

IMP — A WHILE-language and its Semantics

Gerwin Klein, Heiko Loetzbeyer, Tobias Nipkow, Robert Sandner

November 22, 2007

Abstract

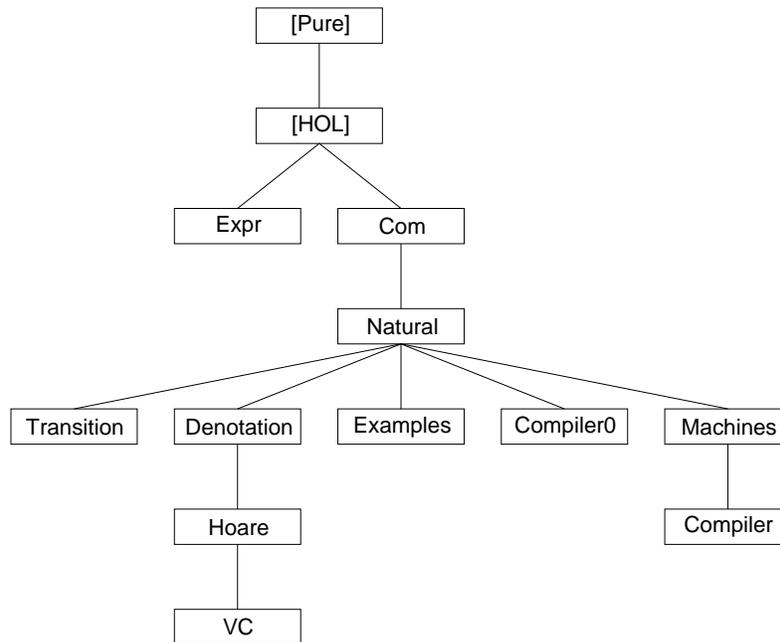
The denotational, operational, and axiomatic semantics, a verification condition generator, and all the necessary soundness, completeness and equivalence proofs. Essentially a formalization of the first 100 pages of [3].

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 | 9 |
| 4 | Transition Semantics of Commands | 11 |
| 4.1 | The transition relation | 11 |
| 4.2 | Examples | 13 |
| 4.3 | Basic properties | 13 |
| 4.4 | Equivalence to natural semantics (after Nielson and Nielson) | 14 |
| 4.5 | Winskel's Proof | 18 |
| 4.6 | A proof without n | 21 |
| 5 | Denotational Semantics of Commands | 23 |

| | | |
|----------|--------------------------------------------|-----------|
| 6 | Inductive Definition of Hoare Logic | 24 |
| 7 | Verification Conditions | 27 |
| 8 | Examples | 30 |
| 8.1 | An example due to Tony Hoare | 31 |
| 8.2 | Factorial | 31 |
| 9 | A Simple Compiler | 32 |
| 9.1 | An abstract, simplistic machine | 32 |
| 9.2 | The compiler | 33 |
| 9.3 | Context lifting lemmas | 33 |
| 9.4 | Compiler correctness | 34 |
| 9.5 | Instructions | 36 |
| 9.6 | M0 with PC | 36 |
| 9.7 | M0 with lists | 37 |
| 9.8 | The compiler | 40 |
| 9.9 | Compiler correctness | 40 |



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

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')"
by (rule,cases set: evala) auto

lemma [simp]: "(X(x),sigma) -a-> i = (i = sigma x)"
by (rule,cases set: evala) auto

lemma [simp]:
"(Op1 f e,sigma) -a-> i = ($\exists n. i = f n \wedge (e,sigma) -a-> n$)"
by (rule,cases set: evala) auto

lemma [simp]:
"(Op2 f a1 a2,sigma) -a-> i =
($\exists n0 n1. i = f n0 n1 \wedge (a1, sigma) -a-> n0 \wedge (a2, sigma) -a-> n1$)"
by (rule,cases set: evala) auto

lemma [simp]: "((true,sigma) -b-> w) = (w=True)"
by (rule,cases set: evalb) auto

lemma [simp]:

```

"((false,sigma) -b-> w) = (w=False)"
by (rule,cases set: evalb) auto

lemma [simp]:
"((ROP f a0 a1,sigma) -b-> w) =
(? m. (a0,sigma) -a-> m & (? n. (a1,sigma) -a-> n & w = f m n))"
by (rule,cases set: evalb) blast+

lemma [simp]:
"((noti(b),sigma) -b-> w) = (? x. (b,sigma) -b-> x & w = (~x))"
by (rule,cases set: evalb) blast+

lemma [simp]:
"((b0 andi b1,sigma) -b-> w) =
(? x. (b0,sigma) -b-> x & (? y. (b1,sigma) -b-> y & w = (x&y)))"
by (rule,cases set: evalb) blast+

lemma [simp]:
"((b0 ori b1,sigma) -b-> w) =
(? x. (b0,sigma) -b-> x & (? y. (b1,sigma) -b-> y & w = (x|y)))"
by (rule,cases set: evalb) blast+

lemma aexp_iff: "((a,s) -a-> n) = (A a s = n)"
by (induct a arbitrary: n) auto

lemma bexp_iff:
"((b,s) -b-> w) = (B b s = w)"
by (induct b arbitrary: w) (auto simp add: aexp_iff)

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 $\langle c, s \rangle \rightarrow_c s'$ for *Statement* c , *started in state* s , *terminates in state* s' . Formally, $\langle c, s \rangle \rightarrow_c s'$ 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" ("<_,_>/  $\rightarrow_c$  _" [0,0,60] 60)
where
  Skip:    "<skip,s>  $\rightarrow_c$  s"
  | Assign: "<x ::= a,s>  $\rightarrow_c$  s[x $\mapsto$ a s]"

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

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

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

```
"⟨skip, s⟩ ⟶c s' = (s' = s)"
by (rule, erule evalc.cases) auto
```

lemma assign:

```
"⟨x ::= a, s⟩ ⟶c s' = (s' = s[x ↦ a])"
by (rule, erule evalc.cases) auto
```

lemma semi:

```
"⟨c0; c1, s⟩ ⟶c s' = (∃ s''. ⟨c0, s⟩ ⟶c s'' ∧ ⟨c1, s''⟩ ⟶c s)"
by (rule, erule evalc.cases) auto
```

lemma ifTrue:

```
"b s ⟹ ⟨if b then c0 else c1, s⟩ ⟶c s' = ⟨c0, s⟩ ⟶c s'"
by (rule, erule evalc.cases) auto
```

lemma ifFalse:

```
"¬ b s ⟹ ⟨if b then c0 else c1, s⟩ ⟶c s' = ⟨c1, s⟩ ⟶c s'"
by (rule, erule evalc.cases) auto
```

lemma whileFalse:

```
"¬ b s ⟹ ⟨while b do c, s⟩ ⟶c s' = (s' = s)"
by (rule, erule evalc.cases) auto
```

lemma whileTrue:

```
"b s ⟹
⟨while b do c, s⟩ ⟶c s' =
(∃ s''. ⟨c, s⟩ ⟶c s'' ∧ ⟨while b do c, s''⟩ ⟶c s)"
by (rule, erule evalc.cases) auto
```

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!]:

```
"( $\bigwedge s s'. \langle c, s \rangle \longrightarrow_c s' = \langle c', s \rangle \longrightarrow_c s'$ )  $\Longrightarrow c \sim c'$ "
by (unfold equiv_c_def) blast
```

lemma equivD1:

```
" $c \sim c' \Longrightarrow \langle c, s \rangle \longrightarrow_c s' \Longrightarrow \langle c', s \rangle \longrightarrow_c s'$ "
by (unfold equiv_c_def) blast
```

lemma equivD2:

```
" $c \sim c' \Longrightarrow \langle c', s \rangle \longrightarrow_c s' \Longrightarrow \langle c, s \rangle \longrightarrow_c s'$ "
by (unfold equiv_c_def) blast
```

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

lemma unfold_while:

```
"(while b do c)  $\sim$  (if b then c; while b do c else skip)" (is "?w  $\sim$  ?if")
```

proof -

```
— to show the equivalence, we look at the derivation tree for
— each side and from that construct a derivation tree for the other side
{ fix s s' assume w: " $\langle ?w, s \rangle \longrightarrow_c s'$ "
  — as a first thing we note that, if  $b$  is False in state  $s$ ,
  — then both statements do nothing:
  hence " $\neg b \ s \Longrightarrow s = s'$ " by simp
  hence " $\neg b \ s \Longrightarrow \langle ?if, s \rangle \longrightarrow_c s'$ " by simp
  moreover
  — on the other hand, if  $b$  is True in state  $s$ ,
  — then only the WhileTrue rule can have been used to derive  $\langle ?w, s \rangle \longrightarrow_c s'$ 
  { assume b: " $b \ s$ "
    with w obtain s'' where
      " $\langle c, s \rangle \longrightarrow_c s''$ " and " $\langle ?w, s'' \rangle \longrightarrow_c s'$ " by (cases set: evalc) auto
    — now we can build a derivation tree for the if
    — first, the body of the True-branch:
    hence " $\langle c; ?w, s \rangle \longrightarrow_c s'$ " by (rule Semi)
    — then the whole if
    with b have " $\langle ?if, s \rangle \longrightarrow_c s'$ " by (rule IfTrue)
  }
}
ultimately
```

```

— both cases together give us what we want:
have "<?if, s> →c s'" by blast
}
moreover
— now the other direction:
{ fix s s' assume "if": "<?if, s> →c s'"
— again, if b is False in state s, then the False-branch
— of the if is executed, and both statements do nothing:
hence "¬b s ⇒ s = s'" by simp
hence "¬b s ⇒ <?w, s> →c s'" by simp
moreover
— on the other hand, if b is True in state s,
— then this time only the IfTrue rule can have be used
{ assume b: "b s"
with "if" have "<c; ?w, s> →c s'" by (cases set: evalc) auto
— and for this, only the Semi-rule is applicable:
then obtain s'' where
  "<c, s> →c s''" and "<?w, s''> →c s''" by (cases set: evalc) auto
— with this information, we can build a derivation tree for the while
with b
have "<?w, s> →c s'" by (rule WhileTrue)
}
ultimately
— both cases together again give us what we want:
have "<?w, s> →c s'" by blast
}
ultimately
show ?thesis by blast
qed

```

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"
  using prems
proof (induct arbitrary: u set: evalc)
  fix s u assume "<skip, s> →c u"
  thus "u = s" by simp
next
  fix a s x u assume "<x ::= a, s> →c u"
  thus "u = s[x ↦ a s]" by simp
next
  fix c0 c1 s s1 s2 u
  assume IH0: "∧u. <c0, s> →c u ⇒ u = s2"
  assume IH1: "∧u. <c1, s2> →c u ⇒ u = s1"

  assume "<c0; c1, s> →c u"

```

```

then obtain s' where
  c0: " $\langle c0, s \rangle \rightarrow_c s'$ " and
  c1: " $\langle c1, s' \rangle \rightarrow_c u$ "
  by auto

from c0 IH0 have "s'=s2" by blast
with c1 IH1 show "u=s1" by blast
next
fix b c0 c1 s s1 u
assume IH: " $\bigwedge u. \langle c0, s \rangle \rightarrow_c u \implies u = s1$ "

assume "b s" and " $\langle \text{if } b \text{ then } c0 \text{ else } c1, s \rangle \rightarrow_c u$ "
hence " $\langle c0, s \rangle \rightarrow_c u$ " by simp
with IH show "u = s1" by blast
next
fix b c0 c1 s s1 u
assume IH: " $\bigwedge u. \langle c1, s \rangle \rightarrow_c u \implies u = s1$ "

assume " $\neg b s$ " and " $\langle \text{if } b \text{ then } c0 \text{ else } c1, s \rangle \rightarrow_c u$ "
hence " $\langle c1, s \rangle \rightarrow_c u$ " by simp
with IH show "u = s1" by blast
next
fix b c s u
assume " $\neg b s$ " and " $\langle \text{while } b \text{ do } c, s \rangle \rightarrow_c u$ "
thus "u = s" by simp
next
fix b c s s1 s2 u
assume "IHc": " $\bigwedge u. \langle c, s \rangle \rightarrow_c u \implies u = s2$ "
assume "IHw": " $\bigwedge u. \langle \text{while } b \text{ do } c, s2 \rangle \rightarrow_c u \implies u = s1$ "

assume "b s" and " $\langle \text{while } b \text{ do } c, s \rangle \rightarrow_c u$ "
then obtain s' where
  c: " $\langle c, s \rangle \rightarrow_c s'$ " and
  w: " $\langle \text{while } b \text{ do } c, s' \rangle \rightarrow_c u$ "
  by auto

from c "IHc" have "s' = s2" by blast
with w "IHw" show "u = s1" by blast
qed

```

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

```

theorem
  assumes " $\langle c, s \rangle \rightarrow_c t$ " and " $\langle c, s \rangle \rightarrow_c u$ "
  shows "u = t"
  using prems
proof (induct arbitrary: u)
  — the simple skip case for demonstration:
  fix s u assume " $\langle \text{skip}, s \rangle \rightarrow_c u$ "

```

```

    thus "u = s" by simp
next
  — and the only really interesting case, while:
  fix b c s s1 s2 u
  assume "IHc": "∧u. ⟨c,s⟩ →c u ⇒ u = s2"
  assume "IHw": "∧u. ⟨while b do c,s2⟩ →c u ⇒ u = s1"

  assume "b s" and "⟨while b do c,s⟩ →c u"
  then obtain s' where
    c: "⟨c,s⟩ →c s'" and
    w: "⟨while b do c,s'⟩ →c u"
  by auto

  from c "IHc" have "s' = s2" by blast
  with w "IHw" show "u = s1" by blast
qed (best dest: evalc_cases [THEN iffD1])+ — prove the rest automatically
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 $((com \times state) \times com \times state)$ set but instead: $((com\ option \times state) \times com\ option \times state)$ 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" ("<_>")

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 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"
  ("_ -n→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^*"

```

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>)"
  by (induct y, rule, cases set: evalc1, auto)

```

```

lemma Assign_1: "<x := a, s> →1 y = (y = <s[x ↦ a s]>)"
  by (induct y, rule, cases set: evalc1, auto)

```

```

lemma Cond_1:
  "<if b then c1 else c2, s> →1 y = ((b s → y = <c1, s>) ∧ (¬b s → y = <c2, s>))"
  by (induct y, rule, cases set: evalc1, auto)

```

```

lemma While_1:
  "<while b do c, s> →1 y = (y = <if b then c; while b do c else skip, s>)"
  by (induct y, rule, cases set: evalc1, auto)

```

lemmas [simp] = SKIP_1 Assign_1 Cond_1 While_1

4.2 Examples

lemma

```
"s x = 0  $\implies$   $\langle$ while  $\lambda s. s\ x \neq 1$  do  $(x ::= \lambda s. s\ x + 1)$ ,  $s$  $\rangle \longrightarrow_1^*$   $\langle s[x \mapsto 1] \rangle$ "
(is "_  $\implies$   $\langle ?w, \_ \rangle \longrightarrow_1^* \_$ ")
```

proof -

```
let ?c = "x ::=  $\lambda s. s\ x + 1$ "
let ?if = "if  $\lambda s. s\ x \neq 1$  then ?c; ?w else skip"
assume [simp]: "s x = 0"
have " $\langle ?w, s \rangle \longrightarrow_1 \langle ?if, s \rangle$ " ..
also have " $\langle ?if, s \rangle \longrightarrow_1 \langle ?c; ?w, s \rangle$ " by simp
also have " $\langle ?c; ?w, s \rangle \longrightarrow_1 \langle ?w, s[x \mapsto 1] \rangle$ " 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  $\implies$   $\langle$ while  $\lambda s. s\ x \neq 1$  do  $(x ::= \lambda s. s\ x + 1)$ ,  $s$  $\rangle \longrightarrow_1^*$  s'"
(is "_  $\implies$   $\langle ?w, \_ \rangle \longrightarrow_1^* s'$ ")
```

proof -

```
let ?c = "x ::=  $\lambda s. s\ x + 1$ "
let ?if = "if  $\lambda s. s\ x \neq 1$  then ?c; ?w else skip"
assume [simp]: "s x = 2"
note update_def [simp]
have " $\langle ?w, s \rangle \longrightarrow_1 \langle ?if, s \rangle$ " ..
also have " $\langle ?if, s \rangle \longrightarrow_1 \langle ?c; ?w, s \rangle$ " by simp
also have " $\langle ?c; ?w, s \rangle \longrightarrow_1 \langle ?w, s[x \mapsto 3] \rangle$ " by (rule Semi1) simp
also have " $\langle ?w, s[x \mapsto 3] \rangle \longrightarrow_1 \langle ?if, s[x \mapsto 3] \rangle$ " ..
also have " $\langle ?if, s[x \mapsto 3] \rangle \longrightarrow_1 \langle ?c; ?w, s[x \mapsto 3] \rangle$ " by simp
also have " $\langle ?c; ?w, s[x \mapsto 3] \rangle \longrightarrow_1 \langle ?w, s[x \mapsto 4] \rangle$ " by (rule Semi1) simp
also have " $\langle ?w, s[x \mapsto 4] \rangle \longrightarrow_1 \langle ?if, s[x \mapsto 4] \rangle$ " ..
also have " $\langle ?if, s[x \mapsto 4] \rangle \longrightarrow_1 \langle ?c; ?w, s[x \mapsto 4] \rangle$ " by simp
also have " $\langle ?c; ?w, s[x \mapsto 4] \rangle \longrightarrow_1 \langle ?w, s[x \mapsto 5] \rangle$ " by (rule Semi1) simp
oops
```

4.3 Basic properties

There are no *stuck* programs:

lemma no_stuck: " $\exists y. \langle c, s \rangle \longrightarrow_1 y$ "

proof (induct c)

— case Semi:

```
fix c1 c2 assume " $\exists y. \langle c1, s \rangle \longrightarrow_1 y$ "
then obtain y where " $\langle c1, s \rangle \longrightarrow_1 y$ " ..
then obtain c1' s' where " $\langle c1, s \rangle \longrightarrow_1 \langle s' \rangle \vee \langle c1, s \rangle \longrightarrow_1 \langle c1', s' \rangle$ "
  by (cases y, cases "fst y") auto
thus " $\exists s'. \langle c1; c2, s \rangle \longrightarrow_1 s'$ " 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 [elim!]: "⟨s⟩ →1 y ⇒ P"
  by (induct y, auto elim: evalc1.cases)

lemma evalc_None_retrancl [simp, dest!]: "⟨s⟩ →1* s' ⇒ s' = ⟨s⟩"
  by (induct set: rtrancl) auto
lemma evalc1_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 ∧ n=1"
  by (cases n) auto

```

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 (induct n arbitrary: c1 c2 s s'')
  case 0
  then show ?case by simp
next
  case (Suc n)

  from '⟨c1; c2, s⟩ -Suc n→1 ⟨s''⟩'
  obtain co s''' where
    1: "⟨c1; c2, s⟩ →1 (co, s''')" and
    n: "⟨co, s'''⟩ -n→1 ⟨s''⟩"
  by auto

  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
    "co = Some c2" and "s''' = 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
    "co = Some (c1'; c2)" and "s''' = s'" and

```

```

      c1: "<c1, s> →1 <c1', s'>"
    by auto
  with n have "<c1'; c2, s'> -n→1 <s''>" by simp
  with Suc.hyps obtain i j s0 where
    c1': "<c1', s'> -i→1 <s0>" and
    c2:  "<c2, s0> -j→1 <s''>" and
    i:   "n = i+j"
  by fast

  from c1 c1'
  have "<c1, s> -(i+1)→1 <s0>" by (auto 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

lemma semiI:
  "<c0, s> -n→1 <s''> ⇒ <c1, s''> →1* <s'> ⇒ <c0; c1, s> →1* <s'>"
proof (induct n arbitrary: c0 s s'')
  case 0
  from '<c0, s> -(0::nat)→1 <s''>'
  have False by simp
  thus ?case ..
next
  case (Suc n)
  note c0 = '<c0, s> -Suc n→1 <s''>'
  note c1 = '<c1, s''> →1* <s'>'
  note 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:
  assumes "<c,s> →c s'"
  shows "<c, s> →1* <s'>"
  using prems
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_imp_rel_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_imp_rel_pow)
with w have "<c; ?w,s> →1* <s''>" by - (rule semiI)
finally show "<while b do c,s> →1* <s''>" ..
qed

```

Finally, the equivalence theorem:

```

theorem evalc_equiv_evalc1:
  "<c, s> →c s' = <c,s> →1* <s''>"
proof
  assume "<c,s> →c s'"
  then 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_imp_rel_pow)
  moreover
  have "<c, s> -n→1 <s''> ⇒ <c,s> →c s'"
  proof (induct arbitrary: c s s' rule: less_induct)
    fix n
    assume IH: "∧m c s s'. m < n ⇒ <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 IH i c1
      have "<c1,s> →c s''" .
      moreover
      from IH j c2
      have "<c2,s''> →c s''" .

```

```

    moreover
    note semi
    ultimately
    show " $\langle c, s \rangle \rightarrow_c s'$ " by blast
next
fix b c1 c2 assume If: " $c = \text{if } b \text{ then } c1 \text{ else } c2$ "
{ assume True: " $b \text{ s} = \text{True}$ "
  with If c n
  have " $\langle c1, s \rangle \rightarrow_1 \langle s' \rangle$ " by auto
  with n IH
  have " $\langle c1, s \rangle \rightarrow_c s'$ " by blast
  with If True
  have " $\langle c, s \rangle \rightarrow_c s'$ " by simp
}
moreover
{ assume False: " $b \text{ s} = \text{False}$ "
  with If c n
  have " $\langle c2, s \rangle \rightarrow_1 \langle s' \rangle$ " by auto
  with n IH
  have " $\langle c2, s \rangle \rightarrow_c s'$ " by blast
  with If False
  have " $\langle c, s \rangle \rightarrow_c s'$ " by simp
}
ultimately
show " $\langle c, s \rangle \rightarrow_c s'$ " by (cases "b s") auto
next
fix b c' assume w: " $c = \text{while } b \text{ do } c'$ "
with c n
have " $\langle \text{if } b \text{ then } c'; \text{ while } b \text{ do } c' \text{ else skip}, s \rangle \rightarrow_1 \langle s' \rangle$ "
  (is " $\langle ?\text{if}, _ \rangle \rightarrow_1 _$ ") by auto
with n IH
have " $\langle \text{if } b \text{ then } c'; \text{ while } b \text{ do } c' \text{ else skip}, s \rangle \rightarrow_c s'$ " by blast
moreover note unfold_while [of b c']
— while b do c'  $\sim$  if b then c'; while b do c' else skip
ultimately
have " $\langle \text{while } b \text{ do } c', s \rangle \rightarrow_c s'$ " by (blast dest: equivD2)
with w show " $\langle c, s \rangle \rightarrow_c s'$ " by simp
qed
qed
ultimately
show " $\langle c, s \rangle \rightarrow_c s'$ " by blast
qed

```

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 -
  assume "¬ b s"
  have "⟨?w, s⟩ →1 ⟨if b then c; ?w else skip, s⟩" ..
  also from '¬ b s' have "⟨if b then c; ?w else skip, s⟩ →1 ⟨skip, s⟩" ..
  also have "⟨skip, s⟩ →1 ⟨s⟩" ..
  finally show "⟨?w, s⟩ →1* ⟨s⟩" ..
qed

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 -
  assume "b s"
  have "⟨?w, s⟩ →1 ⟨if b then c; ?w else skip, s⟩" ..
  also from 'b s' have "⟨if b then c; ?w else skip, s⟩ →1 ⟨c; ?w, s⟩" ..
  finally show "⟨?w, s⟩ →1* ⟨c; ?w, s⟩" ..
qed

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⟩"
apply (induct set: evalc)

— SKIP
apply blast

— ASSIGN
apply fast

— SEMI
apply (fast dest: rtrancl_imp_UN_rel_pow intro: semiI)

— IF
apply (fast intro: converse_rtrancl_into_rtrancl)
apply (fast intro: converse_rtrancl_into_rtrancl)

— WHILE
apply fast
apply (fast dest: rtrancl_imp_UN_rel_pow intro: converse_rtrancl_into_rtrancl semiI)

```

done

lemma lemma2:

```
"⟨c;d,s⟩ -n→1 ⟨u⟩ ⇒ ∃ t m. ⟨c,s⟩ →1* ⟨t⟩ ∧ ⟨d,t⟩ -m→1 ⟨u⟩ ∧ m ≤ n"
apply (induct n arbitrary: c d s u)
— case n = 0
  apply fastsimp
— induction step
  apply (fast intro!: le_SucI le_refl dest!: rel_pow_Suc_D2
        elim!: rel_pow_imp_rtrancl converse_rtrancl_into_rtrancl)
done
```

lemma evalc1_impl_evalc:

```
"⟨c,s⟩ →1* ⟨t⟩ ⇒ ⟨c,s⟩ →c t"
apply (induct c arbitrary: s t)
apply (safe dest!: rtrancl_imp_UN_rel_pow)
```

— SKIP

```
apply (simp add: SKIP_n)
```

— ASSIGN

```
apply (fastsimp elim: rel_pow_E2)
```

— SEMI

```
apply (fast dest!: rel_pow_imp_rtrancl lemma2)
```

— IF

```
apply (erule rel_pow_E2)
apply simp
apply (fast dest!: rel_pow_imp_rtrancl)
```

— WHILE, induction on the length of the computation

```
apply (rename_tac b c s t n)
apply (erule_tac P = "?X -n→1 ?Y" in rev_mp)
apply (rule_tac x = "s" in spec)
apply (induct_tac n rule: nat_less_induct)
apply (intro strip)
apply (erule rel_pow_E2)
  apply simp
apply (simp only: split_paired_all)
apply (erule evalc1_E)
```

apply simp

```
apply (case_tac "b x")
```

— WhileTrue

```
  apply (erule rel_pow_E2)
    apply simp
  apply (clarify dest!: lemma2)
  apply atomize
```

```

  apply (erule allE, erule allE, erule impE, assumption)
  apply (erule_tac x=mb in allE, erule impE, fastsimp)
  apply blast
— WhileFalse
apply (erule rel_pow_E2)
  apply simp
apply (simp add: SKIP_n)
done

```

proof of the equivalence of evalc and evalc1

```

lemma evalc1_eq_evalc: " $\langle c, s \rangle \longrightarrow_1^* \langle t \rangle = \langle c, s \rangle \longrightarrow_c t$ "
  by (fast elim!: evalc1_impl_evalc evalc_impl_evalc1)

```

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 " $\langle c1, s1 \rangle \longrightarrow_1^* \langle s2 \rangle$ "
    and " $\langle c2, s2 \rangle \longrightarrow_1^* cs3$ "
  shows " $\langle c1; c2, s1 \rangle \longrightarrow_1^* cs3$ "
proof -
— The induction rule needs P to be a function of Some c1
from prems
have " $\langle (\lambda c. \text{if } c = \text{None then } c2 \text{ else the } c; c2) (\text{Some } c1), s1 \rangle \longrightarrow_1^* cs3$ "
  apply (induct rule: converse_rtrancl_induct2)
  apply simp
  apply (rename_tac c s')
  apply simp
  apply (rule conjI)
  apply fast
  apply clarify
  apply (case_tac c)
  apply (auto intro: converse_rtrancl_into_rtrancl)
  done
  then show ?thesis by simp
qed

```

```

lemma evalc_impl_evalc1': " $\langle c, s \rangle \longrightarrow_c s1 \implies \langle c, s \rangle \longrightarrow_1^* \langle s1 \rangle$ "
  apply (induct set: evalc)

```

```

— SKIP
apply fast

```

```

— ASSIGN
apply fast

```

```

— SEMI
apply (fast intro: my_lemma1)

— IF
apply (fast intro: converse_rtrancl_into_rtrancl)
apply (fast intro: converse_rtrancl_into_rtrancl)

— WHILE
apply fast
apply (fast intro: converse_rtrancl_into_rtrancl my_lemma1)

done

```

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 $\langle c',s' \rangle$ 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 $-->$ 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) →1 (c',s') ⇒ c ≠ None ⇒
  ⟨if c'=None then skip else the c',s'⟩ →c t ⇒ ⟨the c,s⟩ →c t"
by (induct arbitrary: t set: evalc1)
  (auto elim!: evalc1_term_cases equivD2 [OF unfold_while])

lemma FB_lemma2:
  "(c,s) →1* (c',s') ⇒ c ≠ None ⇒
  ⟨if c' = None then skip else the c',s'⟩ →c t ⇒ ⟨the c,s⟩ →c t"
apply (induct rule: converse_rtrancl_induct2, force)
apply (fastsimp elim!: evalc1_term_cases intro: FB_lemma3)
done

lemma evalc1_impl_evalc': "⟨c,s⟩ →1* ⟨t⟩ ⇒ ⟨c,s⟩ →c t"
by (fastsimp dest: FB_lemma2)

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)"
```

```
by (unfold Gamma_def mono_def) fast
```

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

```

apply simp
apply (subst lfp_unfold [OF Gamma_mono]) — lhs only
apply (simp add: Gamma_def)
done

```

```

lemma com1: " $\langle c, s \rangle \rightarrow_c t \implies (s, t) \in C(c)$ "

```

```

apply (induct set: evalc)
apply auto

```

```

apply (unfold Gamma_def)
apply (subst lfp_unfold[OF Gamma_mono, simplified Gamma_def])
apply fast
apply (subst lfp_unfold[OF Gamma_mono, simplified Gamma_def])
apply fast
done

```

```

lemma com2: " $(s, t) \in C(c) \implies \langle c, s \rangle \rightarrow_c t$ "
apply (induct c arbitrary: s t)

```

```

apply simp_all
apply fast
apply fast

```

```

apply (erule lfp_induct2 [OF _ Gamma_mono])
apply (unfold Gamma_def)
apply fast
done

```

```

lemma denotational_is_natural: " $(s, t) \in C(c) = (\langle c, s \rangle \rightarrow_c t)$ "
  by (fast elim: com2 dest: com1)

```

```

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 $\mapsto$ 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}"
apply (erule hoare.conseq)
apply assumption
apply fast
done

lemma hoare_conseq2: "[| |- {P}c{Q}; !s. Q s --> Q' s |] ==> |- {P}c{Q'}"
apply (rule hoare.conseq)
prefer 2 apply (assumption)
apply fast
apply fast
done

lemma hoare_sound: "|- {P}c{Q} ==> |= {P}c{Q}"
apply (unfold hoare_valid_def)
apply (induct set: hoare)
  apply (simp_all (no_asm_simp))
  apply fast
  apply fast
apply (rule allI, rule allI, rule impI)
apply (erule lfp_induct2)
  apply (rule Gamma_mono)
apply (unfold Gamma_def)
apply fast
done

lemma wp_SKIP: "wp skip Q = Q"
apply (unfold wp_def)
apply (simp (no_asm))

```

```

done

lemma wp_Ass: "wp (x:=a) Q = (%s. Q(s[x↦a s]))"
apply (unfold wp_def)
apply (simp (no_asm))
done

lemma wp_Semi: "wp (c;d) Q = wp c (wp d Q)"
apply (unfold wp_def)
apply (simp (no_asm))
apply (rule ext)
apply fast
done

lemma wp_If:
  "wp (if b then c else d) Q = (%s. (b s --> wp c Q s) & (~b s --> wp d Q s))"
apply (unfold wp_def)
apply (simp (no_asm))
apply (rule ext)
apply fast
done

lemma wp_While_True:
  "b s ==> wp (while b do c) Q s = wp (c;while b do c) Q s"
apply (unfold wp_def)
apply (subst C_While_If)
apply (simp (no_asm_simp))
done

lemma wp_While_False: "~b s ==> wp (while b do c) Q s = Q s"
apply (unfold wp_def)
apply (subst C_While_If)
apply (simp (no_asm_simp))
done

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)"
  by simp

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}))"
apply (simp (no_asm))
apply (rule iffI)
apply (rule weak_coinduct)
  apply (erule CollectI)
apply safe
apply simp

```

```

    apply simp
  apply (simp add: wp_def Gamma_def)
  apply (intro strip)
  apply (rule mp)
    prefer 2 apply (assumption)
  apply (erule lfp_induct2)
  apply (fast intro!: monoI)
  apply (subst gfp_unfold)
    apply (fast intro!: monoI)
  apply fast
done

declare C_while [simp del]

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

lemma wp_is_pre: " $\vdash \{wp\ c\ Q\} c \{Q\}$ "
  apply (induct c arbitrary: Q)
    apply (simp_all (no_asm))
    apply fast+
  apply (blast intro: hoare_conseq1)
  apply (rule hoare_conseq2)
  apply (rule hoare.While)
  apply (rule hoare_conseq1)
    prefer 2 apply fast
  apply safe
  apply simp
  apply simp
done

lemma hoare_relative_complete: " $\vdash \{P\}c\{Q\} \implies \vdash \{P\}c\{Q\}$ "
  apply (rule hoare_conseq1 [OF _ wp_is_pre])
  apply (unfold hoare_valid_def wp_def)
  apply fast
done

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 l1: "!s. P s --> P s" by fast

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

```

apply (induct c arbitrary: Q)
  apply (simp_all (no_asm))
  apply fast
  apply fast
  apply fast

apply atomize
apply (tactic "Deepen_tac 4 1")

apply atomize
apply (intro allI impI)
apply (rule conseq)
  apply (rule l)
  apply (rule While)
defer
  apply fast
apply (rule_tac P="awp c fun2" in conseq)
  apply fast
  apply fast
apply fast
done

lemma awp_mono [rule_format (no_asm)]:
  "!P Q. (!s. P s --> Q s) --> (!s. awp c P s --> awp c Q s)"
apply (induct c)
  apply (simp_all (no_asm_simp))
apply (rule allI, rule allI, rule impI)
apply (erule allE, erule allE, erule mp)
apply (erule allE, erule allE, erule mp, assumption)
done

lemma vc_mono [rule_format (no_asm)]:
  "!P Q. (!s. P s --> Q s) --> (!s. vc c P s --> vc c Q s)"
apply (induct c)
  apply (simp_all (no_asm_simp))
apply safe
apply (erule allE, erule allE, erule impE, erule_tac [2] allE, erule_tac [2] mp)
prefer 2 apply assumption
apply (fast elim: awp_mono)
done

lemma vc_complete: assumes der: "|- {P}c{Q}"
  shows "( $\exists$  ac. astrip ac = c & ( $\forall$  s. vc ac Q s) & ( $\forall$  s. P s --> awp ac Q s))"
  (is "? ac. ?Eq P c Q ac")
using der
proof induct
  case skip
  show ?case (is "? ac. ?C ac")
  proof show "?C Askip" by simp qed
next

```

```

case (ass P x a)
show ?case (is "? ac. ?C ac")
proof show "?C(Aass x a)" by simp qed
next
case (semi P c1 Q c2 R)
from semi.hyps obtain ac1 where ih1: "?Eq P c1 Q ac1" by fast
from semi.hyps obtain ac2 where ih2: "?Eq Q c2 R ac2" by fast
show ?case (is "? ac. ?C ac")
proof
  show "?C(Asemi ac1 ac2)"
  using ih1 ih2 by simp (fast elim!: awp_mono vc_mono)
qed
next
case (If P b c1 Q c2)
from If.hyps obtain ac1 where ih1: "?Eq (%s. P s & b s) c1 Q ac1" by fast
from If.hyps obtain ac2 where ih2: "?Eq (%s. P s & ~b s) c2 Q ac2" by fast
show ?case (is "? ac. ?C ac")
proof
  show "?C(Aif b ac1 ac2)"
  using ih1 ih2 by simp
qed
next
case (While P b c)
from While.hyps obtain ac where ih: "?Eq (%s. P s & b s) c P ac" by fast
show ?case (is "? ac. ?C ac")
proof show "?C(Awhile b P ac)" using ih by simp qed
next
case conseq thus ?case by(fast elim!: awp_mono vc_mono)
qed

lemma vcawp_vc_awp: "vcawp c Q = (vc c Q, awp c Q)"
  by (induct c arbitrary: Q) (simp_all add: Let_def)

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  $\longrightarrow$  Q x"
    and 2: " $\langle w, s \rangle \longrightarrow_c t$ "
  shows " $w = \text{While } P \ c \Longrightarrow \langle \text{While } Q \ c, t \rangle \longrightarrow_c u \Longrightarrow \langle \text{While } Q \ c, s \rangle \longrightarrow_c u$ "
  using 2 apply induct
  using 1 apply auto
  done
```

```
lemma lemma2 [rule_format (no_asm)]:
  "[| !x. P x  $\longrightarrow$  Q x;  $\langle w, s \rangle \longrightarrow_c u$  |]  $\Longrightarrow$ 
  !c. w = While Q c  $\longrightarrow$   $\langle$ While P c; While Q c, s $\rangle \longrightarrow_c u$ "
  apply (erule evalc.induct)
  apply (simp_all (no_asm_simp))
  apply blast
  apply (case_tac "P s")
  apply auto
  done
```

```
lemma Hoare_example: "!x. P x  $\longrightarrow$  Q x  $\Longrightarrow$ 
  ( $\langle$ While P c; While Q c, s $\rangle \longrightarrow_c t$ ) = ( $\langle$ While Q c, s $\rangle \longrightarrow_c t$ )"
  by (blast intro: lemma1 lemma2 dest: semi [THEN iffD1])
```

8.2 Factorial

```
lemma factorial_3: "a $\sim$ b  $\Longrightarrow$ 
   $\langle$ factorial a b, Mem(a:=3) $\rangle \longrightarrow_c$  Mem(b:=6, a:=0)"
  by (simp add: factorial_def)
```

the same in single step mode:

```
lemmas [simp del] = evalc_cases
lemma "a $\sim$ b  $\Longrightarrow$   $\langle$ factorial a b, Mem(a:=3) $\rangle \longrightarrow_c$  Mem(b:=6, a:=0)"
  apply (unfold factorial_def)
  apply (frule not_sym)
  apply (rule evalc.intros)
  apply (rule evalc.intros)
  apply simp
  apply (rule evalc.intros)
  apply simp
  apply (rule evalc.intros)
  apply (rule evalc.intros)
  apply simp
```

```

apply (rule evalc.intros)
apply simp
apply (rule evalc.intros)
apply simp
apply (rule evalc.intros)
apply (rule evalc.intros)
apply simp
apply (rule evalc.intros)
apply simp
apply (rule evalc.intros)
apply simp
done

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⟩"
  using prems
  by induct auto

lemma app_left_1:
  assumes "is2 ⊢ ⟨s1,i1⟩ -1→ ⟨s2,i2⟩"
  shows "is1 @ is2 ⊢ ⟨s1,size is1+i1⟩ -1→ ⟨s2,size is1+i2⟩"
  using prems
  by induct auto

declare rtrancl_induct2 [induct set: rtrancl]

lemma app_right:
  assumes "is1 ⊢ ⟨s1,i1⟩ -*→ ⟨s2,i2⟩"
  shows "is1 @ is2 ⊢ ⟨s1,i1⟩ -*→ ⟨s2,i2⟩"
  using prems
proof induct
  show "is1 @ is2 ⊢ ⟨s1,i1⟩ -*→ ⟨s1,i1⟩" by simp
next
  fix s1' i1' s2 i2
  assume "is1 @ is2 ⊢ ⟨s1,i1⟩ -*→ ⟨s1',i1'⟩"
  and "is1 ⊢ ⟨s1',i1'⟩ -1→ ⟨s2,i2⟩"
  thus "is1 @ is2 ⊢ ⟨s1,i1⟩ -*→ ⟨s2,i2⟩"
  by (blast intro: app_right_1 rtrancl_trans)

```

```

qed

lemma app_left:
  assumes "is2 ⊢ ⟨s1,i1⟩ -> ⟨s2,i2⟩"
  shows "is1 @ is2 ⊢ ⟨s1,size is1+i1⟩ -> ⟨s2,size is1+i2⟩"
using prems
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'⟩"
  and "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

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:
  assumes "is ⊢ ⟨s1,i1⟩ -> ⟨s2,i2⟩"
  shows "instr # is ⊢ ⟨s1,Suc i1⟩ -> ⟨s2,Suc i2⟩"
proof -
  from app_left [OF prems, of "[instr]"]
  show ?thesis by simp
qed

```

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")
  using prems
proof induct
  show "∧s. ?P skip s s" by simp
next
  show "∧a s x. ?P (x ::= a) s (s[x↦ a s])" by force
next
  fix c0 c1 s0 s1 s2
  assume "?P c0 s0 s1"
  hence "compile c0 @ compile c1 ⊢ ⟨s0,0⟩ -> ⟨s1,length(compile c0)⟩"
  by (rule app_right)

```

```

moreover assume "?P c1 s1 s2"
hence "compile c0 @ compile c1 ⊢ ⟨s1,length(compile c0)⟩ -*→
  ⟨s2,length(compile c0)+length(compile c1)⟩"
proof -
  show "∧is1 is2 s1 s2 i2.
    is2 ⊢ ⟨s1,0⟩ -*→ ⟨s2,i2⟩ ⇒
    is1 @ is2 ⊢ ⟨s1,size is1⟩ -*→ ⟨s2,size is1+i2⟩"
    using app_left[of _ 0] by simp
  qed
  ultimately have "compile c0 @ compile c1 ⊢ ⟨s0,0⟩ -*→
    ⟨s2,length(compile c0)+length(compile c1)⟩"
    by (rule rtrancl_trans)
  thus "?P (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: "?P 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 "?P (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: "?P 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 "?P (if b then c0 else c1) s0 s1" .
next
  fix b c and s::state
  assume "¬b s"
  thus "?P (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: "?P c s0 s1" and IHw: "?P (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 "?P (while b do c) s0 s2".
qed

```

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: $R^* = Id \cup (R \circ R^*)$
 by (fast intro: rtrancl_into_rtrancl elim: rtranclE)

lemma converse_rtrancl_eq: $R^* = Id \cup (R^* \circ R)$
 by (subst r_comp_rtrancl_eq[symmetric], rule rtrancl_eq)

lemmas converse_rel_powE = rel_pow_E2

lemma R_0_Rn_commute: $R \circ R^n = R^n \circ R$
 by (induct n) (simp, simp add: 0_assoc [symmetric])

lemma converse_in_rel_pow_eq:
 $((x,z) \in R^n) = (n=0 \wedge z=x \vee (\exists m y. n = \text{Suc } m \wedge (x,y) \in R \wedge (y,z) \in R^m))$
 apply (rule iffI)
 apply (blast elim: converse_rel_powE)
 apply (fastsimp simp add: gr0_conv_Suc R_0_Rn_commute)
 done

lemma rel_pow_plus: $R^{(m+n)} = R^n \circ R^m$
 by (induct n) (simp, simp add: 0_assoc)

lemma rel_pow_plusI: $[(x,y) \in R^m; (y,z) \in R^n] \implies (x,z) \in R^{(m+n)}$
 by (simp add: rel_pow_plus_rel_compI)

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 :: $\text{"instr list} \Rightarrow ((\text{nat} \times \text{state}) \times (\text{nat} \times \text{state}))\text{set}$ "
and exec01' :: $\text{"[instrs, nat, state, nat, state]} \Rightarrow \text{bool}$ "

```

    ("(_/ ⊢ (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')) : step1"

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)"
  apply(rule iffI)
  defer
  apply(clearsimp simp add: step1.intros split: instr.split_asm split_if_asm)
  apply(erule execE)
  apply(simp_all)
  done

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⟩))"
  by(subst converse_in_rel_pow_eq, simp)

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⟩))"
  apply(simp add: rtrancl_is_UN_rel_pow del:exec_simp)
  apply(blast)
  done

declare nth_append[simp]

lemma rev_revD: "rev xs = rev ys ⇒ xs = ys"
  by simp

lemma [simp]: "(rev xs @ rev ys = rev zs) = (ys @ xs = zs)"
  apply(rule iffI)
  apply(rule rev_revD, simp)
  apply fastsimp
  done

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⟩"
  apply(induct set: step1)
  apply(simp add:exec01.SET)
  apply(fastsimp intro:exec01.JMPFT)
  apply simp
  apply(rule exec01.JMPFF)
  apply simp

```

```

    apply fastsimp
    apply simp
    apply simp
    apply simp
  apply (fastsimp simp add: exec01.JMPB)
done

```

```

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⟩"
  apply (induct arbitrary: p q p' q' set: exec01)
    apply (clarsimp simp add: neq_Nil_conv append_eq_conv_conj)
    apply (drule sym)
    apply simp
    apply (rule rev_revD)
    apply simp
  apply (clarsimp simp add: neq_Nil_conv append_eq_conv_conj)
  apply (drule sym)
  apply simp
  apply (rule rev_revD)
  apply simp
  apply (simp (no_asm_use) add: neq_Nil_conv append_eq_conv_conj, clarify)+
  apply (drule sym)
  apply simp
  apply (rule rev_revD)
  apply simp
  apply (clarsimp simp add: neq_Nil_conv append_eq_conv_conj)
  apply (drule sym)
  apply (simp add: rev_take)
  apply (rule rev_revD)
  apply (simp add: rev_drop)
done

```

```

theorem M_equiv:
  "⟨q,p,s⟩ -1→ ⟨q',p',t⟩ =
   (rev p' @ q' = rev p @ q ∧ rev p @ q ⊢ ⟨size p,s⟩ -1→ ⟨size p',t⟩)"
  by (blast dest: direction1 direction2)

```

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. 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: " $\langle c, s \rangle \longrightarrow_c t$ "
shows " $\bigwedge p q. \langle \text{compile } c @ p, q, s \rangle \dashv\rightarrow \langle p, \text{rev}(\text{compile } c) @ q, t \rangle$ "
  (is " $\bigwedge p q. ?P c s t p q$ ")
proof -
  from A show " $\bigwedge p q. ?thesis p q$ "
  proof induct
    case Skip thus ?case by simp
  next
    case Assign thus ?case by force
  next
    case Semi thus ?case by simp (blast intro:rtrancl_trans)
  next
    fix b c0 c1 s0 s1 p q
    assume IH: " $\bigwedge p q. ?P c0 s0 s1 p q$ "
    assume "b s0"
    thus "?P (if b then c0 else c1) s0 s1 p q"
      by(simp add: IH[THEN rtrancl_trans])
  next
    case IfFalse thus ?case by(simp)
  next
    case WhileFalse thus ?case by simp
  next
    fix b c and s0::state and s1 s2 p q
    assume b: "b s0" and
      IHc: " $\bigwedge p q. ?P c s0 s1 p q$ " and
      IHw: " $\bigwedge p q. ?P (\text{while } b \text{ do } c) s1 s2 p q$ "
    show "?P (while b do c) s0 s2 p q"
      using b IHc[THEN rtrancl_trans] IHw by(simp)
  qed
qed

```

The other direction!

```

inductive_cases [elim!]: " $\langle ([], p, s), (is', p', s') \rangle : \text{step1}$ "

```

```

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

```

```

apply(rule iffI)
  apply(erule converse_rel_powE, simp, fast)
apply simp
done

lemma [simp]: " $\langle \langle [], q, s \rangle \text{-}^* \rightarrow \langle p', q', t \rangle \rangle = (p' = [] \wedge q' = q \wedge t = s)$ "
by(simp add: rtrancl_is_UN_rel_pow)

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) = (( $\forall j \in$  forws instr.  $j \leq$  size is+n)  $\wedge$ 
    ( $\forall j \in$  backws instr.  $j \leq$  m)  $\wedge$  closed (Suc m) n is)"

lemma [simp]:
  " $\bigwedge m n.$  closed m n (C1@C2) =
    (closed m (n+size C2) C1  $\wedge$  closed (m+size C1) n C2)"
by(induct C1, simp, simp add:add_ac)

theorem [simp]: " $\bigwedge m n.$  closed m n (compile c)"
by(induct c, simp_all add:backws_def forws_def)

lemma drop_lem: " $n \leq$  size(p1@p2)
 $\implies$  (p1' @ p2 = drop n p1 @ drop (n - size p1) p2) =
  (n  $\leq$  size p1 & p1' = drop n p1)"
apply(rule iffI)
  defer apply simp
apply(subgoal_tac "n  $\leq$  size p1")
  apply simp
apply(rule ccontr)
apply(drule_tac f = length in arg_cong)
apply simp
done

lemma reduce_exec1:
  " $\langle i \# p1 @ p2, q1 @ q2, s \rangle \text{-}1 \rightarrow \langle p1' @ p2, q1' @ q2, s' \rangle \implies$ 
   $\langle i \# p1, q1, s \rangle \text{-}1 \rightarrow \langle p1', q1', s' \rangle$ "
by(clarsimp simp add: drop_lem split:instr.split_asm split_if_asm)

```

```

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"
apply(clearsimp simp add:forws_def backws_def
      split:instr.split_asm split_if_asm)
done

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(induct n)
  fix C1 C2 r
  assume "?H C1 C2 r 0"
  thus "?P C1 C2 r 0" by simp
next
  fix C1 C2 r n
  assume IH: "∧C1 C2 r. ?CL C1 C2 ⇒ ?H C1 C2 r n ⇒ ?P C1 C2 r n"
  assume CL: "?CL C1 C2" and H: "?H C1 C2 r (Suc n)"
  show "?P C1 C2 r (Suc n)"
  proof (cases C2)
    assume "C2 = []" with H show ?thesis by simp
  next
    fix instr t1C2
    assume C2: "C2 = instr # t1C2"
    from H C2 obtain p' q' r'
      where 1: "⟨instr # t1C2 @ p1 @ p2, C1 @ q,r⟩ -1→ ⟨p',q',r'⟩"
      and n: "⟨p',q',r'⟩ -n→ ⟨p2,rev p1 @ rev C2 @ C1 @ q,t⟩"
      by(fastsimp simp add:R_0_Rn_commute)
    from CL closed_exec1[OF _ 1] C2
    obtain C2' C1' where pq': "p' = C2' @ p1 @ p2 ∧ q' = C1' @ q"
      and same: "rev C1' @ C2' = rev C1 @ C2"
      by fastsimp
    have rev_same: "rev C2' @ C1' = rev C2 @ C1"
    proof -
      have "rev C2' @ C1' = rev(rev C1' @ C2')" by simp
      also have "... = rev(rev C1 @ C2)" by(simp only:same)
      also have "... = rev C2 @ C1" by simp
      finally show ?thesis .
    qed
    hence rev_same': "∧p. rev C2' @ C1' @ p = rev C2 @ C1 @ p" by simp
    from n have n': "⟨C2' @ p1 @ p2,C1' @ q,r'⟩ -n→
      ⟨p2,rev p1 @ rev C2' @ C1' @ q,t⟩"
      by(simp add:pq' rev_same')

```

```

from IH[OF _ n'] CL
obtain s n1 n2 where n1: "<C2',C1',r'> -n1→ <[],rev C2 @ C1,s>" and
  "<p1 @ p2,rev C2 @ C1 @ q,s> -n2→ <p2,rev p1 @ rev C2 @ C1 @ q,t> ∧
  n = n1 + n2"
  by(fastsimp simp add: same_rev_same rev_same')
moreover
from 1 n1 pq' C2 have "<C2,C1,r> -Suc n1→ <[],rev C2 @ C1,s>"
  by (simp del:relpow.simps exec_simp) (fast dest:reduce_exec1)
ultimately show ?thesis by (fastsimp simp del:relpow.simps)
qed
qed

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"
using closed_execn_decomp[of "[]" ,simplified] by simp

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>"
by(simp add:rtrancl_is_UN_rel_pow)(fast dest: execn_decomp)

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 (induct c)
  fix s t
  assume "<compile SKIP,[],s> -*→ <[],rev(compile SKIP),t>"
  thus "<SKIP,s> →c t" by simp
next
  fix s t v f
  assume "<compile(v ::= f),[],s> -*→ <[],rev(compile(v ::= f)),t>"
  thus "<v ::= f,s> →c t" by simp
next
  fix s1 s3 c1 c2
  let ?C1 = "compile c1" let ?C2 = "compile c2"
  assume IH1: "∧ s t. <?C1,[],s> -*→ <[],rev ?C1,t> ⇒ <c1,s> →c t"
    and IH2: "∧ s t. <?C2,[],s> -*→ <[],rev ?C2,t> ⇒ <c2,s> →c t"
  assume "<compile(c1;c2),[],s1> -*→ <[],rev(compile(c1;c2)),s3>"
  then obtain s2 where exec1: "<?C1,[],s1> -*→ <[],rev ?C1,s2>" and
    exec2: "<?C2,rev ?C1,s2> -*→ <[],rev(compile(c1;c2)),s3>"
    by(fastsimp dest:exec_star_decomp[of _ _ "[]" "[]" ,simplified])
  from exec2 have exec2': "<?C2,[],s2> -*→ <[],rev ?C2,s3>"
    using exec_star_decomp[of _ "[]" "[]" ] by fastsimp
  have "<c1,s1> →c s2" using IH1 exec1 by simp
  moreover have "<c2,s2> →c s3" using IH2 exec2' by fastsimp
  ultimately show "<c1;c2,s1> →c s3" ..

```

```

next
  fix s t b c1 c2
  let ?if = "IF b THEN c1 ELSE c2" let ?C = "compile ?if"
  let ?C1 = "compile c1" let ?C2 = "compile c2"
  assume IH1: " $\bigwedge s t. \langle ?C1, [], s \rangle \text{-}^* \rightarrow \langle [], \text{rev } ?C1, t \rangle \implies \langle c1, s \rangle \rightarrow_c t$ "
    and IH2: " $\bigwedge s t. \langle ?C2, [], s \rangle \text{-}^* \rightarrow \langle [], \text{rev } ?C2, t \rangle \implies \langle c2, s \rangle \rightarrow_c t$ "
    and H: " $\langle ?C, [], s \rangle \text{-}^* \rightarrow \langle [], \text{rev } ?C, t \rangle$ "
  show " $\langle ?if, s \rangle \rightarrow_c t$ "
  proof cases
    assume b: "b s"
    with H have " $\langle ?C1, [], s \rangle \text{-}^* \rightarrow \langle [], \text{rev } ?C1, t \rangle$ "
      by (fastsimp dest:exec_star_decomp
        [of _ "[JMPF ( $\lambda x. \text{False}$ ) (size ?C2)]@?C2" "[]",simplified])
    hence " $\langle c1, s \rangle \rightarrow_c t$ " by(rule IH1)
    with b show ?thesis ..
  next
    assume b: " $\neg b s$ "
    with H have " $\langle ?C2, [], s \rangle \text{-}^* \rightarrow \langle [], \text{rev } ?C2, t \rangle$ "
      using exec_star_decomp[of _ "[]" "[]"] by simp
    hence " $\langle c2, s \rangle \rightarrow_c t$ " by(rule IH2)
    with b show ?thesis ..
  qed
next
  fix b c s t
  let ?w = "WHILE b DO c" let ?W = "compile ?w" let ?C = "compile c"
  let ?j1 = "JMPF b (size ?C + 1)" let ?j2 = "JMPB (size ?C + 1)"
  assume IHc: " $\bigwedge s t. \langle ?C, [], s \rangle \text{-}^* \rightarrow \langle [], \text{rev } ?C, t \rangle \implies \langle c, s \rangle \rightarrow_c t$ "
    and H: " $\langle ?W, [], s \rangle \text{-}^* \rightarrow \langle [], \text{rev } ?W, t \rangle$ "
  from H obtain k where ob: " $\langle ?W, [], s \rangle \text{-}k \rightarrow \langle [], \text{rev } ?W, t \rangle$ "
    by(simp add:rtrancl_is_UN_rel_pow) blast
  { fix n have " $\bigwedge s. \langle ?W, [], s \rangle \text{-}n \rightarrow \langle [], \text{rev } ?W, t \rangle \implies \langle ?w, s \rangle \rightarrow_c t$ "
    proof (induct n rule: less_induct)
      fix n
      assume IHm: " $\bigwedge m s. [m < n; \langle ?W, [], s \rangle \text{-}m \rightarrow \langle [], \text{rev } ?W, t \rangle] \implies \langle ?w, s \rangle \rightarrow_c t$ "
      fix s
      assume H: " $\langle ?W, [], s \rangle \text{-}n \rightarrow \langle [], \text{rev } ?W, t \rangle$ "
      show " $\langle ?w, s \rangle \rightarrow_c t$ "
      proof cases
        assume b: "b s"
        then obtain m where m: "n = Suc m"
          and " $\langle ?C @ [?j2], [?j1], s \rangle \text{-}m \rightarrow \langle [], \text{rev } ?W, t \rangle$ "
          using H by fastsimp
        then obtain r n1 n2 where n1: " $\langle ?C, [], s \rangle \text{-}n1 \rightarrow \langle [], \text{rev } ?C, r \rangle$ "
          and n2: " $\langle [?j2], \text{rev } ?C @ [?j1], r \rangle \text{-}n2 \rightarrow \langle [], \text{rev } ?W, t \rangle$ "
          and n12: "m = n1+n2"
          using execn_decomp[of _ "[?j2]"]
          by(simp del: execn_simp) fast
        have n2n: "n2 - 1 < n" using m n12 by arith
        note b
        moreover

```

```

    { from n1 have "<?C, [], s> -*→ <[], rev ?C, r>"
      by (simp add:rtrancl_is_UN_rel_pow) fast
      hence "<c, s> →c r" by(rule IHc)
    }
    moreover
    { have "n2 - 1 < n" using m n12 by arith
      moreover from n2 have "<?W, [], r> -n2- 1→ <[], rev ?W, t>" by fastsimp
      ultimately have "<?w, r> →c t" by(rule IHm)
    }
    ultimately show ?thesis ..
  next
    assume b: "¬ b s"
    hence "t = s" using H by simp
    with b show ?thesis by simp
  qed
}
qed
}
with ob show "<?w, s> →c t" by fast
qed

```

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.