[ n<size P; P!n = JMPF b i; ~b s; m=n+i ]  $\implies$  P  $\vdash$  <s,n> -1 $\rightarrow$  <s,m>"
/ JMPB[simp]:
  "[[ n<size P; P!n = JMPB i; i <= n; j = n-i ]  $\implies$  P  $\vdash$  <s,n> -1 $\rightarrow$  <s,j>"

```

abbreviation

```

stepa :: "[instr list, state, nat, state, nat]  $\Rightarrow$  bool"
  ("_  $\vdash$  (3<_,_>/ -* $\rightarrow$  <_,_>)" [50,0,0,0,0] 50) where
  "P  $\vdash$  <s,m> -* $\rightarrow$  <t,n> == ((s,m),t,n) : ((step1 P)^*)"

```

abbreviation

```

stepan :: "[instr list, state, nat, nat, state, nat]  $\Rightarrow$  bool"
  ("_  $\vdash$  (3<_,_>/ -(_) $\rightarrow$  <_,_>)" [50,0,0,0,0,0] 50) where
  "P  $\vdash$  <s,m> -(i) $\rightarrow$  <t,n> == ((s,m),t,n) : ((step1 P)^i)"

```

9.2 The compiler

```

consts compile :: "com  $\Rightarrow$  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]

```

9.3 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:
  assumes "is1 ⊢ ⟨s1,i1⟩ -1→ ⟨s2,i2⟩"
  shows "is1 @ is2 ⊢ ⟨s1,i1⟩ -1→ ⟨s2,i2⟩"
  ⟨proof⟩

lemma app_left_1:
  assumes "is2 ⊢ ⟨s1,i1⟩ -1→ ⟨s2,i2⟩"
  shows "is1 @ is2 ⊢ ⟨s1,size is1+i1⟩ -1→ ⟨s2,size is1+i2⟩"
  ⟨proof⟩

```

```

declare rtranc1_induct2 [induct set: rtranc1]

```

```

lemma app_right:
  assumes "is1 ⊢ ⟨s1,i1⟩ -*→ ⟨s2,i2⟩"
  shows "is1 @ is2 ⊢ ⟨s1,i1⟩ -*→ ⟨s2,i2⟩"
  ⟨proof⟩

lemma app_left:
  assumes "is2 ⊢ ⟨s1,i1⟩ -*→ ⟨s2,i2⟩"
  shows "is1 @ is2 ⊢ ⟨s1,size is1+i1⟩ -*→ ⟨s2,size is1+i2⟩"
  ⟨proof⟩

lemma app_left2:
  "[ is2 ⊢ ⟨s1,i1⟩ -*→ ⟨s2,i2⟩; j1 = size is1+i1; j2 = size is1+i2 ] ==>
    is1 @ is2 ⊢ ⟨s1,j1⟩ -*→ ⟨s2,j2⟩"
  ⟨proof⟩

lemma app1_left:
  assumes "is ⊢ ⟨s1,i1⟩ -*→ ⟨s2,i2⟩"
  shows "instr # is ⊢ ⟨s1,Suc i1⟩ -*→ ⟨s2,Suc i2⟩"
  ⟨proof⟩

```

9.4 Compiler correctness

```

declare rtranc1_into_rtranc1[trans]
  converse_rtranc1_into_rtranc1[trans]

```

`rtranc1_trans[trans]`

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

theorem

assumes " $\langle c, s \rangle \longrightarrow_c t$ "

shows " $\text{compile } c \vdash \langle s, 0 \rangle \dashv\!\rightarrow \langle t, \text{length}(\text{compile } c) \rangle$ " (is "?P c s t")

<proof>

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

Missing: the other direction! I did much of it, and although the main lemma is very similar to the one in the new development, the lemmas surrounding it seemed much more complicated. In the end I gave up.

end

theory Machines imports Natural begin

lemma *rtranc1_eq*: " $R^* = \text{Id} \cup (R \circ R^*)$ "

<proof>

lemma *converse_rtranc1_eq*: " $R^* = \text{Id} \cup (R^* \circ R)$ "

<proof>

lemmas *converse_rel_powE* = *rel_pow_E2*

lemma *R_O_Rn_commute*: " $R \circ R^n = R^n \circ R$ "

<proof>

lemma *converse_in_rel_pow_eq*:

" $((x, z) \in R^n) = (n=0 \wedge z=x \vee (\exists m. n = \text{Suc } m \wedge (x, y) \in R \wedge (y, z) \in R^m))$ "

<proof>

lemma *rel_pow_plus*: " $R^{(m+n)} = R^n \circ R^m$ "

<proof>

lemma *rel_pow_plusI*: " $\llbracket (x, y) \in R^m; (y, z) \in R^n \rrbracket \implies (x, z) \in R^{(m+n)}$ "

<proof>

9.5 Instructions

There are only three instructions:

datatype *instr* = *SET loc aexp* | *JMPF bexp nat* | *JMPB nat*

types *instrs* = "*instr list*"

9.6 M0 with PC

inductive_set

```
exec01 :: "instr list  $\Rightarrow$  ((nat $\times$ state)  $\times$  (nat $\times$ state))set"
and exec01' :: "[instrs, nat, state, nat, state]  $\Rightarrow$  bool"
  ("( $\_ \vdash (1\langle \_, / \rangle) / -1 \rightarrow (1\langle \_, / \rangle)$ )" [50,0,0,0,0] 50)
for P :: "instr list"
```

where

```
"p  $\vdash \langle i, s \rangle -1 \rightarrow \langle j, t \rangle$  == ((i,s),j,t) : (exec01 p)"
/ SET: "[ n < size P; P!n = SET x a ]  $\Rightarrow$  P  $\vdash \langle n, s \rangle -1 \rightarrow \langle \text{Suc } n, s[x \mapsto a] \rangle$ "
/ JMPFT: "[ n < size P; P!n = JMPF b i; b s ]  $\Rightarrow$  P  $\vdash \langle n, s \rangle -1 \rightarrow \langle \text{Suc } n, s \rangle$ "
/ JMPFF: "[ n < size P; P!n = JMPF b i;  $\neg b$  s; m=n+i+1; m  $\leq$  size P ]
 $\Rightarrow$  P  $\vdash \langle n, s \rangle -1 \rightarrow \langle m, s \rangle$ "
/ JMPB: "[ n < size P; P!n = JMPB i; i  $\leq$  n; j = n-i ]  $\Rightarrow$  P  $\vdash \langle n, s \rangle -1 \rightarrow \langle j, s \rangle$ "
```

abbreviation

```
exec0s :: "[instrs, nat, state, nat, state]  $\Rightarrow$  bool"
  ("( $\_ \vdash (1\langle \_, / \rangle) / -\ast \rightarrow (1\langle \_, / \rangle)$ )" [50,0,0,0,0] 50) where
  "p  $\vdash \langle i, s \rangle -\ast \rightarrow \langle j, t \rangle$  == ((i,s),j,t) : (exec01 p) $^{\ast}$ "
```

abbreviation

```
exec0n :: "[instrs, nat, state, nat, nat, state]  $\Rightarrow$  bool"
  ("( $\_ \vdash (1\langle \_, / \rangle) / -\rightarrow (1\langle \_, / \rangle)$ )" [50,0,0,0,0] 50) where
  "p  $\vdash \langle i, s \rangle -n \rightarrow \langle j, t \rangle$  == ((i,s),j,t) : (exec01 p) $^n$ "
```

9.7 M0 with lists

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

types config = "instrs \times instrs \times state"

inductive_set

```
stepa1 :: "(config  $\times$  config)set"
and stepa1' :: "[instrs, instrs, state, instrs, instrs, state]  $\Rightarrow$  bool"
  ("((1( $\_ \vdash (1\langle \_, / \rangle) / -1 \rightarrow (1\langle \_, / \rangle)$ ))" 50)
```

where

```
"(p,q,s)  $-1 \rightarrow (p',q',t)$  == ((p,q,s),p',q',t) : stepa1"
/ " $\langle \text{SET } x \ a \# p, q, s \rangle -1 \rightarrow \langle p, \text{SET } x \ a \# q, s[x \mapsto a] \rangle$ "
/ " $b \ s \Rightarrow \langle \text{JMPF } b \ i \# p, q, s \rangle -1 \rightarrow \langle p, \text{JMPF } b \ i \# q, s \rangle$ "
/ "[  $\neg b \ s$ ; i  $\leq$  size p ]
 $\Rightarrow \langle \text{JMPF } b \ i \# p, q, s \rangle -1 \rightarrow \langle \text{drop } i \ p, \text{rev}(\text{take } i \ p) @ \text{JMPF } b \ i \# q, s \rangle$ "
/ "i  $\leq$  size q
 $\Rightarrow \langle \text{JMPB } i \# p, q, s \rangle -1 \rightarrow \langle \text{rev}(\text{take } i \ q) @ \text{JMPB } i \# p, \text{drop } i \ q, s \rangle$ "
```

abbreviation

```
stepa :: "[instrs, instrs, state, instrs, instrs, state]  $\Rightarrow$  bool"
  ("((1( $\_ \vdash (1\langle \_, / \rangle) / -\ast \rightarrow (1\langle \_, / \rangle)$ ))" 50) where
  "(p,q,s)  $-\ast \rightarrow (p',q',t)$  == ((p,q,s),p',q',t) : (stepa1 $^{\ast}$ )"
```

abbreviation

```

stepan :: "[instrs,instrs,state, nat, instrs,instrs,state] ⇒ bool"
  ("((1⟨_,/_,/_⟩) / _ → (1⟨_,/_,/_⟩))" 50) where
  "⟨p,q,s⟩ -i→ ⟨p',q',t⟩ == (⟨p,q,s⟩,p',q',t) : (stepa1~i)"

inductive_cases execE: "(i#is,p,s), (is',p',s')) : stepa1"

lemma exec_simp[simp]:
  "(⟨i#p,q,s⟩ -1→ ⟨p',q',t⟩) = (case i of
    SET x a ⇒ t = s[x↦ a s] ∧ p' = p ∧ q' = i#q |
    JMPF b n ⇒ t=s ∧ (if b s then p' = p ∧ q' = i#q
      else n ≤ size p ∧ p' = drop n p ∧ q' = rev(take n p) @ i # q) |
    JMPB n ⇒ n ≤ size q ∧ t=s ∧ p' = rev(take n q) @ i # p ∧ q' = drop n q)"
  ⟨proof⟩

lemma execn_simp[simp]:
  "(⟨i#p,q,s⟩ -n→ ⟨p'',q'',u⟩) =
    (n=0 ∧ p'' = i#p ∧ q'' = q ∧ u = s ∨
    (∃ m p' q' t. n = Suc m ∧
      (⟨i#p,q,s⟩ -1→ ⟨p',q',t⟩ ∧ ⟨p',q',t⟩ -m→ ⟨p'',q'',u⟩)))"
  ⟨proof⟩

lemma exec_star_simp[simp]: "(⟨i#p,q,s⟩ -*→ ⟨p'',q'',u⟩) =
  (p'' = i#p & q''=q & u=s |
  (∃ p' q' t. ⟨i#p,q,s⟩ -1→ ⟨p',q',t⟩ ∧ ⟨p',q',t⟩ -*→ ⟨p'',q'',u⟩))"
  ⟨proof⟩

declare nth_append[simp]

lemma rev_revD: "rev xs = rev ys ⇒ xs = ys"
  ⟨proof⟩

lemma [simp]: "(rev xs @ rev ys = rev zs) = (ys @ xs = zs)"
  ⟨proof⟩

lemma direction1:
  "⟨q,p,s⟩ -1→ ⟨q',p',t⟩ ⇒
    rev p' @ q' = rev p @ q ∧ rev p @ q ⊢ ⟨size p,s⟩ -1→ ⟨size p',t⟩"
  ⟨proof⟩

lemma direction2:
  "rpq ⊢ ⟨sp,s⟩ -1→ ⟨sp',t⟩ ⇒
    rpq = rev p @ q & sp = size p & sp' = size p' →
    rev p' @ q' = rev p @ q → ⟨q,p,s⟩ -1→ ⟨q',p',t⟩"
  ⟨proof⟩

theorem M_equiv:

```

```

"( $\langle q, p, s \rangle \rightarrow -1 \langle q', p', t \rangle$ ) =
  ( $\text{rev } p' @ q' = \text{rev } p @ q \wedge \text{rev } p @ q \vdash \langle \text{size } p, s \rangle \rightarrow -1 \langle \text{size } p', t \rangle$ )"
  <proof>

```

end

theory Compiler imports Machines begin

9.8 The compiler

```

consts compile :: "com  $\Rightarrow$  instr list"
primrec
  "compile skip = []"
  "compile (x:=a) = [SET x a]"
  "compile (c1;c2) = compile c1 @ compile c2"
  "compile (if b then c1 else c2) =
    [JMPF b (length(compile c1) + 1)] @ compile c1 @
    [JMPF ( $\lambda x. \text{False}$ ) (length(compile c2))] @ compile c2"
  "compile (while b do c) = [JMPF b (length(compile c) + 1)] @ compile c @
    [JMPB (length(compile c)+1)]"

```

9.9 Compiler correctness

```

theorem assumes A: " $\langle c, s \rangle \rightarrow_c t$ "
shows " $\bigwedge p q. \langle \text{compile } c @ p, q, s \rangle \rightarrow^* \langle p, \text{rev}(\text{compile } c) @ q, t \rangle$ "
  (is " $\bigwedge p q. ?P \ c \ s \ t \ p \ q$ ")
  <proof>

```

The other direction!

```

inductive_cases [elim!]: "(([], p, s), (is', p', s')) : step1"

```

```

lemma [simp]: " $\langle [], q, s \rangle \rightarrow -n \langle p', q', t \rangle = (n=0 \wedge p' = [] \wedge q' = q \wedge t = s)$ "
  <proof>

```

```

lemma [simp]: " $\langle [], q, s \rangle \rightarrow^* \langle p', q', t \rangle = (p' = [] \wedge q' = q \wedge t = s)$ "
  <proof>

```

```

constdefs
  forws :: "instr  $\Rightarrow$  nat set"
  "forws instr == case instr of
    SET x a  $\Rightarrow$  {0} |
    JMPF b n  $\Rightarrow$  {0,n} |
    JMPB n  $\Rightarrow$  {}"
  backws :: "instr  $\Rightarrow$  nat set"
  "backws instr == case instr of
    SET x a  $\Rightarrow$  {} |
    JMPF b n  $\Rightarrow$  {} |
    JMPB n  $\Rightarrow$  {n}"

```



```

consts closed :: "nat  $\Rightarrow$  nat  $\Rightarrow$  instr list  $\Rightarrow$  bool"
primrec
  "closed m n [] = True"
  "closed m n (instr#is) = (( $\forall j \in \text{forws instr. } j \leq \text{size is}+n$ )  $\wedge$ 
    ( $\forall j \in \text{backws instr. } j \leq m$ )  $\wedge$  closed (Suc m) n is)"

lemma [simp]:
  " $\bigwedge m n. \text{closed } m n (C1@C2) =$ 
    (closed m (n+size C2) C1  $\wedge$  closed (m+size C1) n C2)"
  <proof>

theorem [simp]: " $\bigwedge m n. \text{closed } m n (\text{compile } c)$ "
  <proof>

lemma drop_lem: "n  $\leq$  size(p1@p2)
   $\implies \langle p1' @ p2 = \text{drop } n \text{ } p1 @ \text{drop } (n - \text{size } p1) \text{ } p2 \rangle =$ 
    (n  $\leq$  size p1  $\wedge$  p1' = drop n p1)"
  <proof>

lemma reduce_exec1:
  " $\langle i \# p1 @ p2, q1 @ q2, s \rangle -1 \rightarrow \langle p1' @ p2, q1' @ q2, s' \rangle \implies$ 
     $\langle i \# p1, q1, s \rangle -1 \rightarrow \langle p1', q1', s' \rangle$ "
  <proof>

lemma closed_exec1:
  "[[ closed 0 0 (rev q1 @ instr # p1);
     $\langle \text{instr } \# p1 @ p2, q1 @ q2, r \rangle -1 \rightarrow \langle p', q', r' \rangle$  ]  $\implies$ 
     $\exists p1' q1'. p' = p1'@p2 \wedge q' = q1'@q2 \wedge \text{rev } q1' @ p1' = \text{rev } q1 @ \text{instr } \# p1$ ]"
  <proof>

theorem closed_execl_decomp: " $\bigwedge C1 C2 r. \llbracket \text{closed } 0 0 (\text{rev } C1 @ C2);$ 
   $\langle C2 @ p1 @ p2, C1 @ q, r \rangle -n \rightarrow \langle p2, \text{rev } p1 @ \text{rev } C2 @ C1 @ q, t \rangle \rrbracket$ 
   $\implies \exists s n1 n2. \langle C2, C1, r \rangle -n1 \rightarrow \langle [], \text{rev } C2 @ C1, s \rangle \wedge$ 
     $\langle p1@p2, \text{rev } C2 @ C1 @ q, s \rangle -n2 \rightarrow \langle p2, \text{rev } p1 @ \text{rev } C2 @ C1 @ q, t \rangle \wedge$ 
    n = n1+n2"
  (is " $\bigwedge C1 C2 r. \llbracket ?CL \ C1 \ C2; ?H \ C1 \ C2 \ r \ n \rrbracket \implies ?P \ C1 \ C2 \ r \ n$ ")
  <proof>

lemma execl_decomp:
  " $\langle \text{compile } c @ p1 @ p2, q, r \rangle -n \rightarrow \langle p2, \text{rev } p1 @ \text{rev}(\text{compile } c) @ q, t \rangle$ 
   $\implies \exists s n1 n2. \langle \text{compile } c, [], r \rangle -n1 \rightarrow \langle [], \text{rev}(\text{compile } c), s \rangle \wedge$ 
     $\langle p1@p2, \text{rev}(\text{compile } c) @ q, s \rangle -n2 \rightarrow \langle p2, \text{rev } p1 @ \text{rev}(\text{compile } c) @ q, t \rangle \wedge$ 
    n = n1+n2"
  <proof>

lemma exec_star_decomp:
  " $\langle \text{compile } c @ p1 @ p2, q, r \rangle -* \rightarrow \langle p2, \text{rev } p1 @ \text{rev}(\text{compile } c) @ q, t \rangle$ "

```

$\implies \exists s. \langle \text{compile } c, [], r \rangle \multimap \langle [], \text{rev}(\text{compile } c), s \rangle \wedge$
 $\langle p1 @ p2, \text{rev}(\text{compile } c) @ q, s \rangle \multimap \langle p2, \text{rev } p1 @ \text{rev}(\text{compile } c) @ q, t \rangle "$
 $\langle \text{proof} \rangle$

Warning: $\langle \text{compile } c @ p, q, s \rangle \multimap \langle p, \text{rev}(\text{compile } c) @ q, t \rangle \implies \langle c, s \rangle \longrightarrow_c t$ is not true!

theorem " $\bigwedge s t.$
 $\langle \text{compile } c, [], s \rangle \multimap \langle [], \text{rev}(\text{compile } c), t \rangle \implies \langle c, s \rangle \longrightarrow_c t "$
 $\langle \text{proof} \rangle$

end

References

- [1] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications*. Wiley, 1992.
- [2] Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *Lect. Notes in Comp. Sci.*, pages 180–192. Springer-Verlag, 1996.
- [3] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.