IMP — A `WHILE`-language and its Semantics

Gerwin Klein, Heiko Loetzbeyer, Tobias Nipkow, Robert Sandner

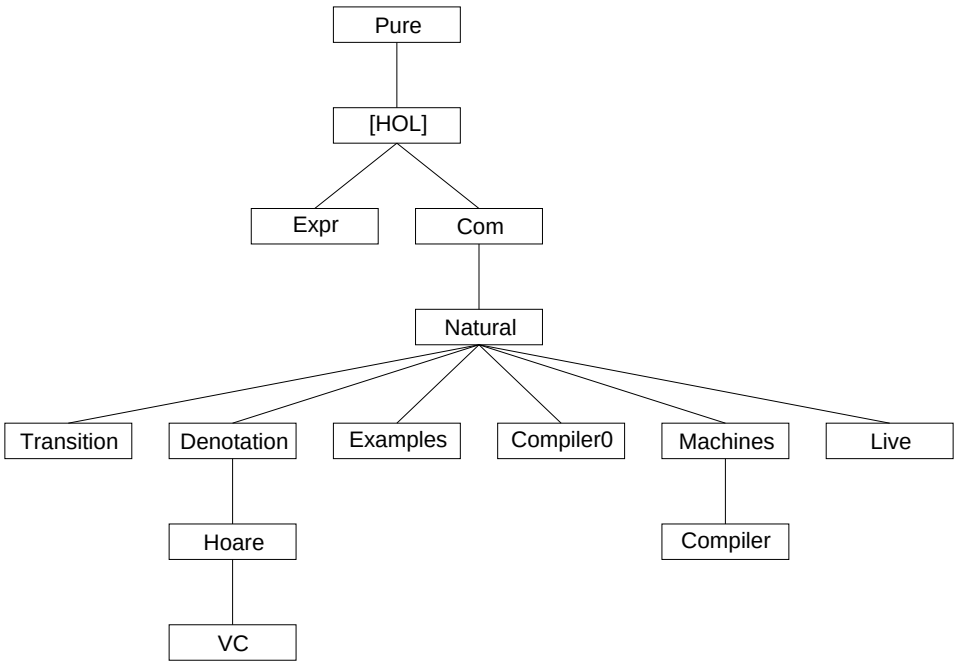
April 19, 2009

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

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

typedecl *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

```
primrec A :: "aexp => state => nat"
where
  "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 :: "bexp => state => bool"
where
  "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 (adresses/names) for variables

types
```
val   = nat — or anything else, nat used in examples
state = "loc ⇒ val"
aexp  = "state ⇒ val"
bexp  = "state ⇒ 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)
```

notation *(latex)*
  SKIP ("skip") **and**
  Cond ("if _ then _ else _" *60*) **and**
  While ("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`:

definition
  update :: "('a ⇒ 'b) ⇒ 'a ⇒ 'b ⇒ ('a ⇒ 'b)" ("_/[_ ::= /_]" [900,0,0] 900) **where**
  "update = fun_upd"

notation *(xsymbols)*
  update ("_/[_ ↦ /_]" [900,0,0] 900)

The big-step execution relation `evalc` is defined inductively:

inductive
  evalc :: "[com,state,state] ⇒ bool" ("⟨_,_⟩/ ⟶c _" [0,0,60] 60)
where
  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:

⟦⟨x1,x2⟩ ⟶c x3; ⋀s. P skip s s; ⋀x a s. P (x :== a) s (s[x ↦ a s]);

```
⋀c0 s s'' c1 s'.
   [⟨c0,s⟩ ⟶c s''; P c0 s s''; ⟨c1,s''⟩ ⟶c s'; P c1 s'' s'']
   ⟹ P (c0; c1) s s';
⋀b s c0 s' c1. [b s; ⟨c0,s⟩ ⟶c s'; P c0 s s'] ⟹ P (if b then c0 else c1) s s';
⋀b s c1 s' c0. [¬ b s; ⟨c1,s⟩ ⟶c s'; P c1 s s'] ⟹ P (if b then c0 else c1) s s';
⋀b s c. ¬ b s ⟹ P (while b do c) s s;
⋀b s c 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 x1 x2 x3
```

(⋀ 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:*
```
"⟨skip,s⟩ ⟶c s' = (s' = s)"
```
⟨*proof*⟩

lemma *assign:*
```
"⟨x :== a,s⟩ ⟶c s' = (s' = s[x↦a s])"
```
⟨*proof*⟩

lemma *semi:*
```
"⟨c0; c1, s⟩ ⟶c s' = (∃s''. ⟨c0,s⟩ ⟶c s'' ∧ ⟨c1,s''⟩ ⟶c s')"
```
⟨*proof*⟩

lemma *ifTrue:*
```
"b s ⟹ ⟨if b then c0 else c1, s⟩ ⟶c s' = ⟨c0,s⟩ ⟶c s'"
```
⟨*proof*⟩

lemma *ifFalse:*
```
"¬b s ⟹ ⟨if b then c0 else c1, s⟩ ⟶c s' = ⟨c1,s⟩ ⟶c s'"
```
⟨*proof*⟩

lemma *whileFalse:*
```
"¬ b s ⟹ ⟨while b do c,s⟩ ⟶c s' = (s' = s)"
```
⟨*proof*⟩

lemma *whileTrue:*
```
"b s ⟹
⟨while b do c, s⟩ ⟶c s' =
(∃s''. ⟨c,s⟩ ⟶c s'' ∧ ⟨while b do c, s''⟩ ⟶c s')"
```
⟨*proof*⟩

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:

definition
```
  equiv_c :: "com ⇒ com ⇒ bool" ("_ ∼ _") where
  "c ∼ c' = (∀s s'. ⟨c, s⟩ ⟶c s' = ⟨c', s⟩ ⟶c s')"
```

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

lemma *equivI [intro!]:*
```
  "(⋀s s'. ⟨c, s⟩ ⟶c s' = ⟨c', s⟩ ⟶c s') ⟹ c ∼ c'"
```
⟨*proof*⟩

lemma *equivD1:*
```
  "c ∼ c' ⟹ ⟨c, s⟩ ⟶c s' ⟹ ⟨c', s⟩ ⟶c s'"
```
⟨*proof*⟩

lemma *equivD2:*
```
  "c ∼ c' ⟹ ⟨c', s⟩ ⟶c s' ⟹ ⟨c, s⟩ ⟶c s'"
```
⟨*proof*⟩

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*⟩

3.3 Execution is deterministic

The following proof presents all the details:

theorem *com_det:*
```
  assumes "⟨c,s⟩ ⟶c t" and "⟨c,s⟩ ⟶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 "⟨c,s⟩ ⟶c t" and "⟨c,s⟩ ⟶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 × state) × com × state) set* but instead: *((com option × state) × com option × state) set*

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

abbreviation
```
  angle :: "[com, state] ⇒ com option × state" ("<_,_>") where
  "<c,s> == (Some c, s)"
```
abbreviation
```
  angle2 :: "state ⇒ com option × state"  ("<_>") where
  "<s> == (None, s)"

notation (xsymbols)
  angle  ("⟨_,_⟩") and
  angle2  ("⟨_⟩")

notation (HTML output)
  angle  ("⟨_,_⟩") and
  angle2  ("⟨_⟩")
```

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 s]⟩"

| 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 -n→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"*
⟨*proof*⟩

lemma *evalc_None_retrancl [simp, dest!]: "⟨s⟩ −→₁* s' ⟹ s' = ⟨s⟩"*

⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩**lemma** *evalc1_None_0 [simp]*: "⟨*s*⟩ -*n*→$_1$ *y* = (*n* = 0 ∧ *y* = ⟨*s*⟩)"
⟨*proof*⟩

lemma *SKIP_n*: "⟨skip, *s*⟩ -*n*→$_1$ ⟨*s'*⟩ ⟹ *s'* = *s* ∧ *n=1*"
⟨*proof*⟩

4.4 Equivalence to natural semantics (after Nielson and Nielson)

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

lemma *semiD:*
"⟨*c1; c2, s*⟩ -*n*→$_1$ ⟨*s''*⟩ ⟹
∃*i j s'*. ⟨*c1, s*⟩ -*i*→$_1$ ⟨*s'*⟩ ∧ ⟨*c2, s'*⟩ -*j*→$_1$ ⟨*s''*⟩ ∧ *n* = *i+j*"
⟨*proof*⟩

lemma *semiI:*
"⟨*c0,s*⟩ -*n*→$_1$ ⟨*s''*⟩ ⟹ ⟨*c1,s''*⟩ ⟶$_1^*$ ⟨*s'*⟩ ⟹ ⟨*c0; c1, s*⟩ ⟶$_1^*$ ⟨*s'*⟩"
⟨*proof*⟩

The easy direction of the equivalence proof:

lemma *evalc_imp_evalc1:*
assumes "⟨*c,s*⟩ ⟶$_c$ *s'*"
shows "⟨*c, s*⟩ ⟶$_1^*$ ⟨*s'*⟩"
⟨*proof*⟩

Finally, the equivalence theorem:

theorem *evalc_equiv_evalc1:*
"⟨*c, s*⟩ ⟶$_c$ *s'* = ⟨*c,s*⟩ ⟶$_1^*$ ⟨*s'*⟩"
⟨*proof*⟩

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]:*
"¬ *b s* ⟹ ⟨while *b* do *c,s*⟩ ⟶$_1^*$ ⟨*s*⟩" (**is** "_ ⟹ ⟨*?w, s*⟩ ⟶$_1^*$ ⟨*s*⟩")
⟨*proof*⟩

lemma *whileTrue1 [intro]:*
"*b s* ⟹ ⟨while *b* do *c,s*⟩ ⟶$_1^*$ ⟨*c;* while *b* do *c, s*⟩"
(**is** "_ ⟹ ⟨*?w, s*⟩ ⟶$_1^*$ ⟨*c;?w,s*⟩")
⟨*proof*⟩

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) --> (SKIP,t)) => (<c,s> -c-> t)

This is a quick one, dealing with the cases skip, assignment
```

and while_false.

Lemma 2
((c,s) -*-> (c',s')) /\ <c',s'> -c'-> t
  =>
<c,s> -c-> t

This is proved by rule induction on the -*-> relation
and the induction step makes use of a third lemma:

Lemma 3
((c,s) --> (c',s')) /\ <c',s'> -c'-> t
  =>
<c,s> -c-> t

This captures the essence of the proof, as it shows that <c',s'>
behaves as the continuation of <c,s> with respect to the natural
semantics.
The proof of Lemma 3 goes by rule induction on the --> 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:*
  "$(c,s) \longrightarrow_1 (c',s') \Longrightarrow c \neq None \Longrightarrow$
  $\langle$*if c'=None then* skip *else the c',s'*$\rangle \longrightarrow_c t \Longrightarrow \langle$*the c,s*$\rangle \longrightarrow_c t$"
  $\langle$*proof*$\rangle$

lemma *FB_lemma2:*
  "$(c,s) \longrightarrow_1^* (c',s') \Longrightarrow c \neq None \Longrightarrow$
   $\langle$*if c' = None then* skip *else the c',s'*$\rangle \longrightarrow_c t \Longrightarrow \langle$*the c,s*$\rangle \longrightarrow_c t$"
  $\langle$*proof*$\rangle$

lemma *evalc1_impl_evalc':* "$\langle c,s\rangle \longrightarrow_1^* \langle t\rangle \Longrightarrow \langle c,s\rangle \longrightarrow_c t$"
  $\langle$*proof*$\rangle$

end

5 Denotational Semantics of Commands

theory *Denotation* **imports** *Natural* **begin**

types *com_den* = "(state$\times$state)set"

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

primrec C :: "com => com_den"
where
  C_skip:   "C skip   = Id"
| C_assign: "C (x :== a) = {(s,t). t = s[x↦a(s)]}"
| C_comp:   "C (c0;c1)   = C(c1) O 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*"

definition
```
  hoare_valid :: "[assn,com,assn] => bool" ("|= {(1_)}/ (_)/ {(1_)}" 50) where
  "|= {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'}"
```

definition
```
  wp :: "com => assn => assn" where
  "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

primrec awp :: "acom => assn => assn"
where
  "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 :: "acom => assn => assn"
where
  "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 :: "acom => com"
where
  "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 :: "acom => assn => assn × assn"
where
  "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 l: "!s. P s --> P s" ⟨proof⟩

lemma vc_sound: "(!s. vc c Q s) --> |- {awp c Q} astrip c {Q}"
⟨proof⟩

lemma awp_mono [rule_format (no_asm)]:
  "!P Q. (!s. P s --> Q s) --> (!s. awp c P s --> awp c Q s)"
⟨proof⟩

lemma vc_mono [rule_format (no_asm)]:
  "!P Q. (!s. P s --> Q s) --> (!s. vc c P s --> vc c Q s)"
⟨proof⟩

lemma vc_complete: assumes der: "|- {P}c{Q}"
  shows "(∃ac. astrip ac = c & (∀s. vc ac Q s) & (∀s. P s --> 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**

definition
factorial :: "loc => loc => com" **where**
"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 ⟶ Q x"*
and *2: "⟨w,s⟩ ⟶c t"*
shows *"w = While P c ⟹ ⟨While Q c,t⟩ ⟶c u ⟹ ⟨While Q c,s⟩ ⟶c u"*
⟨*proof*⟩

lemma *lemma2 [rule_format (no_asm)]:*
"[| !x. P x ⟶ Q x; ⟨w,s⟩ ⟶c u |] ==>
!c. w = While Q c ⟶ ⟨While P c; While Q c,s⟩ ⟶c u"
⟨*proof*⟩

lemma *Hoare_example: "!x. P x ⟶ Q x ==>*
(⟨While P c; While Q c, s⟩ ⟶c t) = (⟨While Q c, s⟩ ⟶c t)"
⟨*proof*⟩

8.2 Factorial

lemma *factorial_3: "a~=b ==>*
⟨factorial a b, Mem(a:=3)⟩ ⟶c Mem(b:=6, a:=0)"
⟨*proof*⟩

the same in single step mode:

lemmas *[simp del] = evalc_cases*
lemma *"a~=b ⟹ ⟨factorial a b, Mem(a:=3)⟩ ⟶c Mem(b:=6, a:=0)"*
⟨*proof*⟩

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
  stepa1 :: "instr list ⇒ ((state×nat) × (state×nat))set"
  and *stepa1' :: "[instr list,state,nat,state,nat] ⇒ bool"*
    ("_ ⊢ (3⟨_,_⟩/ -1→ ⟨_,_⟩)" [50,0,0,0,0] 50)
  for *P :: "instr list"*
where
  "P ⊢ ⟨s,m⟩ -1→ ⟨t,n⟩ == ((s,m),t,n) : stepa1 P"
| *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⟩"

abbreviation
  stepa :: "[instr list,state,nat,state,nat] ⇒ bool"
    ("_ ⊢/ (3⟨_,_⟩/ -→ ⟨_,_⟩)" [50,0,0,0,0] 50)* **where**
  "P ⊢ ⟨s,m⟩ -→ ⟨t,n⟩ == ((s,m),t,n) : ((stepa1 P)^*)"*

abbreviation
  stepan :: "[instr list,state,nat,nat,state,nat] ⇒ bool"
    ("_ ⊢/ (3⟨_,_⟩/ -(_)→ ⟨_,_⟩)" [50,0,0,0,0,0] 50) **where**
  "P ⊢ ⟨s,m⟩ -(i)→ ⟨t,n⟩ == ((s,m),t,n) : ((stepa1 P)^i)"

9.2 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]*

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 *rtrancl_induct2 [induct set: rtrancl]*

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 *rtrancl_into_rtrancl[trans]*
        converse_rtrancl_into_rtrancl[trans]
        rtrancl_trans[trans]

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

theorem
  assumes *"⟨c,s⟩ ⟶$_c$ t"*
  shows *"compile c ⊢ ⟨s,0⟩ -*→ ⟨t,length(compile c)⟩" (***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 *rtrancl_eq: "R^* = Id ∪ (R O R^*)"*
⟨*proof*⟩

lemma *converse_rtrancl_eq: "R^* = Id ∪ (R^* O R)"*
⟨*proof*⟩

lemmas *converse_rel_powE = rel_pow_E2*

lemma *R_O_Rn_commute: "R O R^n = R^n O R"*
⟨*proof*⟩

lemma *converse_in_rel_pow_eq:*
"((x,z) ∈ R^n) = (n=0 ∧ z=x ∨ (∃m y. n = Suc m ∧ (x,y) ∈ R ∧ (y,z) ∈ R^m))"
⟨*proof*⟩

lemma *rel_pow_plus: "R^(m+n) = R^n O R^m"*
⟨*proof*⟩

lemma *rel_pow_plusI: "⟦ (x,y) ∈ R^m; (y,z) ∈ R^n ⟧ ⟹ (x,z) ∈ 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 ⇒ ((nat×state) × (nat×state))set"
and *exec01' :: "[instrs, nat,state, nat,state] ⇒ bool"*
("(_/ ⊢ (1⟨_,/_⟩)/ -1→ (1⟨_,/_⟩))" [50,0,0,0,0] 50)

```
  for P :: "instr list"
where
  "p ⊢ ⟨i,s⟩ -1→ ⟨j,t⟩ == ((i,s),j,t) : (exec01 p)"
| SET: "⟦ n<size P; P!n = SET x a ⟧ ⟹ P ⊢ ⟨n,s⟩ -1→ ⟨Suc n,s[x↦ a s]⟩"
| JMPFT: "⟦ n<size P; P!n = JMPF b i;  b s ⟧ ⟹ P ⊢ ⟨n,s⟩ -1→ ⟨Suc n,s⟩"
| JMPFF: "⟦ n<size P; P!n = JMPF b i; ¬b s; m=n+i+1; m ≤ size P ⟧
       ⟹ P ⊢ ⟨n,s⟩ -1→ ⟨m,s⟩"
| JMPB:  "⟦ n<size P; P!n = JMPB i; i ≤ n; j = n-i ⟧ ⟹ P ⊢ ⟨n,s⟩ -1→ ⟨j,s⟩"

abbreviation
  exec0s :: "[instrs, nat,state, nat,state] ⇒ bool"
    ("(_/ ⊢ (1⟨_,/_⟩)/ -*→ (1⟨_,/_⟩))" [50,0,0,0,0] 50)  where
  "p ⊢ ⟨i,s⟩ -*→ ⟨j,t⟩ == ((i,s),j,t) : (exec01 p)^*"

abbreviation
  exec0n :: "[instrs, nat,state, nat, nat,state] ⇒ bool"
    ("(_/ ⊢ (1⟨_,/_⟩)/ -_→ (1⟨_,/_⟩))" [50,0,0,0,0] 50)  where
  "p ⊢ ⟨i,s⟩ -n→ ⟨j,t⟩ == ((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 × instrs × state"


inductive_set
  stepa1 :: "(config × config)set"
  and stepa1' :: "[instrs,instrs,state, instrs,instrs,state] ⇒ bool"
    ("((1⟨_,/_,/_⟩)/ -1→ (1⟨_,/_,/_⟩))" 50)
where
  "⟨p,q,s⟩ -1→ ⟨p',q',t⟩ == ((p,q,s),p',q',t) : stepa1"
| "⟨SET x a#p,q,s⟩ -1→ ⟨p,SET x a#q,s[x↦ a s]⟩"
| "b s ⟹ ⟨JMPF b i#p,q,s⟩ -1→ ⟨p,JMPF b i#q,s⟩"
| "⟦ ¬ b s; i ≤ size p ⟧
   ⟹ ⟨JMPF b i # p, q, s⟩ -1→ ⟨drop i p, rev(take i p) @ JMPF b i # q, s⟩"
| "i ≤ size q
   ⟹ ⟨JMPB i # p, q, s⟩ -1→ ⟨rev(take i q) @ JMPB i # p, drop i q, s⟩"

abbreviation
  stepa :: "[instrs,instrs,state, instrs,instrs,state] ⇒ bool"
    ("((1⟨_,/_,/_⟩)/ -*→ (1⟨_,/_,/_⟩))" 50)  where
  "⟨p,q,s⟩ -*→ ⟨p',q',t⟩ == ((p,q,s),p',q',t) : (stepa1^*)"

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_eqiv:*
```
"(⟨q,p,s⟩ -1→ ⟨q',p',t⟩) =
 (rev p' @ q' = rev p @ q ∧ rev p @ q ⊢ ⟨size p,s⟩ -1→ ⟨size p',t⟩)"
```
⟨*proof*⟩

end

theory *Compiler* **imports** *Machines* **begin**

9.8 The compiler

primrec *compile :: "com ⇒ instr list"*
where
  "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 (λx. 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: "⟨c,s⟩ ⟶c t"*
shows *"⋀p q. ⟨compile c @ p,q,s⟩ -*⟶ ⟨p,rev(compile c)@q,t⟩"*
  *(***is** *"⋀p q. ?P c s t p q")*
⟨*proof*⟩

The other direction!

inductive_cases *[elim!]: "(([],p,s),(is',p',s')) : stepa1"*

lemma *[simp]: "(⟨[],q,s⟩ -n⟶ ⟨p',q',t⟩) = (n=0 ∧ p' = [] ∧ q' = q ∧ t = s)"*
⟨*proof*⟩

lemma *[simp]: "(⟨[],q,s⟩ -*⟶ ⟨p',q',t⟩) = (p' = [] ∧ q' = q ∧ t = s)"*
⟨*proof*⟩

definition
  forws :: "instr ⇒ nat set" **where**
  "forws instr = (case instr of
     SET x a ⇒ {0} |
     JMPF b n ⇒ {0,n} |
     JMPB n ⇒ {})"

definition
  backws :: "instr ⇒ nat set" **where**
  "backws instr = (case instr of
     SET x a ⇒ {} |
     JMPF b n ⇒ {} |
     JMPB n ⇒ {n})"

primrec *closed :: "nat ⇒ nat ⇒ instr list ⇒ bool"*
where

```
  "closed m n [] = True"
| "closed m n (instr#is) = ((∀j ∈ forws instr. j ≤ size is+n) ∧
                           (∀j ∈ backws instr. j ≤ m) ∧ closed (Suc m) n is)"
```

lemma *[simp]:*
```
 "⋀m n. closed m n (C1@C2) =
         (closed m (n+size C2) C1 ∧ closed (m+size C1) n C2)"
```
⟨*proof*⟩

theorem *[simp]:* `"⋀m n. closed m n (compile c)"`
⟨*proof*⟩

lemma *drop_lem:*
```
"n ≤ size(p1@p2)
 ⟹ (p1' @ p2 = drop n p1 @ drop (n - size p1) p2) =
    (n ≤ size p1 & p1' = drop n p1)"
```
⟨*proof*⟩

lemma *reduce_exec1:*
```
 "⟨i # p1 @ p2,q1 @ q2,s⟩ -1→ ⟨p1' @ p2,q1' @ q2,s'⟩ ⟹
  ⟨i # p1,q1,s⟩ -1→ ⟨p1',q1',s'⟩"
```
⟨*proof*⟩

lemma *closed_exec1:*
```
 "⟦ closed 0 0 (rev q1 @ instr # p1);
    ⟨instr # p1 @ p2, q1 @ q2,r⟩ -1→ ⟨p',q',r'⟩ ⟧ ⟹
  ∃p1' q1'. p' = p1'@p2 ∧ q' = q1'@q2 ∧ rev q1' @ p1' = rev q1 @ instr # p1"
```
⟨*proof*⟩

theorem *closed_execn_decomp:* `"⋀C1 C2 r.`
```
 ⟦ closed 0 0 (rev C1 @ C2);
   ⟨C2 @ p1 @ p2, C1 @ q,r⟩ -n→ ⟨p2,rev p1 @ rev C2 @ C1 @ q,t⟩ ⟧
 ⟹ ∃s n1 n2. ⟨C2,C1,r⟩ -n1→ ⟨[],rev C2 @ C1,s⟩ ∧
     ⟨p1@p2,rev C2 @ C1 @ q,s⟩ -n2→ ⟨p2, rev p1 @ rev C2 @ C1 @ q,t⟩ ∧
         n = n1+n2"
(is "⋀C1 C2 r. ⟦?CL C1 C2; ?H C1 C2 r n⟧ ⟹ ?P C1 C2 r n")
```
⟨*proof*⟩

lemma *execn_decomp:*
```
"⟨compile c @ p1 @ p2,q,r⟩ -n→ ⟨p2,rev p1 @ rev(compile c) @ q,t⟩
 ⟹ ∃s n1 n2. ⟨compile c,[],r⟩ -n1→ ⟨[],rev(compile c),s⟩ ∧
     ⟨p1@p2,rev(compile c) @ q,s⟩ -n2→ ⟨p2, rev p1 @ rev(compile c) @ q,t⟩ ∧
         n = n1+n2"
```
⟨*proof*⟩

lemma *exec_star_decomp:*
```
"⟨compile c @ p1 @ p2,q,r⟩ -*→ ⟨p2,rev p1 @ rev(compile c) @ q,t⟩
 ⟹ ∃s. ⟨compile c,[],r⟩ -*→ ⟨[],rev(compile c),s⟩ ∧
     ⟨p1@p2,rev(compile c) @ q,s⟩ -*→ ⟨p2, rev p1 @ rev(compile c) @ q,t⟩"
```
⟨*proof*⟩

Warning: ⟨*compile c @ p,q,s*⟩ *-*→* ⟨*p,rev (compile c) @ q,t*⟩ ⟹ ⟨*c,s*⟩ ⟶$_c$ *t* is not true!

theorem *"⋀s t.*
⟨*compile c,[],s*⟩ *-*→* ⟨*[],rev(compile c),t*⟩ ⟹ ⟨*c,s*⟩ ⟶$_c$ *t"*
⟨*proof*⟩

end

theory *Live* **imports** *Natural*
begin

Which variables/locations does an expression depend on? Any set of variables that completely determine the value of the expression, in the worst case all locations:

consts *Dep :: "((loc ⇒ 'a) ⇒ 'b) ⇒ loc set"*
specification *(Dep)*
dep_on: "(∀x∈Dep e. s x = t x) ⟹ e s = e t"
⟨*proof*⟩

The following definition of *Dep* looks very tempting *Dep e = {a. ∃s t. (∀x. x ≠ a ⟶ s x = t x) ∧ e s ≠ e t}* but does not work in case *e* depends on an infinite set of variables. For example, if *e s* tests if *s* is 0 at infinitely many locations. Then *Dep e* incorrectly yields the empty set!

If we had a concrete representation of expressions, we would simply write a recursive free-variables function.

primrec *L :: "com ⇒ loc set ⇒ loc set"* **where**
"L SKIP A = A" |
"L (x :== e) A = A-{x} ∪ Dep e" |
"L (c1; c2) A = (L c1 ∘ L c2) A" |
"L (IF b THEN c1 ELSE c2) A = Dep b ∪ L c1 A ∪ L c2 A" |
"L (WHILE b DO c) A = Dep b ∪ A ∪ L c A"

primrec *"kill" :: "com ⇒ loc set"* **where**
"kill SKIP = {}" |
"kill (x :== e) = {x}" |
"kill (c1; c2) = kill c1 ∪ kill c2" |
"kill (IF b THEN c1 ELSE c2) = Dep b ∪ kill c1 ∩ kill c2" |
"kill (WHILE b DO c) = {}"

primrec *gen :: "com ⇒ loc set"* **where**
"gen SKIP = {}" |
"gen (x :== e) = Dep e" |
"gen (c1; c2) = gen c1 ∪ (gen c2-kill c1)" |
"gen (IF b THEN c1 ELSE c2) = Dep b ∪ gen c1 ∪ gen c2" |
"gen (WHILE b DO c) = Dep b ∪ gen c"

lemma *L_gen_kill: "L c A = gen c ∪ (A - kill c)"*
⟨*proof*⟩

lemma *L_idemp: "L c (L c A) ⊆ L c A"*
⟨*proof*⟩

theorem *L_sound: "∀ x ∈ L c A. s x = t x ⟹ ⟨c,s⟩ ⟶c s' ⟹ ⟨c,t⟩ ⟶c t' ⟹*
∀x∈A. s' x = t' x"
⟨*proof*⟩

primrec *bury :: "com ⇒ loc set ⇒ com"* **where**
"bury SKIP _ = SKIP" |
"bury (x :== e) A = (if x:A then x:== e else SKIP)" |
"bury (c1; c2) A = (bury c1 (L c2 A); bury c2 A)" |
"bury (IF b THEN c1 ELSE c2) A = (IF b THEN bury c1 A ELSE bury c2 A)" |
"bury (WHILE b DO c) A = (WHILE b DO bury c (Dep b ∪ A ∪ L c A))"

theorem *bury_sound:*
"∀ x ∈ L c A. s x = t x ⟹ ⟨c,s⟩ ⟶c s' ⟹ ⟨bury c A,t⟩ ⟶c t' ⟹
∀x∈A. s' x = t' x"
⟨*proof*⟩

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.