Semantics

TN

January 28, 2011

Contents

1 Arithmetic and Boolean Expressions 4
   1.1 Arithmetic Expressions ............................... 4
   1.2 Optimization ........................................... 4
   1.3 Boolean Expressions ................................... 5
   1.4 Optimization ........................................... 5

2 Arithmetic Stack Machine andCompilation 7
   2.1 Arithmetic Stack Machine ............................... 7
   2.2 Compilation ............................................. 7

3 IMP — A Simple Imperative Language 8
   3.1 From functions to lists ................................... 8
   3.2 Big-Step Semantics of Commands .......................... 9
   3.3 Rule inversion ............................................ 11
   3.4 Command Equivalence ................................... 12
   3.5 Execution is deterministic ............................... 14

4 Small-Step Semantics of Commands 14
   4.1 The transition relation ................................... 14
   4.2 Executability ............................................. 15
   4.3 Proof infrastructure ..................................... 15
      4.3.1 Induction rules ...................................... 15
      4.3.2 Proof automation .................................... 15
   4.4 Equivalence with big-step semantics ...................... 16
   4.5 Final configurations and infinite reductions ............ 19

5 A Compiler for IMP 19
   5.1 Instructions and Stack Machine .......................... 19
   5.2 Verification infrastructure .............................. 21
   5.3 Compilation ............................................. 23
   5.4 Preservation of semantics ................................ 24
12 Hoare Logic for Total Correctness

13 Extensions and Variations of IMP

13.1 Procedures and Local Variables ........................................ 66
  13.1.1 Dynamic Scoping of Procedures and Variables ........... 67
  13.1.2 Static Scoping of Procedures, Dynamic of Variables ... 68
  13.1.3 Static Scoping of Procedures and Variables .......... 69

13.2 A C-like Language ............................................................. 70

13.3 Towards an OO Language: A Language of Records .......... 72
1 Arithmetic and Boolean Expressions

theory AExp imports Main begin

1.1 Arithmetic Expressions

types
  name = nat — For simplicity in examples
  state = name ⇒ nat

datatype aexp = N nat | V name | Plus aexp aexp

fun aval :: aexp ⇒ state ⇒ nat where
  aval (N n) = n |
  aval (V x) st = st x |
  aval (Plus e1 e2) st = aval e1 st + aval e2 st

Subscripts are for visual beauty only!

value aval (Plus (V 0) (N 5)) (nth [2,1])

1.2 Optimization

Evaluate constant subexpressions:

fun asimp-const :: aexp ⇒ aexp where
  asimp-const (N n) = N n |
  asimp-const (V x) = V x |
  asimp-const (Plus e1 e2) =
    (case (asimp-const e1, asimp-const e2) of
    (N n1, N n2) ⇒ N(n1+n2) |
    (e1',e2') ⇒ Plus e1' e2')

theorem aval-asimp-const[simp]:
  aval (asimp-const a) st = aval a st
apply (induct a)
apply (auto split: aexp.split)
done

Now we also eliminate all occurrences 0 in additions. The standard
method: optimized versions of the constructors:

fun plus :: aexp ⇒ aexp ⇒ aexp where
  plus (N 0) e = e |
  plus e (N 0) = e |
  plus (N n1) (N n2) = N(n1+n2) |
  plus e1 e2 = Plus e1 e2
lemma aval-plus[simp]:
  aval (plus e1 e2) st = aval e1 st + aval e2 st
apply (induct e1 e2 rule: plus.induct)
apply simp-all
done

fun asimp :: aexp ⇒ aexp where
asimp (N n) = N n |
asimp (V x) = V x |
asimp (Plus e1 e2) = plus (asimp e1) (asimp e2)

  Note that in asimp-const the optimized constructor was inlined. Making
it a separate function AExp.plus improves modularity of the code and the
proofs.
value asimp (Plus (Plus (N 0) (N 0)) (Plus (V 5) (N 0)))

theorem aval-asimp[simp]:
  aval (asimp a) st = aval a st
apply (induct a)
apply simp-all
done

done

theory BExp imports AExp begin

1.3 Boolean Expressions

datatype bexp = B bool | Not bexp | And bexp bexp | Less aexp aexp

primrec bval :: bexp ⇒ state ⇒ bool where
bval (B bv) - = bv |
bval (Not b) st = (¬ bval b st) |
bval (And b1 b2) st = (if bval b1 st then bval b2 st else False) |
bval (Less a1 a2) st = (aval a1 st < aval a2 st)

value bval (Less (V 1) (Plus (N 3) (V 0))) (nth [1,3])

1.4 Optimization

Optimized constructors:

fun less :: aexp ⇒ aexp ⇒ bexp where
less (N n1) (N n2) = B(n1 < n2) |
less a1 a2 = Less a1 a2

**lemma** [simp]: bval (less a1 a2) st = (aval a1 st < aval a2 st)
**apply** (induct a1 a2 rule: less.induct)
**apply** simp-all
**done**

**fun** and :: bexp ⇒ bexp ⇒ bexp where
and (B True) b = b |
and b (B True) = b |
and (B False) b = B False |
and b (B False) = B False |
and b1 b2 = And b1 b2

**lemma** bval-and[simp]: bval (and b1 b2) st = (bval b1 st & bval b2 st)
**apply** (induct b1 b2 rule: and.induct)
**apply** simp-all
**done**

**fun** not :: bexp ⇒ bexp where
not (B True) = B False |
not (B False) = B True |
not b = Not b

**lemma** bval-not[simp]: bval (not b) st = (∼bval b st)
**apply** (induct b rule: not.induct)
**apply** simp-all
**done**

Now the overall optimizer:

**fun** bsimp :: bexp ⇒ bexp where
bsimp (Less a1 a2) = less (asimp a1) (asimp a2) |
bsimp (And b1 b2) = and (bsimp b1) (bsimp b2) |
bsimp (Not b) = not(bsimp b) |
bsimp (B bv) = B bv

**value** bsimp (And (Less (N 0) (N 1)) b)

**value** bsimp (And (Less (N 1) (N 0)) (B True))

**theorem** bval (bsimp b) st = bval b st
**apply** (induct b)
**apply** simp-all
**done**
2 Arithmetic Stack Machine and Compilation

theory ASM imports AExp begin

2.1 Arithmetic Stack Machine

datatype ainstr = PUSH-N nat | PUSH-V nat | ADD

types stack = nat list

abbreviation hd2 xs == hd(tl xs)
abbreviation tl2 xs == tl(tl xs)

Abbreviations are transparent: they are unfolded after parsing and folded back again before printing. Internally, they do not exist.

fun aexec1 :: ainstr ⇒ state ⇒ stack ⇒ stack where
aexec1 (PUSH-N n) - stk = n # stk |
aexec1 (PUSH-V n) s stk = s(n) # stk |
aexec1 ADD - stk = (hd2 stk + hd stk) # tl2 stk

fun aexec :: ainstr list ⇒ state ⇒ stack ⇒ stack where
aexec [] - stk = stk |
aexec (i#is) s stk = aexec is s (aexec1 i s stk)

value aexec [PUSH-N 5, PUSH-V 2, ADD] (nth[42,43,44]) [50]

lemma aexec-append[simp]:
  aexec (is1@is2) s stk = aexec is2 s (aexec is1 s stk)
apply (induct is1 arbitrary: stk)
apply (auto)
done

2.2 Compilation

fun acomp :: aexp ⇒ ainstr list where
acomp (N n) = [PUSH-N n] |
acomp (V n) = [PUSH-V n] |
acomp (Plus e1 e2) = acomp e1 @ acomp e2 @ [ADD]

value acomp (Plus (Plus (V 0) (N 1)) (V 2))
theorem aexec-aomp [simp]: aexec (aomp e) s stk = aval e s ≠ stk
apply (induct e arbitrary: stk)
apply (auto)
done
end

3 IMP — A Simple Imperative Language

theory Com imports BExp begin

datatype
  com = SKIP
  | Assign name aexp (- ::= - [1000, 61] 61)
  | Semi com com (-:: - [60, 61] 60)
  | If bexp com com ((IF -/ THEN -/ ELSE -) [0, 0, 61] 61)
  | While bexp com ((WHILE -/ DO -) [0, 61] 61)
end

theory Util imports Main begin

3.1 From functions to lists

value [0 ..< 3]

value map f [0 ..< 3]

definition list :: (nat ⇒ 'a) ⇒ nat ⇒ 'a list where
list s n = map s [0 ..< n]

value list f 3
end

theory Big-Step imports Com Util begin
3.2 Big-Step Semantics of Commands

inductive
big-step :: com \times state \Rightarrow state \Rightarrow bool (infix \Rightarrow 55)
where
Skip: \ (SKIP, s) \Rightarrow s |
Assign: \ (x ::= a, s) \Rightarrow s(x ::= aval a s) |
Semi: \ [ (c_1, s_1) \Rightarrow s_2; (c_2, s_2) \Rightarrow s_3 ] \Rightarrow (c_1; c_2, s_1) \Rightarrow s_3 |
IfTrue: \ [ bval b s; (c_1, s) \Rightarrow t ] \Rightarrow (IF b THEN c_1 ELSE c_2, s) \Rightarrow t |
IfFalse: \ [ \neg bval b s; (c_2, s) \Rightarrow t ] \Rightarrow (IF b THEN c_1 ELSE c_2, s) \Rightarrow t |
WhileFalse: \neg bval b s \Rightarrow (WHILE b DO c, s) \Rightarrow s |
WhileTrue: \ [ bval b s_1; (c, s_1) \Rightarrow s_2; (WHILE b DO c, s_2) \Rightarrow s_3 ] \Rightarrow (WHILE b DO c, s_1) \Rightarrow s_3 |

schematic-lemma ex: (0 ::= N 5; 2 ::= V 0, s) \Rightarrow ?t
apply (rule Semi)
apply (rule Assign)
simp
apply (rule Assign)
done

thm ex[simplified]

We want to execute the big-step rules:

code-pred big-step .

For inductive definitions we need command values instead of value.

values \{ t. (SKIP, nth[4]) \Rightarrow t \}

We need to translate the result state into a list to display it. See function list in Util.

inductive exec where
(c, nth ns) \Rightarrow s \Rightarrow exec c ns (list s (length ns))

code-pred exec .

values \{ ns. exec SKIP [42,43] ns \}

values \{ ns. exec (0 ::= N 2) [0] ns \}
values \{ns.

exec

(WHILE Less (V 0) (V 1) DO (0 := Plus (V 0) (N 5)))

[0,1][ ns]

Note: exec only defined for executing the semantics, not for proofs.

Proof automation:

declare big-step.intros [intro]

The standard induction rule

\[x1 \Rightarrow x2; \ \ \wedge s. \ P \ (\text{SKIP}, \ s) \ s; \ \ \wedge x a s. \ P \ (x := a, \ s) \ (s(x := \text{aval} \ a \ s)); \ \ \wedge c_1 s_1 s_2 c_2 s_3.

\[[(c_1, s_1) \Rightarrow s_2; \ P \ (c_1, s_1) s_2; (c_2, s_2) \Rightarrow s_3; \ P \ (c_2, s_2) s_3]\]

\[\Rightarrow P \ (c_1; c_2, s_1) s_3;\]

\[\wedge b s c_1 t c_2.

\[\text{bval} \ b \ s; (c_1, s) \Rightarrow t; \ P \ (c_1, s) t \ \Rightarrow \ P \ (\text{IF} \ b \ \text{THEN} \ c_1 \ \text{ELSE} \ c_2, \ s) \ t;\]

\[\wedge b s c_2 t c_1.

\[\neg\ \text{bval} \ b \ s; (c_2, s) \Rightarrow t; \ P \ (c_2, s) t \ \Rightarrow \ P \ (\text{IF} \ b \ \text{THEN} \ c_1 \ \text{ELSE} \ c_2, \ s) \ t;\]

\[\wedge b s c. \neg \ \text{bval} \ b \ s \ \Rightarrow \ P \ (\text{WHILE} \ b \ \text{DO} \ c, \ s) \ s;\]

\[\wedge b s c_1 c s_2 s_3.

\[\text{bval} \ b \ s_1; (c, s_1) \Rightarrow s_2; \ P \ (c, s_1) s_2; (\text{WHILE} \ b \ \text{DO} \ c, s_2) \Rightarrow s_3; \]

\[P \ (\text{WHILE} \ b \ \text{DO} \ c, s_2) s_3]\]

\[\Rightarrow P \ (\text{WHILE} \ b \ \text{DO} \ c, s_1) s_3]\]

\[\Rightarrow P \ x1 \ x2\]

thm big-step.induct

A customized induction rule for (c,s) pairs:

lemmas big-step-induct = big-step.induct[split-format(complete)]

thm big-step-induct

\[[(x1a, x1b) \Rightarrow x2a; \ \ \wedge s. \ P \ \text{SKIP} \ s \ s; \ \ \wedge x a s. \ P \ (x := a) \ s \ (s(x := \text{aval} \ a \ s)); \]

\[\wedge c_1 s_1 s_2 c_2 s_3.

\[[(c_1, s_1) \Rightarrow s_2; \ P \ (c_1, s_1) s_2; (c_2, s_2) \Rightarrow s_3; \ P \ (c_2, s_2) s_3]\]

\[\Rightarrow P \ (c_1; c_2, s_1) s_3;\]

\[\wedge b s c_1 t c_2.

\[\text{bval} \ b \ s; (c_1, s) \Rightarrow t; \ P \ c_1 s t \ \Rightarrow \ P \ (\text{IF} \ b \ \text{THEN} \ c_1 \ \text{ELSE} \ c_2) \ s \ t;\]

\[\wedge b s c_2 t c_1.

\[\neg \ \text{bval} \ b \ s; (c_2, s) \Rightarrow t; \ P \ c_2 s t \ \Rightarrow \ P \ (\text{IF} \ b \ \text{THEN} \ c_1 \ \text{ELSE} \ c_2) \ s \ t;\]

\[\wedge b s c. \neg \ \text{bval} \ b \ s \ \Rightarrow \ P \ (\text{WHILE} \ b \ \text{DO} \ c) \ s \ s;\]

10
\( b \ s_1 \ c \ s_2 \ s_3. \\
[eval \ b \ s_1; (c, s_1) \Rightarrow s_2; P \ c \ s_1 \ s_2; (WHILE \ b \ DO \ c, s_2) \Rightarrow s_3; \\
P (WHILE \ b \ DO \ c) \ s_2 \ s_3] \\
\Rightarrow P (WHILE \ b \ DO \ c) \ s_1 \ s_3] \\
\Rightarrow P x1a x1b x2a

3.3 Rule inversion

What can we deduce from \((SKIP, s) \Rightarrow t\)? That \(s = t\). This is how we can automatically prove it:

**inductive-cases skipE[elim!]:** \((SKIP, s) \Rightarrow t\)  
**thm skipE**

This is an *elimination rule*. The [elim] attribute tells auto, blast and friends (but not simp!) to use it automatically; [elim!] means that it is applied eagerly.

Similarly for the other commands:

**inductive-cases AssignE[elim!]:** \((x := a, s) \Rightarrow t\)  
**thm AssignE**

**inductive-cases SemiE[elim!]:** \((c1; c2, s) \Rightarrow s3\)  
**thm SemiE**

**inductive-cases IfE[elim!]:** \((IF \ b \ THEN \ c1 \ ELSE \ c2, s) \Rightarrow t\)  
**thm IfE**

**inductive-cases WhileE[elim!]:** \((WHILE \ b \ DO \ c, s) \Rightarrow t\)  
**thm WhileE**

Only [elim]: [elim!] would not terminate.

An automatic example:

**lemma** \((IF \ b \ THEN \ SKIP \ ELSE \ SKIP, s) \Rightarrow t \Rightarrow t = s\)  
**by blast**

Rule inversion by hand via the “cases” method:

**lemma assumes** \((IF \ b \ THEN \ SKIP \ ELSE \ SKIP, s) \Rightarrow t\)  
**shows** \(t = s\)  
**proof—**  
**from** assms **show** \(?thesis\)**

**proof cases** — inverting assms  
**case** IfTrue **thm** IfTrue  
**thus** \(?thesis\) **by** blast  
**next**

**case** IfFalse **thus** \(?thesis\) **by** blast  
**qed**

**qed**
3.4 Command Equivalence

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:

**abbreviation**

$equiv-c :: com \Rightarrow com \Rightarrow bool$ (**infix $\sim$ 50 **where**

$c \sim c' == (\forall s t. (c, s) \Rightarrow t = (c', s) \Rightarrow t)$

Warning: $\sim$ is the symbol written \ < s i m > (without spaces).

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

**lemma unfold-while:**

$(WHILE b DO c) \sim (IF b THEN c; WHILE b DO c ELSE SKIP) (is \ ?w \sim \ ?iw)$

**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 $s$ $t$ assume $(\?w, s) \Rightarrow t$

  — as a first thing we note that, if $b$ is False in state $s$,

  — then both statements do nothing:

  {  
    assume $\neg bval b s$

    hence $t = s$ using $(\?w, s) \Rightarrow t$ by blast

    hence $(\?iw, s) \Rightarrow t$ using $(\neg bval b s)$ by blast

  }

  moreover

  — on the other hand, if $b$ is True in state $s$,

  — then only the WhileTrue rule can have been used to derive $(\?w, s)$

  $\Rightarrow t$

  {  
    assume $bval b s$

    with $(\?w, s) \Rightarrow t$ obtain $s'$ where

    $(c, s) \Rightarrow s'$ and $(\?w, s') \Rightarrow t$ by auto

    — now we can build a derivation tree for the IF

    — first, the body of the True-branch:

    hence $(c; ?w, s) \Rightarrow t$ by (rule Semi)

    — then the whole IF

    with $(bval b s)$ have $(\?iw, s) \Rightarrow t$ by (rule IfTrue)

  }

  ultimately

  — both cases together give us what we want:

  have $(\?iw, s) \Rightarrow t$ by blast

  }

moreover

— now the other direction:
\[
\{ \text{fix } s \ t \ \text{assume } (?i w, s) \Rightarrow t \\
\quad \text{— again, if } b \text{ is } False \text{ in state } s, \text{ then the False-branch} \\
\quad \text{— of the } IF \text{ is executed, and both statements do nothing:} \\
\{ \text{assume } \neg bval \ b \ s \\
\quad \text{hence } s = t \text{ using } (?i w, s) \Rightarrow t \text{ by blast} \\
\quad \text{hence } (?w, s) \Rightarrow t \text{ using } (\neg bval \ b \ s) \text{ by blast} \\
\} \\
\quad \text{moreover} \\
\quad \text{— on the other hand, if } b \text{ is } True \text{ in state } s, \\
\quad \text{— then this time only the } IfTrue \text{ rule can have be used} \\
\{ \text{assume } bval \ b \ s \\
\quad \text{with } (?i w, s) \Rightarrow t \text{ have } (c; ?w, s) \Rightarrow t \text{ by auto} \\
\quad \text{— and for this, only the Semi-rule is applicable:} \\
\quad \text{then obtain } s’ \text{ where} \\
\quad \ (c, s) \Rightarrow s’ \text{ and } (?w, s’) \Rightarrow t \text{ by auto} \\
\quad \text{— with this information, we can build a derivation tree for the } WHILE \\
\quad \text{with } (bval \ b \ s) \\
\quad \text{have } (?w, s) \Rightarrow t \text{ by (rule WhileTrue)} \\
\} \text{ ultimately} \\
\quad \text{— both cases together again give us what we want:} \\
\quad \text{have } (?w, s) \Rightarrow t \text{ by blast} \\
\} \\
\quad \text{ultimately} \\
\quad \text{show } \text{thesis by blast} \\
\text{qed} 
\]

Luckily, such lengthy proofs are seldom necessary. Isabelle can prove many such facts automatically.

\textbf{lemma}

\[(WHILE \ b \ DO \ c) \sim (IF \ b \ THEN \ c; \ WHILE \ b \ DO \ c \ ELSE \ SKIP)\]

by blast

\textbf{lemma} \ triv-if:

\[(IF \ b \ THEN \ c \ ELSE \ c) \sim c\]

by blast

\textbf{lemma} \ commute-if:

\[(IF \ b1 \ THEN \ (IF \ b2 \ THEN \ c11 \ ELSE \ c12) \ ELSE \ c2) \sim (IF \ b2 \ THEN \ (IF \ b1 \ THEN \ c11 \ ELSE \ c2) \ ELSE \ (IF \ b1 \ THEN \ c12 \ ELSE \ c2))\]

by blast
3.5 Execution is deterministic

This proof is automatic.

**Theorem** \( \text{big-step-determ}: \[ (c,s) \Rightarrow t; (c,s) \Rightarrow u \] \Rightarrow u = t \)

**Proof** (induct arbitrary: \( u \) rule: big-step.induct)

apply blast+
done

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

**Theorem**

\( (c,s) \Rightarrow t \Rightarrow (c,s) \Rightarrow t' \Rightarrow t' = t \)

**Proof** (induct arbitrary: \( t' \) rule: big-step.induct)

— the only interesting case, \( \text{WhileTrue} \):

fix \( b c s s1 t t' \)

— The assumptions of the rule:

assume \( \text{bval} b s \) and \( (c,s) \Rightarrow s1 \) and \((\text{WHILE} b \text{ DO} c, s1) \Rightarrow t \)

— Ind.Hyp; note the \( \wedge \) because of arbitrary:

assume \( \text{IHc: } \wedge t'. (c,s) \Rightarrow t' \Rightarrow t' = s1 \)

assume \( \text{IHw: } \wedge t'. (\text{WHILE} b \text{ DO} c, s1) \Rightarrow t' \Rightarrow t' = t \)

— Premise of implication:

assume \( (\text{WHILE} b \text{ DO} c, s) \Rightarrow t' \)

with \( \langle \text{bval} b s \rangle \) obtain \( s1' \) where

\( c: (c,s) \Rightarrow s1' \text{ and } \)

\( w: (\text{WHILE} b \text{ DO} c, s1') \Rightarrow t' \)

by auto

from \( c \text{ IHc have } s1' = s1 \) by blast

with \( w \text{ IHw show } t' = t \) by blast

qed blast+ — prove the rest automatically

end

4 Small-Step Semantics of Commands

**Theory** \( \text{Small-Step imports Big-Step begin} \)

4.1 The transition relation

**Inductive**

\( \text{small-step :: com * state } \Rightarrow \text{ com * state } \Rightarrow \text{ bool (infix } \to 55) \)

**Where**

Assign: \( (x := a, s) \to (\text{SKIP, } s(x := \text{aval a s})) \)
Semi1: \((\text{SKIP};c_2,s) \to (c_2,s)\) |
Semi2: \((c_1,s) \to (c_1',s) \implies (c_1;c_2,s) \to (c_1';c_2,s')\) |

IfTrue: \(\text{bval } b \; s \implies (\text{IF } b \; \text{THEN } c_1 \; \text{ELSE } c_2,s) \to (c_1,s)\) |
IfFalse: \(\neg \text{bval } b \; s \implies (\text{IF } b \; \text{THEN } c_1 \; \text{ELSE } c_2,s) \to (c_2,s)\) |

While: \((\text{WHILE } b \; \text{DO } c_1,s) \to (\text{IF } b \; \text{THEN } c_1; \; \text{WHILE } b \; \text{DO } c \; \text{ELSE } \text{SKIP};s)\)

inductive small-steps :: com * state ⇒ com * state ⇒ bool (infix →∗ 55) where
refl: cs →∗ cs |
step: cs → cs' ⇒ cs' →∗ cs'' ⇒ cs →∗ cs''

4.2 Executability

code-pred small-step .
code-pred small-steps .

inductive execl :: com ⇒ nat list ⇒ com ⇒ nat list ⇒ bool where
small-steps (c,nth ns) (c',t) ⇒ execl c ns c' (list t (size ns))

code-pred execl .

values \{(c',t) . execl (0 ::= V 2; 1 ::= V 0) [3,7,5] c' t\}

4.3 Proof infrastructure

4.3.1 Induction rules

The default induction rule small-step.induct only works for lemmas of the form \(a \to b \implies \ldots\) where \(a\) and \(b\) are not already pairs (DUMMY,DUMMY).

We can generate a suitable variant of small-step.induct for pairs by “splitting” the arguments \(\to\) into pairs:

lemmas small-step-induct = small-step.induct[split-format(complete)]

Similarly for \(\to∗\):

lemmas small-steps-induct = small-steps.induct[split-format(complete)]

4.3.2 Proof automation

declare small-step.intros[simp,intro]
declare small-steps.refl[simp,intro]

15
lemma step1 [simp, intro]: \( cs \to cs' \Rightarrow cs \to^* cs' \)
by (metis refl step)

So called transitivity rules. See below.

declare step [trans] step1 [trans]

lemma step2 [trans]:
\( cs \to cs' \Rightarrow cs' \to cs'' \Rightarrow cs \to^* cs'' \)
by (metis refl step)

lemma small-steps-trans [trans]:
\( cs \to^* cs' \Rightarrow cs' \to^* cs'' \Rightarrow cs \to^* cs'' \)
proof (induct rule: small-steps.induct)
  case refl thus \(?case\)
next
  case step thus \(?case\) by (metis small-steps.step)
qed

Rule inversion:

inductive-cases SkipE [elim!]: (SKIP, s) \to ct
thm SkipE
inductive-cases AssignE [elim!]: (x::=a, s) \to ct
thm AssignE
inductive-cases SemiE [elim]: (c1; c2, s) \to ct
thm SemiE
inductive-cases IfE [elim!]: (IF b THEN c1 ELSE c2, s) \to ct
inductive-cases WhileE [elim]: (WHILE b DO c, s) \to ct

A simple property:

lemma deterministic:
\( cs \to cs' \Rightarrow cs \to cs'' \Rightarrow cs'' = cs' \)
apply (induct arbitrary: \(cs''\) rule: small-step.induct)
apply blast
done

4.4 Equivalence with big-step semantics

lemma rtrancl-semi2: \((c1,s) \to^* (c1',s') \Rightarrow (c1; c2, s) \to^* (c1'; c2, s') \)
proof (induct rule: small-steps-induct)
  case refl thus \(?case\) by simp
next
  case step
  thus \(?case\) by (metis Semi2 small-steps.step)
qed
lemma semi-comp:
\[
\begin{align*}
\left( (c_1, s_1) \rightarrow^\ast (\text{SKIP}, s_2) \right) \quad &\text{and} \quad \left( (c_2, s_2) \rightarrow^\ast (\text{SKIP}, s_3) \right) \\
\implies (c_1 ; c_2, s_1) \rightarrow^\ast (\text{SKIP}, s_3)
\end{align*}
\]

by (blast intro: small-steps.step rtrancl-semi2 small-steps-trans)

The following proof corresponds to one on the board where one would show chains of \( \rightarrow \) and \( \rightarrow^\ast \) steps. This is what the also/finally proof steps do: they compose chains, implicitly using the rules declared with attribute [trans] above.

lemma big-to-small:
\[
\begin{align*}
c s \Rightarrow t \implies c s \rightarrow^\ast (\text{SKIP}, t)
\end{align*}
\]

proof (induct rule: big-step.induct)

fix s show (\( \text{SKIP} \), s) \( \rightarrow^\ast \) (SKIP, s) by simp

next

fix x a s show (x := a, s) \( \rightarrow^\ast \) (\( \text{SKIP} \), s(x := aval a s)) by auto

next

fix c1 c2 s1 s2 s3
assume (c1, s1) \( \rightarrow^\ast \) (\( \text{SKIP} \), s2) and (c2, s2) \( \rightarrow^\ast \) (\( \text{SKIP} \), s3)
thus (c1 ; c2, s1) \( \rightarrow^\ast \) (\( \text{SKIP} \), s3) by (rule semi-comp)

next

fix s::state and b c0 c1 t
assume bval b s
hence (IF b THEN c0 ELSE c1, s) \( \rightarrow \) (c0, s) by simp
also assume (c0, s) \( \rightarrow^\ast \) (\( \text{SKIP} \), t)
finally show (IF b THEN c0 ELSE c1, s) \( \rightarrow^\ast \) (\( \text{SKIP} \), t) . — = by assumption

next

fix s::state and b c0 c1 t
assume \( \neg \) bval b s
hence (IF b THEN c0 ELSE c1, s) \( \rightarrow \) (c1, s) by simp
also assume (c1, s) \( \rightarrow^\ast \) (\( \text{SKIP} \), t)
finally show (IF b THEN c0 ELSE c1, s) \( \rightarrow^\ast \) (\( \text{SKIP} \), t) .

next

fix b c and s::state
assume b: \( \neg \) bval b s
let ?if = IF b THEN c; WHILE b DO c ELSE SKIP
have (WHILE b DO c, s) \( \rightarrow \) (?if, s) by blast
also have (?if, s) \( \rightarrow \) (\( \text{SKIP} \), s) by (simp add: b)
finally show (WHILE b DO c, s) \( \rightarrow^\ast \) (\( \text{SKIP} \), s) by auto

next

fix b c s s' t
let ?w = WHILE b DO c
let ?if = IF b THEN c; ?w ELSE SKIP
assume \( w : (\text{?w}, s) \rightarrow^* (\text{SKIP}, t) \)
assume \( c: (c, s) \rightarrow^* (\text{SKIP}, s') \)
assume \( b: \text{bval} b s \)

have \( (\text{?w}, s) \rightarrow (\text{?if}, s) \) by blast
also have \( (\text{?if}, s) \rightarrow (c; ?w, s) \) by (simp add: b)
also have \( (c; ?w, s) \rightarrow^* (\text{SKIP}, t) \) by (rule semi-comp[OF c w])
finally show \( \text{WHILE} b \text{ DO } c, s \rightarrow^* (\text{SKIP}, t) \) by auto

qed

Each case of the induction can be proved automatically:

lemma \( cs \Rightarrow t \Longrightarrow cs \rightarrow^* (\text{SKIP}, t) \)
proof (induct rule: big-step.induct)
case Skip show ?case by blast
next
case Assign show ?case by blast
next
case Semi thus ?case by (blast intro: semi-comp)
next
case IfTrue thus ?case by (blast intro: step)
next
case IfFalse thus ?case by (blast intro: step)
next
case WhileFalse thus ?case
by (metis step step1 small-step.IfFalse small-step.While)
next
case WhileTrue
thus ?case
by (metis While semi-comp small-step.IfTrue step[OF (a,b),standard])

qed

lemma small1-big-continue:
\( cs \rightarrow cs' \Longrightarrow cs' \Rightarrow t \Longrightarrow cs \Rightarrow t \)
apply (induct arbitrary: \( t \) rule: small-step.induct)
apply auto
done

lemma small-big-continue:
\( cs \rightarrow^* cs' \Longrightarrow cs' \Rightarrow t \Longrightarrow cs \Rightarrow t \)
apply (induct rule: small-steps.induct)
apply (auto intro: small1-big-continue)
done

lemma small-to-big: \( cs \rightarrow^* (\text{SKIP}, t) \Longrightarrow cs \Rightarrow t \)
Finally, the equivalence theorem:

**Theorem big-iff-small:**

\[ cs \Rightarrow t = cs \rightarrow^* (\text{SKIP},t) \]

**by** (metis big-to-small small-to-big)

### 4.5 Final configurations and infinite reductions

**Definition** final \( cs \leftrightarrow \neg(\exists cs'. cs \rightarrow cs') \)

**Lemma** finalD: final \( (c,s) \implies c = \text{SKIP} \)

**by** (simp add: final-def)

**Lemma** final-iff-SKIP: final \( (c,s) = (c = \text{SKIP}) \)

**by** (metis SkipE finalD final-def)

Now we can show that \( \Rightarrow \) yields a final state iff \( \rightarrow \) terminates:

**Lemma** big-iff-small-termination:

\[(\exists t. cs \Rightarrow t) \iff (\exists cs'. cs \rightarrow cs' \land \text{final } cs')\]

**by** (simp add: big-iff-small final-iff-SKIP)

This is the same as saying that the absence of a big step result is equivalent with absence of a terminating small step sequence, i.e. with nontermination. Since \( \rightarrow \) is deterministic, there is no difference between may and must terminate.

**end**

### 5 A Compiler for IMP

**Theory** Compiler imports Big-Step begin

### 5.1 Instructions and Stack Machine

**Definition** instr =

\[ \text{PUSH-N } \text{nat} \mid \text{PUSH-V } \text{nat} \mid \text{ADD} \mid \text{STORE } \text{nat} \mid \text{JMPF } \text{nat} \mid \text{JMPB } \text{nat} \mid \text{JMPFLESS } \text{nat} \]
**JMPFGE nat**

**types** stack = nat list  
  config = nat × state × stack

**abbreviation** hd2 xs == hd(tl xs)  
**abbreviation** tl2 xs == tl(tl xs)

**inductive** exec1 :: instr list ⇒ config ⇒ config ⇒ bool  
  ((/- ⊢ (- →/-)) [50,0,0] 50)  
  for P :: instr list  
  where

  ![Code here](https://example.com/code.png)

**code-pred exec1**.

**declare** exec1.intros[intro]

**inductive** exec :: instr list ⇒ config ⇒ config ⇒ bool (-/ ⊢ (- →*/ -) 50)  
  where

  ![Code here](https://example.com/code.png)

**declare** exec.intros[intro]

**lemmas** exec-induct = exec.induct[split-format(complete)]

**code-pred** exec .
Integrating the state to list transformation:

\[
\text{inductive } \text{excl} :: \text{instr list } \Rightarrow \text{nat } \Rightarrow \text{nat list } \Rightarrow \text{stack} \\
\Rightarrow \text{nat } \Rightarrow \text{nat list } \Rightarrow \text{stack } \Rightarrow \text{bool } \text{where} \\
P \vdash (i, \text{nth ns}, \text{stk}) \rightarrow (i', s', \text{stk}') \implies \\
\text{excl } P \ i \ \text{ns} \ \text{stk} \ i' (\text{list } s' \ (\text{size } \text{ns})) \ \text{stk}'
\]

code-pred excl .

values \{(i, \text{ns}, \text{stk}), \text{excl} [\text{PUSH-V 1}, \text{STORE 0}] \ 0 \ [3,4] \ [] i \ \text{ns} \ \text{stk}\}

5.2 Verification infrastructure

\text{lemma} \text{exec-trans: } P \vdash c \rightarrow c' \implies P \vdash c' \rightarrow c'' \implies P \vdash c \rightarrow c''

\text{apply (induct rule: exec.induct)}
\text{apply blast}
\text{by (metis exec.step)}

\text{lemma} \text{exec1-subst: } P \vdash c \rightarrow c' \implies c' = c'' \implies P \vdash c \rightarrow c''
\text{by auto}

\text{lemmas} \text{exec1-simps = exec1.intros[THEN exec1-subst]}

Below we need to argue about the execution of code that is embedded in larger programs. For this purpose we show that execution is preserved by appending code to the left or right of a program.

\text{lemma} \text{exec1-appendR: assumes } P \vdash c \rightarrow c' \text{ shows } P @ P' \vdash c \rightarrow c'
\text{proof –}
\text{from assms show } ?\text{thesis}
\text{by cases (simp-all add: exec1-simps nth-append)}
\text{— All cases proved with the final simp-all}
\text{qed}

\text{lemma} \text{exec-appendR: } P \vdash c \rightarrow* c' \implies P \vdash c' \rightarrow* c'' \implies P \vdash c \rightarrow* c''
\text{apply (induct rule: exec.induct)}
\text{apply blast}
\text{by (metis exec1-appendR exec.step)}

\text{lemma} \text{exec1-appendL:}
\text{assumes } P \vdash (i, s, \text{stk}) \rightarrow (i', s', \text{stk}')
\text{shows } P' @ P \vdash (\text{size}(P') + i, s, \text{stk}) \rightarrow (\text{size}(P') + i', s', \text{stk}')
\text{proof –}
\text{from assms show } ?\text{thesis}
\text{by cases (simp-all add: exec1-simps)}
\text{qed}
lemma exec-appendL:
\[ P \vdash (i,s,stk) \rightarrow^* (i',s',stk') \Rightarrow P'@P \vdash (\text{size}(P') + i,s,stk) \rightarrow^* (\text{size}(P') + i',s',stk') \]
apply (induct rule: exec-induct)
apply blast
by (blast intro: exec1-appendL exec.step)

Now we specialise the above lemmas to enable automatic proofs of
\[ P \vdash c \rightarrow^* c' \]
where \( P \) is a mixture of concrete instructions and pieces of code
that we already know how they execute (by induction), combined by \@ and
\#. Backward jumps are not supported. The details should be skipped on a
first reading.

If the pc points beyond the first instruction or part of the program, drop
it:

lemma exec-Cons-Suc[intro]:
\[ P \vdash (i,s,stk) \rightarrow^* (j,t,stk') \Rightarrow \]
\[ \text{instr}#P \vdash (\text{Suc} i,s,stk) \rightarrow^* (\text{Suc} j,t,stk') \]
apply (drule exec-appendL[where \( P'=[\text{instr}] \)])
apply simp
done

done

lemma exec-appendL-if[intro]:
size \( P' \leq i \)
\[ \Rightarrow P \vdash (i - \text{size} P',s,stk) \rightarrow^* (i',s',stk') \Rightarrow P'@P \vdash (i,s,stk) \rightarrow^* (\text{size} P' + i',s',stk') \]
apply (drule exec-appendL[where \( P'=[P'] \)])
apply simp
done

done

Split the execution of a compound program up into the execution of its
parts:

lemma exec-append-trans[intro]:
\[ P \vdash (0,s,stk) \rightarrow^* (i',s',stk') \Rightarrow \]
\[ \text{size} P \leq i \Rightarrow \]
\[ P' \vdash (\text{size} P,s',stk') \rightarrow^* (i'',s'',stk'') \Rightarrow \]
\[ j'' = \text{size} P + i'' \Rightarrow \]
\[ P @ P' \vdash (0,s,stk) \rightarrow^* (j'',s'',stk'') \]
by (metis exec-trans[OF exec-appendR exec-appendL-if])

declare Let-def[simp] nat-number[simp]
5.3 Compilation

**fun acomp :: aexp ⇒ instr list where**
- \(\text{acomp} (N n) = [\text{PUSH}-N n]\)
- \(\text{acomp} (V n) = [\text{PUSH}-V n]\)
- \(\text{acomp} (\text{Plus} a_1 a_2) = \text{acomp} a_1 \@ \text{acomp} a_2 \@ [\text{ADD}]\)

**lemma acomp-correct[intro]:**
- \(\text{acomp} a \vdash (0,s,\text{stk}) \rightarrow^* (\text{size}(\text{acomp} a),s,\text{aval} a \#\text{stk})\)

**apply(induct a arbitrary: stk)**
**apply(fastsimp)+

**done**

**fun bcomp :: bexp ⇒ bool ⇒ nat ⇒ instr list where**
- \(\text{bcomp} (B v) c n = (\text{if} v=c \text{ then } [\text{JMPF} n] \text{ else } [])\)
- \(\text{bcomp} (\text{Not} b) c n = \text{bcomp} b (\neg c) n\)
- \(\text{bcomp} (\text{And} b_1 b_2) c n =\)
  - let \(cb_2 = \text{bcomp} b_2 c n;\)
  - \(m = (\text{if} c \text{ then } \text{size} cb_2 \text{ else } \text{size} cb_2 + n);\)
  - \(cb_1 = \text{bcomp} b_1 \text{ False } m\)
  - \(\text{in } cb_1 \@ cb_2\)
- \(\text{bcomp} (\text{Less} a_1 a_2) c n =\)
  - \(\text{acomp} a_1 \@ \text{acomp} a_2 \@ (\text{if} c \text{ then } [\text{JMPFLESS } n] \text{ else } [\text{JMPFGE } n])\)

**value bcomp (And (Less (V 0) (V 1)) (Not(Less (V 2) (V 3)))) False 3**

**lemma bcomp-correct[intro]:**
- \(\text{bcomp} b c n \vdash (0,s,\text{stk}) \rightarrow^* (\text{size}(\text{bcomp} b c n) + (\text{if} c = \text{bval} b s \text{ then } n \text{ else } 0),s,\text{stk})\)

**proof(induct b arbitrary: c n m)**
- **case Not**
  - from Not[of ~ c] show \(?case by fastsimp**
- **next**
  - **case (And b1 b2)**
  - from And(1)[of False] And(2)[of c] show \(?case by fastsimp**
- **qed fastsimp+**

**fun ccomp :: com ⇒ instr list where**
- \(\text{ccomp} \text{ SKIP } = []\)
- \(\text{ccomp} (x ::= a) = \text{acomp} a \@ [\text{STORE} x]\)
- \(\text{ccomp} (c_1;c_2) = \text{ccomp} c_1 \@ \text{ccomp} c_2\)
- \(\text{ccomp} (\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2) =\)
  - (let \(cc_1 = \text{ccomp } c_1; cc_2 = \text{ccomp } c_2; cb = \text{bcomp } b \text{ False } (\text{size } cc_1 + 1)\)
in cb @ cc1 @ JMPF(size cc2) ≠ cc2) |
ccomp (WHILE b DO c) =
(let cc = ccomp c; cb = bcomp b False (size cc + 1)
in cb @ cc @ [JMPB (size cb + size cc + 1)])

value ccomp (IF Less (V 4) (N 1) THEN 4 ::= Plus (V 4) (N 1) ELSE 3 ::= V 4)

value ccomp (WHILE Less (V 4) (N 1) DO (4 ::= Plus (V 4) (N t)))

5.4 Preservation of semantics

lemma ccomp-correct:
(c,s) ⇒ t ⇒ ccomp c ⊢ (0,s,stk) →* (size(ccomp c),t,stk)

proof(induct arbitrary; stk rule: big-step-induct)

case (Assign x a s)

show ?case by (fatsimp simp:fun-upd-def)

next

case (Semi c1 s1 s2 c2 s3)

let ?cc1 = ccomp c1 let ?cc2 = ccomp c2

have ?cc1 @ ?cc2 ⊢ (0,s1,stk) →* (size ?cc1,s2,stk)

using Semi.hyps(2) by (fatsimp)

moreover

have ?cc1 @ ?cc2 ⊢ (size ?cc1,s2,stk) →* (size ?cc1 ?cc2),s3,stk)

using Semi.hyps(4) by (fatsimp)

ultimately show ?case by simp (blast intro: exec-trans)

next

case (WhileTrue b s1 c s2 s3)

let ?cc = ccomp c

let ?cb = bcomp b False (size ?cc + 1)

let ?cw = ccomp(WHILE b DO c)

have ?cw ⊢ (0,s1,stk) →* (size ?cb + size ?cc,s2,stk)

using WhileTrue(1,3) by fatsimp

moreover

have ?cw ⊢ (size ?cb + size ?cc,s2,stk) →* (0,s2,stk)

by (fatsimp)

moreover

have ?cw ⊢ (0,s2,stk) →* (size ?cw,s3,stk) by(rule WhileTrue(5))

ultimately show ?case by (blast intro: exec-trans)

qed fatsimp+

end
6 A Typed Language

theory Types imports Complex-Main begin

6.1 Arithmetic Expressions

datatype val = Iv int | Rv real

types
  name = nat
  state = name ⇒ val

datatype aexp = Ic int | Rc real | V name | Plus aexp aexp

inductive taval :: aexp ⇒ state ⇒ val ⇒ bool where
  taval (Ic i) s (Iv i) |
  taval (Rc r) s (Rv r) |
  taval (V x) s (s x) |
  taval a1 s (Iv i1) ⇒ taval a2 s (Iv i2) ⇒
  taval (Plus a1 a2) s (Iv(i1+i2)) |
  taval a1 s (Rv r1) ⇒ taval a2 s (Rv r2) ⇒
  taval (Plus a1 a2) s (Rv(r1+r2)) |

inductive-cases [elim!]:
  taval (Ic i) s v  taval (Rc i) s v
  taval (V x) s v
  taval (Plus a1 a2) s v

6.2 Boolean Expressions

datatype bexp = B bool | Not bexp | And bexp bexp | Less aexp aexp

inductive tbval :: bexp ⇒ state ⇒ bool ⇒ bool where
  tbval (B bv) s bv |
  tbval b s bv ⇒ tbval (Not b) s (¬ bv) |
  tbval b1 s bv1 ⇒ tbval b2 s bv2 ⇒ tbval (And b1 b2) s (bv1 & bv2) |
  taval a1 s (Iv i1) ⇒ taval a2 s (Iv i2) ⇒
  taval (Less a1 a2) s (i1 < i2) |
  taval a1 s (Rv r1) ⇒ taval a2 s (Rv r2) ⇒
  taval (Less a1 a2) s (r1 < r2)

6.3 Syntax of Commands

datatype com = SKIP
  | Assign name aexp (· ::= - [1000, 61] 61)
6.4 Small-Step Semantics of Commands

\textbf{inductive}
\[
\text{small-step} :: (\text{com} \times \text{state}) \Rightarrow (\text{com} \times \text{state}) \Rightarrow \text{bool} \quad \text{(infix } \rightarrow \text{ 55)}
\]
\textbf{where}
\begin{align*}
\text{Assign: } & \quad \text{taeval } a \ s \ v \implies (x : = a, s) \rightarrow (\text{SKIP, } s(x := v)) \\
\text{Semi1: } & \quad (\text{SKIP} ; c, s) \rightarrow (c, s) \\
\text{Semi2: } & \quad (c_1, s) \rightarrow (c_1', s') \implies (c_1 ; c_2, s) \rightarrow (c_1', c_2, s')
\end{align*}
\begin{align*}
\text{IfTrue: } & \quad \text{tbval } b \ s \ \text{True} \implies (\text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2, s) \rightarrow (c_1, s) \\
\text{IfFalse: } & \quad \text{tbval } b \ s \ \text{False} \implies (\text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2, s) \rightarrow (c_2, s)
\end{align*}
\begin{align*}
\text{While: } & \quad (\text{WHILE } b \ \text{DO } c, s) \rightarrow (\text{IF } b \ \text{THEN } c ; \ \text{WHILE } b \ \text{DO } c \ \text{ELSE } \text{SKIP}, s)
\end{align*}
\textbf{lemmas} small-step-induct = small-step.induct[split-format(complete)]

6.5 The Type System

\textbf{datatype} \( ty = \text{Ity} \mid \text{Rty} \)

\textbf{types} \( \text{tyenv} = \text{name} \Rightarrow ty \)

\textbf{inductive} \( \text{atyping} :: \text{tyenv} \Rightarrow \text{aexp} \Rightarrow ty \Rightarrow \text{bool} \quad ((1/- \vdash / (\vdash : / \vdash)) \ 50,0,50 \ 50) \)
\textbf{where}
\begin{align*}
\text{Ic-ty: } & \quad \Gamma \vdash \text{Ic } i : \text{Ity} \\
\text{Rc-ty: } & \quad \Gamma \vdash \text{Rc } r : \text{Rty} \\
\text{V-ty: } & \quad \Gamma \vdash V x : \text{Rty} \\
\text{Plus-ty: } & \quad \Gamma \vdash a_1 : \tau \implies \Gamma \vdash a_2 : \tau \implies \Gamma \vdash \text{Plus } a_1 a_2 : \tau
\end{align*}

Warning: the “\(:\)” notation leads to syntactic ambiguities, i.e. multiple parse trees, because “\(:\)” also stands for set membership. In most situations Isabelle’s type system will reject all but one parse tree, but will still inform you of the potential ambiguity.

\textbf{inductive} \( \text{btyping} :: \text{tyenv} \Rightarrow \text{bexp} \Rightarrow \text{bool} \quad \text{(infix } \vdash 50) \)
\textbf{where}
\begin{align*}
\text{B-ty: } & \quad \Gamma \vdash \text{B } bv \\
\text{Not-ty: } & \quad \Gamma \vdash b \implies \Gamma \vdash \text{Not } b \\
\text{And-ty: } & \quad \Gamma \vdash b_1 \implies \Gamma \vdash b_2 \implies \Gamma \vdash \text{And } b_1 b_2
\end{align*}
Less-ty: $\Gamma \vdash a_1 : \tau \implies \Gamma \vdash a_2 : \tau \implies \Gamma \vdash \text{Less } a_1 a_2$

**inductive ctyping :: tyenv \Rightarrow \text{com} \Rightarrow \text{bool} (\text{infix } \vdash 50) \text{ where}**

Skip-ty: $\Gamma \vdash \text{SKIP} \mid$
Assign-ty: $\Gamma \vdash a : \Gamma(x) \implies \Gamma \vdash x ::= a \mid$
Semi-ty: $\Gamma \vdash c_1 \implies \Gamma \vdash c_2 \implies \Gamma \vdash c_1; c_2 \mid$
If-ty: $\Gamma \vdash b \implies \Gamma \vdash c_1 \implies \Gamma \vdash IF b \text{ THEN } c_1 \text{ ELSE } c_2 \mid$
While-ty: $\Gamma \vdash b \implies \Gamma \vdash c \implies \Gamma \vdash \text{WHILE } b \text{ DO } c$

**inductive-cases [elim!]:**

- $\Gamma \vdash x ::= a \Gamma \vdash c_1; c_2$
- $\Gamma \vdash IF b \text{ THEN } c_1 \text{ ELSE } c_2$
- $\Gamma \vdash \text{WHILE } b \text{ DO } c$

6.6 Well-typed Programs Do Not Get Stuck

fun type :: val \Rightarrow ty where
type (Iv i) = Ity \mid
type (Rv r) = Rty

**lemma [simp]:** type $v = \text{Ity} \leftrightarrow (\exists i. v = \text{Iv } i)$
by (cases $v$) simp-all

**lemma [simp]:** type $v = \text{Rty} \leftrightarrow (\exists r. v = \text{Rv } r)$
by (cases $v$) simp-all

definition styping :: tyenv \Rightarrow \text{state} \Rightarrow \text{bool} (\text{infix } \vdash 50) \text{ where} \Gamma \vdash s \leftrightarrow (\forall x. \text{type } (s x) = \Gamma x)$

**lemma apreservation:**

- $\Gamma \vdash a : \tau \implies \text{taval } a s v \implies \Gamma \vdash s \implies \text{type } v = \tau$

**apply** (induct arbitrary: $v$ rule: atyping.induct)

**apply** (fastsimp simp: styping-def)+
done

**lemma aprogress:** $\Gamma \vdash a : \tau \implies \Gamma \vdash s \implies \exists v. \text{taval } a s v$
**proof** (induct rule: atyping.induct)

- **case** (Plus-ty $\Gamma a_1 t a_2$)
- **then obtain** $v_1 v_2$ **where** $v$: taval $a_1 s v_1$ taval $a_2 s v_2$ **by** blast
- **show** ?case
  **proof** (cases $v_1$)
    **case** Iv
    **with** Plus-ty$(1,3,5)$ **v show** ?thesis
      **by** (fastsimp intro: taval.intros(4) dest!: apreservation)
next
case Rv
  with Plus-ty(1,3,5) v show ?thesis
    by (fastsimp intro: taval.intros(5) dest!: apreservation)
qed

lemma bprogress: Γ ⊢ b → Γ ⊢ s → ∃v. tbval b s v
proof (induct rule: btyping.induct)
case (Less-ty Γ a1 t a2)
  then obtain v1 v2 where v: taval a1 s v1 taval a2 s v2
    by (metis aprogress)
  show ?case
  proof (cases v1)
    case Iv
      with Less-ty v show ?thesis
        by (fastsimp intro!: taval.intros(4) dest!:apreservation)
  next
    case Rv
      with Less-ty v show ?thesis
        by (fastsimp intro!: taval.intros(5) dest!:apreservation)
  qed
qed (auto intro: taval.intros)

theorem progress:
  Γ ⊢ c → Γ ⊢ s → c ≠ SKIP → ∃cs'. (c,s) → cs'
proof (induct rule: ctyping.induct)
case Skip-ty thus ?case by simp
next
case Assign-ty
  thus ?case by (metis Assign aprogress)
next
case Semi-ty thus ?case by simp (metis Semi1 Semi2)
next
case (If-ty Γ b c1 c2)
  then obtain bv where tbval b s bv by (metis bprogress)
  show ?case
  proof (cases bv)
    assume bv
      with (tbval b s bv) show ?case by simp (metis IfTrue)
  next
    assume ¬bv
      with (tbval b s bv) show ?case by simp (metis IfFalse)
  qed

28
next
  case While-ty show ?case by (metis While)
qed

theorem styping-preservation:
  \((c, s) \rightarrow (c', s') \implies \Gamma \vdash c \implies \Gamma \vdash s'\)

proof (induct rule: small-step-induct)
  case Assign
  thus ?case by (auto simp: styping-def) (metis Assign 1,3 apreservation)
qed auto

theorem ctyping-preservation:
  \((c, s) \rightarrow (c', s') \implies \Gamma \vdash c' \implies \Gamma \vdash s\)
by (induct rule: small-step-induct) (auto simp: ctyping.intros)

inductive
  small-steps :: \(\text{com} \ast \text{state} \Rightarrow \text{com} \ast \text{state} \Rightarrow \text{bool} \) (infix \(\rightarrow\ast\) 55) where
  refl: \(cs \rightarrow\ast cs\)
  step: \(cs \rightarrow cs' \Rightarrow cs' \rightarrow\ast cs'' \Rightarrow cs \rightarrow\ast cs''\)

lemmas small-steps-induct = small-steps.induct[split-format(complete)]

theorem type-sound:
  \((c, s) \rightarrow\ast (c', s') \implies \Gamma \vdash c \implies \Gamma \vdash s \implies c' \neq \text{SKIP}\)
  \implies \exists cs''. (c', s') \rightarrow cs''
apply (induct rule: small-steps-induct)
apply (metis progress)
by (metis styping-preservation ctyping-preservation)

end

theory Poly-Types imports Types begin

6.7 Type Variables

datatype \(ty = \text{Ity} | \text{Rty} | \text{TV} \text{nat}\)
    
    Everything else remains the same.

types \(tyenv = \text{name} \Rightarrow ty\)

inductive atyping :: tyenv \Rightarrow aexp \Rightarrow ty \Rightarrow bool
  ((1-/ \vdash p / (- : -)) \ [50,0,50] 50)
where
  \(\Gamma \vdash p \text{ Ic i : Ity}\)
\(\Gamma \vdash p \text{ Rc } r : \text{ Rty} | \Gamma \vdash p \text{ V x} : \Gamma x | \Gamma \vdash p a_1 : \tau \implies \Gamma \vdash p a_2 : \tau \implies \Gamma \vdash p \text{ Plus } a_1 a_2 : \tau\)

**inductive** *btyping* :: *tyenv* \(\Rightarrow\) *bexp* \(\Rightarrow\) *bool* (infix \(\vdash\) \(p\) \(50\))

**where**

\(\Gamma \vdash p \text{ B bv} |\)
\(\Gamma \vdash p b \implies \Gamma \vdash p \text{ Not } b |\)
\(\Gamma \vdash p b_1 \implies \Gamma \vdash p b_2 \implies \Gamma \vdash p \text{ And } b_1 b_2 |\)
\(\Gamma \vdash p a_1 : \tau \implies \Gamma \vdash p a_2 : \tau \implies \Gamma \vdash p \text{ Less } a_1 a_2\)

**inductive** *ctyping* :: *tyenv* \(\Rightarrow\) *com* \(\Rightarrow\) *bool* (infix \(\vdash\) \(p\) \(50\))

**where**

\(\Gamma \vdash p \text{ SKIP} |\)
\(\Gamma \vdash p a : \Gamma(x) \implies \Gamma \vdash p x ::= a |\)
\(\Gamma \vdash p c_1 \implies \Gamma \vdash p c_2 \implies \Gamma \vdash p c_1;c_2 |\)
\(\Gamma \vdash p b \implies \Gamma \vdash p c_1 \implies \Gamma \vdash p c_2 \implies \Gamma \vdash p \text{ IF } b \text{ THEN } c_1 \text{ ELSE } c_2 |\)
\(\Gamma \vdash p b \implies \Gamma \vdash p c \implies \Gamma \vdash p \text{ WHILE } b \text{ DO } c\)

```
fun type :: *val* \(\Rightarrow\) *ty* where
type \((Iv i)\) = \(Ity\)
type \((Rv r)\) = \(Rty\)
```

**definition** *styping* :: *tyenv* \(\Rightarrow\) *state* \(\Rightarrow\) *bool* (infix \(\vdash\) \(p\) \(50\))

**where** \(\Gamma \vdash p s \iff (\forall x. \text{ type } (s x) = \Gamma x)\)

```
fun tsubst :: \((nat \Rightarrow ty) \Rightarrow ty \Rightarrow ty\) where
tsubst S \((TV n)\) = \(S n\)
tsubst S \(t\) = \(t\)
```

### 6.8 Typing is Preserved by Substitution

**lemma** subst-atyping: \(E \vdash p a : t \implies \text{ tsubst } S \circ E \vdash p a : \text{ tsubst } S t\)

**apply**(induct rule: atyping.induct)

**apply**(auto intro: atyping.intros)

**done**

**lemma** subst-btyping: \(E \vdash p \text{ (b::bexp)} \implies \text{ tsubst } S \circ E \vdash p b\)

**apply**(induct rule: btyping.induct)

**apply**(auto intro: btyping.intros)

**apply**(drule subst-atyping[where \(S=S\)])

**apply**(drule subst-atyping[where \(S=S\)])

**apply**(simp add: o-def btyping.intros)

**done**
lemma subst-ctyping: \( E \vdash p \ (c::\mathsf{com}) \implies \mathit{tsubst} \circ E \vdash p \ c \)
apply (induct rule: ctyping.induct)
apply (auto intro: ctyping.intros)
apply (drule subst-atyping [where \( S=S \)])
apply (simp add: o-def ctyping.intros)
apply (drule subst-btyping [where \( S=S \)])
apply (simp add: o-def ctyping.intros)
done

end

7  Definite Assignment Analysis

theory Vars imports Util BExp
begin

7.1 The Variables in an Expression

We need to collect the variables in both arithmetic and boolean expressions.
For a change we do not introduce two functions, e.g. \( \mathit{avars} \) and \( \mathit{bvars} \), but
we overload the name \( \mathit{vars} \) via a \textit{type class}, a device that originated with
Haskell:

\begin{verbatim}
class vars =
fixes vars :: 'a \Rightarrow \mathit{name set}
\end{verbatim}

This defines a type class "vars" with a single function of (coincidentally)
the same name. Then we define two separated instances of the class, one
for \( \mathit{aexp} \) and one for \( \mathit{bexp} \):

\begin{verbatim}
instantiation aexp :: vars
begin

fun vars-aexp :: aexp \Rightarrow \mathit{name set} where
vars-aexp (N n) = \{} |
vars-aexp (V x) = \{x\} |
vars-aexp (Plus a1 a2) = vars-aexp a1 \cup vars-aexp a2

instance ..
end
\end{verbatim}
value vars(Plus (V 3) (V 2))

We need to convert functions to lists before we can view them:

value list (vars(Plus (V 3) (V 2))) 4

instantiation bexp :: vars
begin

fun vars-bexp :: bexp ⇒ name set where
  vars-bexp (B bv) = {} |
  vars-bexp (Not b) = vars-bexp b |
  vars-bexp (And b1 b2) = vars-bexp b1 ∪ vars-bexp b2 |
  vars-bexp (Less a1 a2) = vars a1 ∪ vars a2

instance ..

end

value list (vars(Less (Plus (V 3) (V 2)) (V 1))) 5

abbreviation
eq-on :: ('a ⇒ 'b) ⇒ ('a ⇒ 'b) ⇒ 'a set ⇒ bool
((= = / - / on - ) [50,0,50] 50) where
  f = g on X == ∀ x ∈ X. f x = g x

lemma aval-eq-if-eq-on-vars[simp]:
  s1 = s2 on vars a ⇒ aval a s1 = aval a s2
  apply(induct a)
  apply simp-all
  done

lemma bval-eq-if-eq-on-vars:
  s1 = s2 on vars b ⇒ bval b s1 = bval b s2
  proof(induct b)
    case (Less a1 a2)
    hence aval a1 s1 = aval a1 s2 and aval a2 s1 = aval a2 s2 by simp-all
    thus ?case by simp
  qed simp-all

end

theory Def-Ass imports Vars Com
begin
7.2 Definite Assignment Analysis

\[ \text{inductive } D :: \text{name set } \Rightarrow \text{com } \Rightarrow \text{name set } \Rightarrow \text{bool where} \]

\begin{align*}
\text{Skip: } & D \ A \ \text{SKIP} \ A \\
\text{Assign: } & \text{vars } a \subseteq A \implies D \ (x := a) \ (\text{insert } x \ A) \\
\text{Semi: } & [D \ A_1 \ c_1 \ A_2; \ D \ A_2 \ c_2 \ A_3] \implies D \ A_1 \ (c_1; \ c_2) \ A_3 \\
\text{If: } & [\text{vars } b \subseteq A; \ D \ A_1 \ c_1; \ D \ A_2 \ c_2] \implies D \ (\text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2) \ (A_1 \ \text{Int} \ A_2) \\
\text{While: } & [\text{vars } b \subseteq A; \ D \ A \ c \ A'] \implies D \ (\text{WHILE } b \ \text{DO } c) \ A \\
\end{align*}

\[ \text{inductive-cases [elim]}: \]

\begin{align*}
D \ A \ \text{SKIP} \ A' \\
D \ A \ (x := a) \ A' \\
D \ A \ (c_1;c_2) \ A' \\
D \ A \ (\text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2) \ A' \\
D \ A \ (\text{WHILE } b \ \text{DO } c) \ A' \\
\end{align*}

\[ \text{lemma } D\text{-incr}: \]

\[ D \ A \ c \ A' \implies A \subseteq A' \]

\text{by (induct rule: } D\text{.induct) auto}

end

theory Def-Ass-Exp imports Vars begin

7.3 Initialization-Sensitive Expressions Evaluation

types

\[ \text{val } = \text{nat} \]
\[ \text{state } = \text{name } \Rightarrow \text{val option} \]

fun aval :: aexp \Rightarrow state \Rightarrow val option where

aval (N i) s = Some i \\
aval (V x) s = s x \\
aval (Plus a_1 a_2) s =
\begin{cases}
\text{case (aval } a_1 \ s, \text{ aval } a_2 \ s) \ \text{of} \\
\quad \text{Some } i_1, \text{Some } i_2 \ \Rightarrow \text{Some } (i_1+i_2) \\
\quad \text{-} \ \Rightarrow \text{None}
\end{cases}

fun bval :: bexp \Rightarrow state \Rightarrow bool option where
bval \((B \, bv)\) \(s = \text{Some } bv\) |

\[\text{bval } (\text{Not } b) \, s = \left(\text{case } \text{bval } b \, s \text{ of } \text{None } \Rightarrow \text{None } \mid \text{Some } bv \Rightarrow \text{Some}(\neg \, bv)\right)\]

\[\text{bval } (\text{And } b_1 \, b_2) \, s = \left(\text{case } (\text{bval } b_1 \, s, \text{bval } b_2 \, s) \text{ of } \left(\text{Some } bv_1, \text{Some } bv_2\right) \Rightarrow \text{Some}(bv_1 \, \& \, bv_2) \mid - \Rightarrow \text{None}\right)\]

\[\text{bval } (\text{Less } a_1 \, a_2) \, s = \left(\text{case } (\text{aval } a_1 \, s, \text{aval } a_2 \, s) \text{ of } \left(\text{Some } i_1, \text{Some } i_2\right) \Rightarrow \text{Some}(i_1 < i_2) \mid - \Rightarrow \text{None}\right)\]

**Lemma aval-Some**

\[\text{vars } a \subseteq \text{dom } s \Rightarrow \exists \, i. \text{aval } a \, s = \text{Some } i\]

*by (induct \(a\)) auto*

**Lemma bval-Some**

\[\text{vars } b \subseteq \text{dom } s \Rightarrow \exists \, \text{bv}. \text{bval } b \, s = \text{Some } \text{bv}\]

*by (induct \(b\)) (auto dest: aval-Some)*

end

theory Def-Ass-Big imports Com Def-Ass-Exp
begin

7.4 Initialization-Sensitive Big Step Semantics

inductive

\[\text{big-step} :: \left(\text{com} \times \text{state option}\right) \Rightarrow \text{state option} \Rightarrow \text{bool} \text{ (infix } \Rightarrow \text{ 55)}\]

where

\[\text{None: } (c, \text{None}) \Rightarrow \text{None}\]

\[\text{Skip: } (\text{SKIP}, s) \Rightarrow s\]

\[\text{AssignNone: } \text{aval } a \, s = \text{None} \Rightarrow (x ::= a, \text{Some } s) \Rightarrow \text{None}\]

\[\text{Assign: } \text{aval } a \, s = \text{Some } i \Rightarrow (x ::= a, \text{Some } s) \Rightarrow \text{Some}(s(x := \text{Some } i))\]

\[\text{Semi: } (c_1, s_1) \Rightarrow s_2 \Rightarrow (c_2, s_2) \Rightarrow s_3 \Rightarrow (c_1; c_2, s_1) \Rightarrow s_3\]

\[\text{IfNone: } \text{bval } b \, s = \text{None} \Rightarrow (\text{IF } b \, \text{THEN } c_1 \, \text{ELSE } c_2, \text{Some } s) \Rightarrow \text{None}\]

\[\text{IfTrue: } [ \text{bval } b \, s = \text{Some } \text{True} ; (c_1, \text{Some } s) \Rightarrow s' ] \Rightarrow (\text{IF } b \, \text{THEN } c_1 \, \text{ELSE } c_2, \text{Some } s) \Rightarrow s'\]

\[\text{IfFalse: } [ \text{bval } b \, s = \text{Some } \text{False} ; (c_2, \text{Some } s) \Rightarrow s' ] \Rightarrow (\text{IF } b \, \text{THEN } c_1 \, \text{ELSE } c_2, \text{Some } s) \Rightarrow s'\]

\[\text{WhileNone: } \text{bval } b \, s = \text{None} \Rightarrow (\text{WHILE } b \, \text{DO } c, \text{Some } s) \Rightarrow \text{None}\]

\[\text{WhileFalse: } \text{bval } b \, s = \text{Some } \text{False} \Rightarrow (\text{WHILE } b \, \text{DO } c, \text{Some } s) \Rightarrow \text{Some } s\]

\[\text{WhileTrue:}\]

34
theory Def-Ass-Sound-Big imports Def-Ass Def-Ass-Big begin

7.5 Soundness wrt Big Steps

Note the special form of the induction because one of the arguments of the
inductive predicate is not a variable but the term Some s:

theorem Sound:
\[
\begin{align*}
\llbracket \text{bval } b \text{ s} = \text{Some True}; (c, \text{Some s}) & \Rightarrow s' ; (\text{WHILE b DO c, s'}) & \Rightarrow s'' \rrbracket \\
\Rightarrow (\text{WHILE b DO c, Some s}) & \Rightarrow s''
\end{align*}
\]

proof (induct c Some s s' arbitrary: s A A' rule: big-step-induct)
  case AssignNone thus ?case by auto
  case Semi thus ?case by auto metis
  case IfTrue thus ?case by auto blast
  case IfFalse thus ?case by auto blast
  case IfNone thus ?case
    by auto (metis bval-Some option.simps(3) order-trans)
  case WhileNone thus ?case
    by auto (metis bval-Some option.simps(3) order-trans)
  case (WhileTrue b s c s' s'')
    from \(D A (\text{WHILE b DO c}) A') \text{ obtain } A' \text{ where } D A c A' \text{ by blast}
    then obtain t' where s' = Some t' A \subseteq dom t'
      by (metis D-incr WhileTrue(3,7) subset-trans)
    from WhileTrue(5)[OF this(1) WhileTrue(6) this(2)] show ?case .
  qed auto

corollary sound: \[ \llbracket D (\text{dom s}) \text{ c A'}; (c, \text{Some s}) & \Rightarrow s' \rrbracket \Rightarrow s' \neq \text{None} \]
by (metis Sound not-Some-eq subset-refl)

end

8 Live Variable Analysis

theory Live imports Vars Big-Step
begin

8.1 Liveness Analysis

fun L :: com ⇒ name set ⇒ name set where
L SKIP X = X |
L (x ::= a) X = X − {x} ∪ vars a |
L (c1; c2) X = (L c1 o L c2) X |
L (IF b THEN c1 ELSE c2) X = vars b ∪ L c1 X ∪ L c2 X |
L (WHILE b DO c) X = vars b ∪ X ∪ L c X

value list (L (1 ::= V 2; 0 ::= Plus (V 1) (V 2)) {0}) 3
value list (L (WHILE Less (V 0) (V 0) DO 1 ::= V 2) {0}) 3

fun kill :: com ⇒ name set where
kill SKIP = {} |
kill (x ::= a) = {x} |
kill (c1; c2) = kill c1 ∪ kill c2 |
kill (IF b THEN c1 ELSE c2) = kill c1 ∩ kill c2 |
kill (WHILE b DO c) = {}

fun gen :: com ⇒ name set where
gen SKIP = {} |
gen (x ::= a) = vars a |
gen (c1; c2) = gen c1 ∪ (gen c2 − kill c1) |
gen (IF b THEN c1 ELSE c2) = vars b ∪ gen c1 ∪ gen c2 |
gen (WHILE b DO c) = vars b ∪ gen c

lemma L-gen-kill: L c X = (X − kill c) ∪ gen c
by(induct c arbitrary:X) auto

lemma L-While-subset: L c (L (WHILE b DO c) X) ⊆ L (WHILE b DO c) X
by(auto simp add:L-gen-kill)
8.2 Soundness

**theorem** L-sound:

\[(c, s) \Rightarrow s' \Rightarrow s = t \text{ on } L \ c \ X \ \Rightarrow \ \exists \ t'. (c, t) \Rightarrow t' \& s' = t' \text{ on } X\]

**proof** (induct arbitrary: X t rule: big-step-induct)

- **case** Skip then show ?case by auto

- **next**
  - **case** Assign then show ?case by (auto simp: ball-Un)

- **next**
  - **case** (Semi c1 s1 s2 s3 X t1)
    - from Semi(2,5) obtain t2 where
      - t12: \((c1, t1) \Rightarrow t2 \text{ and } s2t2: s2 = t2 \text{ on } L \ c2 \ X\)
      - by simp blast
    - from Semi(4)[OF s2t2] obtain t3 where
      - t23: \((c2, t2) \Rightarrow t3 \text{ and } s3t3: s3 = t3 \text{ on } X\)
      - by auto
    - show ?case using t12 t23 s3t3 by auto

- **next**
  - **case** (IfTrue b s c1 s' c2)
    - hence \(s = t \text{ on } \text{vars } b \ s = t \text{ on } L \ c1 \ X\) by auto
    - from bval-eq-if-eq-on-vars[of this(1)] IfTrue(1) have bval b t by simp
    - from IfTrue(3)[OF s = t on L c1 X] obtain t' where
      - \((c1, t) \Rightarrow t' s' = t' \text{ on } X\) by auto
    - thus ?case using ⟨bval b t⟩ by auto

- **next**
  - **case** (IfFalse b s c2 s' c1)
    - hence \(\sim bval b t \text{ by simp: ball-Un}\) (metis bval-eq-if-eq-on-vars)
    - from IfFalse(3)[OF s = t on L c2 X] obtain t' where
      - \((c2, t) \Rightarrow t' s' = t' \text{ on } X\) by auto
    - thus ?case using ⟨bval b t⟩ by auto

- **next**
  - **case** (WhileFalse b s c)
    - hence \(\sim bval b t \text{ by simp: ball-Un}\) (metis bval-eq-if-eq-on-vars)
    - thus ?case using WhileFalse(2) by auto

- **next**
  - **case** (WhileTrue b s1 c s2 s3 X t1)
    - let \(?w = \text{WHILE } b \text{ DO } c\)
    - from bval b s1) WhileTrue(6) have bval b t1 by (auto simp: ball-Un) (metis bval-eq-if-eq-on-vars)
    - have s1 = t1 on L c (L ?w X) using L-While-subset WhileTrue.prems
      - by (blast)
from WhileTrue(3)[OF this] obtain t2 where
  (c, t1) ⇒ t2 s2 = t2 on L ?w X by auto
from WhileTrue(5)[OF this(2)] obtain t3 where (?w,t2) ⇒ t3 s3 = t3 on X
  by auto
with ⟨bval b t1⟩ ⟨(c, t1) ⇒ t2⟩ show ?case by auto
qed

8.3 Program Optimization
Burying assignments to dead variables:

fun bury :: com ⇒ name set ⇒ com where
  bury SKIP X = SKIP |
  bury (x ::= a) X = (if x:X then x:= a else SKIP) |
  bury (c1; c2) X = (bury c1 (L c2 X); bury c2 X) |
  bury (IF b THEN c1 ELSE c2) X = IF b THEN bury c1 X ELSE bury c2 X |
  bury (WHILE b DO c) X = WHILE b DO bury c (vars b ∪ X ∪ L c X)

We could prove the analogous lemma to L-sound, and the proof would be very similar. However, we phrase it as a semantics preservation property:

theorem bury-sound:
  (c,s) ⇒ s' === s = t on L c X ===
  ∃ t'. (bury c X,t) ⇒ t' & s' = t' on X
proof (induct arbitrary: X t rule: big-step-induct)
  case Skip then show ?case by auto
  next
case Assign then show ?case
    by (auto simp: ball-Un)
  next
case (Semi c1 s1 s2 c2 s3 X t1)
  from Semi(2,5) obtain t2 where
    t12: (bury c1 (L c2 X), t1) ⇒ t2 and s2t2: s2 = t2 on L c2 X
    by simp blast
  from Semi(4)[OF s2t2] obtain t3 where
    t23: (bury c2 X, t2) ⇒ t3 and s3t3: s3 = t3 on X
    by auto
  show ?case using t12 t23 s3t3 by auto
  next
case (IfTrue b s c1 s' c2)
  hence s = t on vars b s = t on L c1 X by auto
  from bval-eq-if-eq-on-vars[OF this(1)] IfTrue(1) have bval b t by simp
  from IfTrue(3)[OF s = t on L c1 X] obtain t' where
    (bury c1 X, t) ⇒ t' s' =t' on X by auto
  next
thus \(?\text{case using}(\sim \text{bval } b\ t)\) by auto

next

\begin{itemize}
  \item case \((\text{IfFalse } b\ s\ c2\ s'\ c1)\)
    \begin{itemize}
      \item \textbf{hence} \(s = t\) on \text{vars} \(b\ s = t\) on \(L\ c2\ X\) by auto
    \end{itemize}
  \item from \text{IfFalse}(1) have \(\sim \text{bval } b\ t\) by simp
  \item from \text{IfFalse}(3) obtain \(t'\) where
    \((\text{bury } c2\ X, t) \Rightarrow t'\ s' = t'\) on \(X\) by auto
  \item thus \(?\text{case using}(\sim \text{bval } b\ t)\) by auto
\end{itemize}

next

\begin{itemize}
  \item case \((\text{WhileFalse } b\ s\ c)\)
    \begin{itemize}
      \item \textbf{hence} \(\sim \text{bval } b\ t\) by \((\text{auto simp: ball-Un})\) (metis \text{bval-eq-if-eq-on-vars})
    \end{itemize}
  \item thus \(?\text{case using} \text{WhileFalse}(2)\) by auto
\end{itemize}

next

\begin{itemize}
  \item case \((\text{WhileTrue } b\ s1\ c\ s2\ s3\ X\ t1)\)
    \begin{itemize}
      \item let \(?w = \text{WHILE } b\ DO\ c\)
      \item from \text{bval}(b\ s1) \text{WhileTrue}(6) have \(\text{bval } b\ t1\)
        \begin{itemize}
          \item by \((\text{auto simp: ball-Un})\) (metis \text{bval-eq-if-eq-on-vars})
        \end{itemize}
      \item have \(s1 = t1\) on \(L\ c\ (L\ ?w\ X)\)
        \begin{itemize}
          \item using \text{L-While-subset} \text{WhileTrue.prem} by blast
        \end{itemize}
      \item from \text{WhileTrue}(3) \(\text{(OF this)}\) obtain \(t2\) where
        \((\text{bury } c\ (L\ ?w\ X), t1) \Rightarrow t2\ s2 = t2\) on \(L\ ?w\ X\) by auto
      \item from \text{WhileTrue}(5) \(\text{(OF this}(2)\) obtain \(t3\)
        \begin{itemize}
          \item where \((\text{bury } ?w\ X, t2) \Rightarrow t3\ s3 = t3\) on \(X\)
            \begin{itemize}
              \item by auto
            \end{itemize}
        \end{itemize}
      \item with \((\text{bval } b\ t1)\) \((\text{bval } c\ (L\ ?w\ X), t1) \Rightarrow t2)\) show \(?\text{case by auto}\)
    \end{itemize}
\end{itemize}

qed

\textbf{corollary} \text{final-bury-sound}: \((c,s) \Rightarrow s' \iff (\text{bury } c\ \text{UNIV}, s) \Rightarrow s'\)

\textbf{using} \text{bury-sound[of \(c\ s\ s'\ \text{UNIV}]\)

\textbf{by} \((\text{auto simp: expand-fun-eq}[\text{symmetric}]\))

Now the opposite direction.

\textbf{lemma} \text{SKIP-bury}[simp]:
\(\text{SKIP} = \text{bury } c\ X \iff c = \text{SKIP} | (\text{EX } x\ a.\ c = x::=a \land x \notin X)\)
\textbf{by} \((\text{cases } c)\) \text{auto}

\textbf{lemma} \text{Assign-bury}[simp]: \(x::=a = \text{bury } c\ X \iff c = x::=a \land x : X\)
\textbf{by} \((\text{cases } c)\) \text{auto}

\textbf{lemma} \text{Semi-bury}[simp]: \(bc1;bc2 = \text{bury } c\ X \iff (\text{EX } c1\ c2.\ c = c1;c2 \land bc2 = \text{bury } c2\ X \land bc1 = \text{bury } c1\ (L\ c2\ X))\)
\textbf{by} \((\text{cases } c)\) \text{auto}

39
lemma If-bury[simp]: IF b THEN bc1 ELSE bc2 = bury c X ⟷
  (EX c1 c2. c = IF b THEN c1 ELSE c2 &
   bc1 = bury c1 X & bc2 = bury c2 X)
by (cases c) auto

lemma While-bury[simp]: WHILE b DO bc' = bury c X ⟷
  (EX c'. c = WHILE b DO c' & bc' = bury c' (vars b ∪ X ∪ L c X))
by (cases c) auto

theorem bury-sound2:
  (bury c X, s) ⇒ s' =⇒ s =⇒ t on L c X =⇒ ∃ t'. (c, t) ⇒ t' & s' = t' on X
proof (induct bury c X s s' arbitrary: c X t rule: big-step-induct)
next
  case Skip then show ?case by auto
next
  case Assign then show ?case by (auto simp: ball-Un)
next
  case (Semi bc1 s1 s2 bc2 s3 c X t1)
  then obtain c1 c2 where c: c = c1;c2
  and bc2: bc2 = bury c2 X and bc1: bc1 = bury c1 (L c2 X) by auto
  from Semi(2)[OF bc1, of t1] Semi.prems c obtain t2 where
  t12: (c1, t1) ⇒ t2 and s2t2: s2 = t2 on L c2 X by auto
  from Semi(4)[OF bc2 s2t2] obtain t3 where
  t23: (c2, t2) ⇒ t3 and s3t3: s3 = t3 on X by auto
  show ?case using c t12 t23 s3t3 by auto
next
  case (IfTrue b s bc1 s' bc2)
  then obtain c1 c2 where c: c = IF b THEN c1 ELSE c2
  and bc1: bc1 = bury c1 X and bc2: bc2 = bury c2 X by auto
  have s = t on vars b s = t on L c1 X using IfTrue.prems c by auto
  from bval-eq-if-eq-on-vars[OF this(1)] IfTrue(1) have bval b t by simp
  from IfTrue(3)[OF bc1 (s = t on L c1 X)] obtain t' where
  (c1, t) ⇒ t' s' = t' on X by auto
  thus ?case using c ⟨bval b t⟩ by auto
next
  case (IfFalse b s bc2 s' bc1)
  then obtain c1 c2 where c: c = IF b THEN c1 ELSE c2
  and bc1: bc1 = bury c1 X and bc2: bc2 = bury c2 X by auto
  have s = t on vars b s = t on L c2 X using IfFalse.prems c by auto
  from bval-eq-if-eq-on-vars[OF this(1)] IfFalse(1) have ~bval b t by simp
  from IfFalse(3)[OF bc2 (s = t on L c2 X)] obtain t' where
  (c2, t) ⇒ t' s' = t' on X by auto

40
thus \textbf{case using} \textit{c \langle \sim bval b t \rangle} \textbf{by auto}

\textbf{next}

\textbf{case (WhileFalse b s c)}

\textbf{hence} \textit{\sim bval b t} \textbf{by} (\textit{auto simp: ball-Un dest: bval-eq-if-eq-on-vars})

\textbf{thus \textbf{case using} WhileFalse} \textbf{by auto}

\textbf{next}

\textbf{case (WhileTrue b s1 bc' s2 s3 c X t1)}

\textbf{then obtain} \textit{c'} \textbf{where} \textit{c} = \textit{WHILE b DO c'}

\textbf{and bc': bc'} = \textit{bury c'} (\textit{vars b} \cup \textit{X} \cup \textit{L c'} \textit{X}) \textbf{by auto}

\textbf{let ?w = While b DO c'}

\textbf{from } (\textit{bval b s1}) \textit{WhileTrue.prems c} \textbf{have \textbf{bval b t1}}

\textbf{by} (\textit{auto simp: ball-Un}) (\textit{metis bval-eq-if-eq-on-vars})

\textbf{have} \textit{s1} = \textit{t1} on \textit{L c'} (\textit{L ?w X})

\textbf{using} \textit{L-While-subset WhileTrue.prems c} \textbf{by blast}

\textbf{with WhileTrue(3)} [\textit{OF bc'} \textit{of t1}] \textbf{obtain} \textit{t2} \textbf{where}

\textit{(c', t1)} \Rightarrow \textit{t2 s2 = t2 on L ?w X} \textbf{by auto}

\textbf{from WhileTrue(5)} [\textit{OF WhileTrue(6)} \textit{of t2}] \textit{c this(2)} \textbf{obtain} \textit{t3}

\textbf{where} \textit{(?w,t2)} \Rightarrow \textit{t3 s3 = t3 on X}

\textbf{by auto}

\textbf{with} (\textit{bval b t1}) ((c', t1) \Rightarrow t2) \textbf{c show \textbf{case by auto}}

\textbf{qed}

\textbf{corollary} final-bury-sound2: (bury c UNIV, s) \Rightarrow \textit{s' \Rightarrow (c,s) \Rightarrow s'}

\textbf{using} bury-sound2 [\textit{of c UNIV}]

\textbf{by} (\textit{auto simp: expand-fun-eq[symmetric]})

\textbf{corollary} bury-iff: (bury c UNIV, s) \Rightarrow \textit{s' \leftrightarrow (c,s) \Rightarrow s'}

\textbf{by (metis final-bury-sound final-bury-sound2)}

\textbf{end}

9 Security Type Systems

\textbf{theory Sec-Type-Expr imports Big-Step begin}

9.1 Security Levels and Expressions

\textbf{types} \textit{level = nat}

The security/confidentiality level of each variable is globally fixed for simplicity. For the sake of examples — the general theory does not rely on it! — variable number \textit{n} has security level \textit{n}:

\newpage
class \textit{sec} = \textit{fixes} \textit{sec} :: 'a \Rightarrow \textit{level}

\textbf{instantiation} \textit{nat} :: \textit{sec}
begin

\textbf{definition} \textit{sec-nat} :: \textit{name} \Rightarrow \textit{level} \textbf{where} \textit{sec n} = n

\textbf{instance} ..
end

\textbf{instantiation} \textit{aexp} :: \textit{sec}
begin

\textbf{fun} \textit{sec-aexp} :: \textit{aexp} \Rightarrow \textit{level} \textbf{where}
\textit{sec-aexp (N n)} = 0 |
\textit{sec-aexp (V x)} = \textit{sec x} |
\textit{sec-aexp (Plus a1 a2)} = \text{max} (\textit{sec-aexp a1}) (\textit{sec-aexp a2})

\textbf{instance} ..
end

\textbf{instantiation} \textit{bexp} :: \textit{sec}
begin

\textbf{fun} \textit{sec-bexp} :: \textit{bexp} \Rightarrow \textit{level} \textbf{where}
\textit{sec-bexp (B bv)} = 0 |
\textit{sec-bexp (Not b)} = \textit{sec-bexp b} |
\textit{sec-bexp (And b1 b2)} = \text{max} (\textit{sec-bexp b1}) (\textit{sec-bexp b2}) |
\textit{sec-bexp (Less a1 a2)} = \text{max} (\textit{sec a1}) (\textit{sec a2})

\textbf{instance} ..
end

\textbf{abbreviation} \textit{eq-le} :: \textit{state} \Rightarrow \textit{state} \Rightarrow \textit{level} \Rightarrow \textit{bool}
\textbf{where}
\textit{s = s'} (\le l) == (\forall x. \textit{sec x} \le l \rightarrow s x = s' x)

\textbf{abbreviation} \textit{eq-less} :: \textit{state} \Rightarrow \textit{state} \Rightarrow \textit{level} \Rightarrow \textit{bool}
\textbf{where}
\textit{s = s'} (< l) == (\forall x. \textit{sec x} < l \rightarrow s x = s' x)
lemma aval-eq-if-eq-le:
\[
\begin{array}{ll}
  s_1 = s_2 \ (\leq l); & \text{sec } a \leq l \implies \text{aval } a \ s_1 = \text{aval } a \ s_2 \\
\end{array}
\]
by (induct a) auto

lemma bval-eq-if-eq-le:
\[
\begin{array}{ll}
  s_1 = s_2 \ (\leq l); & \text{sec } b \leq l \implies \text{bval } b \ s_1 = \text{bval } b \ s_2 \\
\end{array}
\]
by (induct b) (auto simp add: aval-eq-if-eq-le)

end

theory Sec-Typing imports Sec-Type-Expr begin

9.2 Syntax Directed Typing

inductive sec-type :: nat ⇒ com ⇒ bool ((-/ |-) [0,0] 50) where
  Skip:
  \[ l \vdash \text{SKIP} \ |
  \]
Assign:
  \[ \begin{array}{ll}
    \text{sec } x \geq \text{sec } a; & \text{sec } x \geq l \implies l \vdash x ::= a \ |
  \end{array} \]
Semi:
  \[ l \vdash c_1; \quad l \vdash c_2 \implies l \vdash c_1;c_2 \ |
  \]
If:
  \[ \begin{array}{ll}
    \text{max } (\text{sec } b) \ l \vdash c_1; \quad \text{max } (\text{sec } b) \ l \vdash c_2 \implies l \vdash \text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2 \ |
  \end{array} \]
While:
  \[ \text{max } (\text{sec } b) \ l \vdash c \implies l \vdash \text{WHILE } b \ \text{DO } c \]

code-pred (expected-modes: i => i => bool) sec-type .

value 0 \vdash \text{IF Less } (V 1) (V 0) \ \text{THEN } 1 : = N 0 \ \text{ELSE } \text{SKIP}

value 1 \vdash \text{IF Less } (V 1) (V 0) \ \text{THEN } 1 : = N 0 \ \text{ELSE } \text{SKIP}

value 2 \vdash \text{IF Less } (V 1) (V 0) \ \text{THEN } 1 : = N 0 \ \text{ELSE } \text{SKIP}

inductive-cases [elim!]:
  l \vdash x ::= a \quad l \vdash c_1;c_2 \quad l \vdash \text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2 \quad l \vdash \text{WHILE } b \ \text{DO } c

An important property: anti-monotonicity.

lemma anti-mono: \[ l \vdash c; \quad l' \leq l \implies l' \vdash c \]

apply (induct arbitrary: l' rule: sec-type.induct)

apply (metis sec-type.intros(1))
apply (metis le-trans sec-type.intros(2))

43
apply (metis sec-type.intros(3))
apply (metis If le-refl sup-mono sup-nat-def)
apply (metis While le-refl sup-mono sup-nat-def)
done

lemma confinement: \[ (c, s) \Rightarrow t; l \vdash c \] \Rightarrow s = t (< l)
proof (induct rule: big-step-induct)
  case Skip thus \?case by simp
next
  case Assign thus \?case by auto
next
  case Semi thus \?case by auto
next
  case (IfTrue b s c1)
  hence max (sec b) l \vdash c1 by auto
  hence l \vdash c1 by (metis le-maxI2 anti-mono)
  thus \?case using IfTrue.hyps by metis
next
  case (IfFalse b s c2)
  hence max (sec b) l \vdash c2 by auto
  hence l \vdash c2 by (metis le-maxI2 anti-mono)
  thus \?case using IfFalse.hyps by metis
next
  case WhileFalse thus \?case by auto
next
  case (WhileTrue b s1 c)
  hence max (sec b) l \vdash c by auto
  hence l \vdash c by (metis le-maxI2 anti-mono)
  thus \?case using WhileTrue by metis
qed

theorem noninterference:
\[ (c, s) \Rightarrow s'; (c, t) \Rightarrow t'; 0 \vdash c; s = t (\leq l) \]
\[ \Rightarrow s' = t' (\leq l) \]
proof (induct arbitrary: \(t\) \(t'\) rule: big-step-induct)
  case Skip thus \?case by auto
next
  case (Assign x a s)
  have [simp]: \(t' = t(x := \text{aval} a t)\) using Assign by auto
  have sec x >= sec a using \(\emptyset \vdash x := a\) by auto
  show \?case
  proof auto
    assume sec x \leq l
with \( (\sec x \geq \sec a) \) have \( \sec a \leq l \) by arith
thus aval a s = aval a t
by (rule aval-eq-if-eq-le[OF \( (s = t \ (\leq l)) \)])
next
fix y assume \( y \neq x \) sec y \( \leq l \)
thus \( s \ y = t \ y \) using \( (s = t \ (\leq l)) \) by simp
qed
next
case Semi thus \(?case\) by blast
next
case \((\text{IfTrue } b \ s \ c1 \ s' \ c2)\)
have \( \sec b \vdash c1 \ sec b \vdash c2 \) using IfTrue.prems(2) by auto
show \(?case\)
proof cases
assume \( \sec b \leq l \)
hence \( s = t \ (\leq \sec b) \) using \( (s = t \ (\leq l)) \) by auto
hence \( \text{bval } b \ t \) using \( \text{bval } b \ s \) by (simp add: bval-eq-if-eq-le)
with IfTrue.hyps(3) IfTrue.prems(1,3) \( \sec b \vdash c1 \) is anti-mono
show \(?thesis\) by auto
next
assume \( \neg \sec b \leq l \)
have 1: \( \sec b \vdash \text{IF } b \ \text{THEN } c1 \ \text{ELSE } c2 \)
by (rule sec-type.intros)(simp-all add: \( \sec b \vdash c1 \) \( \sec b \vdash c2 \))
from confinement[OF big-step.\( \text{IfTrue}[OF \text{IfTrue}(1,2)] \) \( \neg \sec b \leq l \)]
have \( s = s' \ (\leq l) \) by auto
moreover
from confinement[OF \( \text{IfTrue}.prems(1) \) \( \neg \sec b \leq l \)]
have \( t = t' \ (\leq l) \) by auto
ultimately show \( s' = t' \ (\leq l) \) using \( (s = t \ (\leq l)) \) by auto
qed
next
case \((\text{IfFalse } b \ s \ c2 \ s' \ c1)\)
have \( \sec b \vdash c1 \ sec b \vdash c2 \) using IfFalse.prems(2) by auto
show \(?case\)
proof cases
assume \( \sec b \leq l \)
hence \( s = t \ (\leq \sec b) \) using \( (s = t \ (\leq l)) \) by auto
hence \( \neg \text{bval } b \ t \) using \( \neg \text{bval } b \ s \) by (simp add: bval-eq-if-eq-le)
with IfFalse.hyps(3) IfFalse.prems(1,3) \( \sec b \vdash c2 \) anti-mono
show \(?thesis\) by auto
next
assume \( \neg \sec b \leq l \)
have 1: \( \sec b \vdash \text{IF } b \ \text{THEN } c1 \ \text{ELSE } c2 \)
by (rule sec-type.intros)(simp-all add: \( \sec b \vdash c1 \) \( \sec b \vdash c2 \))
from confinement[OF big-step.IfFalse[OF IfFalse(1,2)] 1] \( \vdash \) sec b \( \leq \) l
have \( s = s' \ (\leq l) \) by auto
moreover
from confinement[OF IfFalse.prems(1) 1] \( \vdash \) sec b \( \leq \) l
have \( t = t' \ (\leq l) \) by auto
ultimately show \( s' = t' \ (\leq l) \) using \( (s = t \ (\leq l)) \) by auto
qed
next
case ( WhileFalse b s c)
have sec b \vdash c using WhileFalse.prems(2) by auto
show \( ? \) case
proof cases
assume sec b \leq l
hence \( s = t \ (\leq sec b) \) using \( (s = t \ (\leq l)) \) by auto
hence \( \neg \ bval \ b \ t \) using \( \langle \neg \ bval \ b \ s \rangle \) by simp add: bval-eq-if-eq-le
with WhileFalse.prems(1,3) show \( ? \) thesis by auto
next
assume \( \neg \ sec \ b \leq l \)
let \( ?w = \text{WHILE} \ b \ DO \ c \)
have sec b \vdash ?w by (rule sec-type.intros)(simp-all add: \langle sec b \vdash c \rangle)
from confinement[OF WhileFalse.prems(1) 1] \( \vdash \) sec b \( \leq \) l
have \( t = t' \ (\leq l) \) by auto
thus \( s = t' \ (\leq l) \) using \( (s = t \ (\leq l)) \) by auto
qed
next
case ( WhileTrue b s1 c s2 s3 t1 t3)
let \( ?w = \text{WHILE} \ b \ DO \ c \)
have sec b \vdash c using WhileTrue.prems(2) by auto
show \( ? \) case
proof cases
assume sec b \leq l
hence \( s1 = t1 \ (\leq sec b) \) using \( (s1 = t1 \ (\leq l)) \) by auto
hence \( \neg \ bval \ b \ t1 \)
using \( \langle \neg \ bval \ b \ s1 \rangle \) by simp add: bval-eq-if-eq-le
then obtain \( t2 \) where \( (c,t1) \Rightarrow t2 \) \( (?w,t2) \Rightarrow t3 \)
using \( \langle (?w,t1) \Rightarrow t3 \rangle \) by auto
from WhileTrue.hyps(5)[OF \langle (?w,t2) \Rightarrow t3 \rangle \langle \neg \ ?w \rangle]
WhileTrue.hyps(3)[OF \langle (c,t1) \Rightarrow t2 \rangle \ anti-mono[OF \langle sec b \vdash c \rangle \langle s1 = t1 \ (\leq l) \rangle]]
show \( ? \) thesis by simp
next
assume \( \neg \ sec \ b \leq l \)
have \( 1: \ sec \ b \vdash \ ?w \) by (rule sec-type.intros)(simp-all add: \langle sec b \vdash c \rangle)
from confinement[OF big-step.WhileTrue[OF WhileTrue(1,2)] 1] \( \vdash \)
sec \( b \leq l \)

have \( s_1 = s_3 \) \((\leq l)\) by auto

moreover

from confinement \([O \text{F WhileTrue.prem}(t) \mid 1] \) \((\sim sec b \leq l)\)

have \( t_1 = t_3 \) \((\leq l)\) by auto

ultimately show \( s_3 = t_3 \) \((\leq l)\) using \( s_1 = t_1 \) \((\leq l)\) by auto

qed

\[ \text{9.3 The Standard Typing System} \]

The predicate \( l \vdash c \) is nicely intuitive and executable. The standard formulation, however, is slightly different, replacing the maximum computation by an antimonotonicity rule. We introduce the standard system now and show the equivalence with our formulation.

\textbf{inductive sec-type’} :: nat \( \Rightarrow \) com \( \Rightarrow \) bool \((/\vdash’/\cdot) [0,0] 50)\ where

\textbf{Skip’}:

\( l \vdash’ \text{SKIP} \)

\textbf{Assign’}:

\[
\begin{align*}
\left[ \text{sec } x \geq \text{sec } a; \text{sec } x \geq l \right] & \implies l \vdash’ x ::= a \\
\text{Semi’}:
\left[ l \vdash’ c_1; l \vdash’ c_2 \right] & \implies l \vdash’ c_1; c_2 \\
\text{If’}:
\left[ \text{sec } b \leq l; l \vdash’ c_1; l \vdash’ c_2 \right] & \implies l \vdash’ \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 \\
\text{While’}:
\left[ \text{sec } b \leq l; l \vdash’ c \right] & \implies l \vdash’ \text{WHILE } b \text{ DO } c \\
\text{anti-mono’}:
\left[ l \vdash’ c; l’ \leq l \right] & \implies l’ \vdash’ c
\end{align*}
\]

\textbf{lemma sec-type-sec-type’} : \( l \vdash c \implies l \vdash’ c \)

\textbf{apply} (induct rule: sec-type.induct)

\textbf{apply} (metis Skip’)

\textbf{apply} (metis Assign’)

\textbf{apply} (metis Semi’)

\textbf{apply} (metis min-max.inf-sup-ord(3) min-max.sup-absorb2 nat-le-linear If’ anti-mono’)

\textbf{by} (metis less-or-eq-imp-le min-max.sup-absorb1 min-max.sup-absorb2 nat-le-linear While’ anti-mono’)

\textbf{lemma sec-type’-sec-type} : \( l \vdash’ c \implies l \vdash c \)

\textbf{apply} (induct rule: sec-type’.induct)

\textbf{apply} (metis Skip)

\textbf{apply} (metis Assign)
apply (metis Semi)
apply (metis min-max.sup-absorb2 If)
apply (metis min-max.sup-absorb2 While)
by (metis anti-mono)

9.4 A Bottom-Up Typing System

inductive sec-type2 :: com ⇒ level ⇒ bool ((|- - : -) [0,0] 50) where
Skip2:
  ⊢ SKIP : l |
Assign2:
  sec x ≥ sec a → ⊢ x ::= a : sec x |
Semi2:
  [[ ⊢ c1 : l1; ⊢ c2 : l2 ]] → ⊢ c1;c2 : min l1 l2 |
If2:
  [[ sec b ≤ min l1 l2; ⊢ c1 : l1; ⊢ c2 : l2 ]]
  → ⊢ IF b THEN c1 ELSE c2 : min l1 l2 |
While2:
  [[ sec b ≤ l; ⊢ c : l ] → ⊢ WHILE b DO c : l]

lemma sec-type2-sec-type'
apply (induct rule: sec-type2.induct)
apply (metis Skip')
apply (metis Assign' eq-imp-le)
apply (metis Semi' anti-mono' min-max.inf.commute min-max.inf-le2)
apply (metis If' anti-mono' min-max.inf-absorb2 min-max.le-iff-inf nat-le-linear)
by (metis While')

lemma sec-type'-sec-type2: l ⊢ c ⇒ ∃ l' ≥ l. ⊢ c : l'
apply (induct rule: sec-type'.induct)
apply (metis Skip2 le-refl)
apply (metis Assign2)
apply (metis Semi2 min-max.inf-greatest)
apply (metis If2 inf-greatest inf-nat-def le-trans)
apply (metis While2 le-trans)
by (metis le-trans)

end

theory Sec-TypingT imports Sec-Type-Expr
begin
9.5 A Termination-Sensitive Syntax Directed System

inductive sec-type :: nat ⇒ com ⇒ bool ((−/ −) [0,0] 50) where
  Skip:
  \[ l ⊢ SKIP \]
  Assign:
  \[ \left[ \begin{array}{l} \text{sec } x \geq \text{sec } a; \ \text{sec } x \geq l \end{array} \right] \implies l ⊢ x ::= a \]
  Semi:
  \[ l \vdash c_1 \implies l \vdash c_2 \implies l \vdash c_1; c_2 \]
  If:
  \[ \left[ \begin{array}{l} \text{max (sec } b \vdash c_1; \ \text{max (sec } b \vdash c_2 ) \end{array} \right] \implies l \vdash \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 \]
  While:
  \[ \text{sec } b = 0 \implies 0 \vdash c \implies 0 \vdash \text{WHILE } b \text{ DO } c \]

code-pred (expected-modes: i => i => bool) sec-type .

inductive-cases [elim!]:
  \[ l \vdash x ::= a \ l \vdash c_1; c_2 \ l \vdash \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 \ l \vdash \text{WHILE } b \text{ DO } c \]

lemma anti-mono: \[ l \vdash c \implies l' \leq l \implies l' \vdash c \]
apply (induct arbitrary: l' rule: sec-type.induct)
apply (metis sec-type.intros(1))
apply (metis le-trans sec-type.intros(2))
apply (metis sec-type.intros(3))
apply (metis If le-refl sup-mono sup-nat-def)
by (metis While le-0-eq)

lemma confinement: \[ (c,s) \Rightarrow t \Rightarrow l \vdash c \Rightarrow s = t \ (< l) \]
proof (induct rule: big-step-induct)
  case Skip thus ?case by simp
next
  case Assign thus ?case by auto
next
  case Semi thus ?case by auto
next
  case (IfTrue b s c1)
  hence max (sec b) l \vdash c1 by auto
  hence l \vdash c1 by (metis le-maxI2 anti-mono)
  thus ?case using IfTrue.hyps by metis
next
  case (IfFalse b s c2)
hence max (sec b) l ⊢ c2 by auto
hence l ⊢ c2 by (metis le-maxI2 anti-mono)
thus ?case using IfFalse.hyps by metis
next
  case WhileFalse thus ?case by auto
next
  case (WhileTrue b s1 c)
  hence l ⊢ c by auto
  thus ?case using WhileTrue by metis
qed

lemma termi-if-non0: l ⊢ c =⇒ l ≠ 0 =⇒ ∃ t. (c,s) ⇒ t
apply (induct arbitrary: s rule: sec-type.induct)
apply (metis big-step.Skip)
apply (metis big-step.Assign)
apply (metis big-step.Semi)
apply (metis IfFalse IfTrue le0 le-antisym le-maxI2)
apply simp
done

theorem noninterference: (c,s) ⇒ s' ⇒ 0 ⊢ c =⇒ s = t (≤ l)
  =⇒ ∃ t'. (c,t) ⇒ t' ∧ s' = t' (≤ l)
proof (induct arbitrary: t rule: big-step-induct)
  case Skip thus ?case by auto
next
  case (Assign x a s)
  have sec x ≥ sec a using ⟨0 ⊢ x ::= a⟩ by auto
  have (x ::= a,t) ⇒ t(x ::= aval a t) by auto
  moreover
  have s(x ::= aval a s) = t(x ::= aval a t) (≤ l)
  proof auto
    assume sec x ≤ l
    with (sec x ≥ sec a) have sec a ≤ l by arith
    thus aval a s = aval a t
    by (rule aval-eq-if-eq-le[OF ⟨s = t (≤ l)⟩])
  next
  fix y assume y ≠ x sec y ≤ l
  thus s y = t y using ⟨s = t (≤ l)⟩ by simp
  qed
ultimately show ?case by blast
next
  case Semi thus ?case by blast
next
  case (IfTrue b s c1 s' c2)
have \( \text{sec } b \vdash c_1 \) \( \text{sec } b \vdash c_2 \) using \( \text{IfTrue.prem} \) by auto

obtain \( t' \) where \( t' \colon (c_1, t) \Rightarrow t's' = t' \leq l \)
using \( \text{IfTrue}(3)[(\text{OF anti-mon})((\text{OF } \text{sec } b \vdash c_1) \text{ IfTrue.prem}(2))] \) by blast

show ?case
proof cases
  assume \( \text{sec } b \leq l \)
  hence \( s = t \leq b \) using \( (s = t \leq l) \) by auto
  hence \( \text{bval } b \) \( t \) by(auto)
  thus \( \text{thesis} \) by(auto)
next
  assume \( \neg \text{sec } b \leq l \)
  hence \( 0: \text{sec } b \neq 0 \) by arith
  have \( 1: \text{sec } b \vdash \text{IF } b \) \( \text{THEN } c_1 \) \( \text{ELSE } c_2 \)
  by(rule \text{sec-type.intros})(\text{simp-all add: (sec } b \vdash c_1) \text{ (sec } b \vdash c_2))
  from \text{confinement}[\text{OF big-step.IIfTrue}(\text{OF IfTrue(1,2)}) 1] (\neg \text{sec } b \leq l)
  have \( s = s' \leq l \) by auto
  moreover
  from \text{termi-if-non0}[\text{OF 1 0, of } t] obtain \( t' \) where
  \( (\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2, t) \Rightarrow t' \) ...
  moreover
  from \text{confinement}[\text{OF this 1}] (\neg \text{sec } b \leq l)
  have \( t = t' \leq l \) by auto
  ultimately
  show \(?case \) using \( (s = t \leq l) \) by auto
qed

next
  case \( \text{IfFalse } b \) \( s \) \( c_2 \) \( s' \) \( c_1 \)
  have \( \text{sec } b \vdash c_1 \) \( \text{sec } b \vdash c_2 \) using \( \text{IfFalse.prem} \) by auto
  obtain \( t' \) where \( t' \colon (c_2, t) \Rightarrow t's' = t' \leq l \)
  using \( \text{IfFalse}(3)[(\text{OF anti-mon})((\text{OF } \text{sec } b \vdash c_2) \text{ IfFalse.prem}(2))] \) by blast
  show \(?case \)
  proof cases
    assume \( \neg \text{sec } b \leq l \)
    hence \( 0: \text{sec } b \neq 0 \) by arith
    have \( 1: \text{sec } b \vdash \text{IF } b \) \( \text{THEN } c_1 \) \( \text{ELSE } c_2 \)
    by(rule \text{sec-type.intros})(\text{simp-all add: (sec } b \vdash c_1) \text{ (sec } b \vdash c_2))
    from \text{confinement}[\text{OF big-step.IIfFalse}(\text{OF IfFalse(1,2)}) 1] (\neg \text{sec } b \leq l)
have \( s = s' (\leq l) \) by auto
moreover
from termi-if-non0[OF 1 0, of \( t \)] obtain \( t' \) where
(If \( b \) THEN \( c1 \) ELSE \( c2 \), \( t \)) \Rightarrow \( t' \)
moreover
from confinement[OF this 1] \( \neg \sec b \leq l \)
have \( t = t' (\leq l) \) by auto
ultimately
show \( ? \) case using \( s = t (\leq l) \) by auto
qed

next

\begin{itemize}
\item \textbf{case (WhileFalse \( b s c \))}
\item hence [simp]: \( \sec b = 0 \) by auto
\item have \( s = t (\leq \sec b) \) using \( s = t (\leq l) \) by auto
\item hence \( \neg \bval b t \) using \( \neg \bval b s \) by (metis bval-eq-if-eq-le le-refl)
\item with WhileFalse.prems(2) show \( ? \) case by auto
\end{itemize}

next

\begin{itemize}
\item \textbf{case (WhileTrue \( b s c s' s'' \))}
\item let \( ?w = \text{WHILE } b \text{ DO } c \)
\item from \( \langle 0 \vdash ?w \rangle \) have [simp]: \( \sec b = 0 \) by auto
\item have \( 0 \vdash c \) using WhileTrue.prems(1) by auto
\item from WhileTrue(3)[OF this WhileTrue.prems(2)]
\item obtain \( t'' \) where \( (c, t) \Rightarrow t'' \) and \( s'' = t'' (\leq l) \) by blast
\item from WhileTrue(5)[OF \( \langle 0 \vdash ?w \rangle \) this(2)]
\item obtain \( t' \) where \( \langle ?w, t'' \rangle \Rightarrow t' \) and \( s' = t' (\leq l) \) by blast
\item from \( \langle \bval b s \rangle \) have \( \bval b t \)
\item using \( \bval-eq-if-eq-le \)[OF \( s = t (\leq l) \)] by auto
\item show \( ? \) case
\item using big-step.WhileTrue[OF \( \langle \bval b t ; (c, t) \Rightarrow t'' \rangle \langle ?w, t'' \rangle \Rightarrow t' \)\]
\item by (metis \( \langle s' = t' (\leq l) \rangle \))
\end{itemize}

qed

\section{The Standard Termination-Sensitive System}

The predicate \( l \vdash c \) is nicely intuitive and executable. The standard formulation, however, is slightly different, replacing the maximum computation by an antimonomonicity rule. We introduce the standard system now and show the equivalence with our formulation.

\begin{itemize}
\item \textbf{inductive sec-type\':} \( \text{nat} \Rightarrow \text{com} \Rightarrow \text{bool} \) \( ((/- \vdash'' -) \; [0,0] \, \langle 50 \rangle) \) \where
\item Skip':
\item \( l \vdash' \text{SKIP} \mid \)
\item Assign':
\item \( \begin{cases}
\text{sec } x \geq \text{sec } a; \text{sec } x \geq l \\
\Rightarrow l \vdash' x ::= a
\end{cases}\)
\item Semi':
\end{itemize}
lemma \( l \vdash c \Rightarrow l \vdash' c \Rightarrow l \vdash' c_1;c_2 \ | \)

\textbf{If}:
\[ [ \text{sec } b \leq l; \ l \vdash' c_1; \ l \vdash' c_2 ] \Rightarrow l \vdash' \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 \ | \]

\textbf{While}:
\[ [ \text{sec } b = 0; \ 0 \vdash' c ] \Rightarrow 0 \vdash' \text{WHILE } b \text{ DO } c \ | \]

\textbf{anti-mono}:
\[ [ l \vdash' c; \ l' \leq l ] \Rightarrow l' \vdash' c \]

\textbf{lemma} \( l \vdash c \Rightarrow l \vdash' c \)
\textbf{apply} (induct rule: sec-type.induct)
\textbf{apply} (metis Skip')
\textbf{apply} (metis Assign')
\textbf{apply} (metis Semi')
\textbf{apply} (metis min-max.inf-sup-ord(3) min-max.sup-absorb2 nat-le-linear If')
\textbf{apply} (anti-mono')
\textbf{by} (metis While')

\textbf{lemma} \( l \vdash' c \Rightarrow l \vdash c \)
\textbf{apply} (induct rule: sec-type',induct)
\textbf{apply} (metis Skip)
\textbf{apply} (metis Assign)
\textbf{apply} (metis Semi)
\textbf{apply} (metis min-max.sup-absorb2 If)
\textbf{apply} (metis While)
\textbf{by} (metis anti-mono)

\textbf{end}

\section{Hoare Logic}

\textbf{theory} \textit{Hoare} \textbf{imports} \textit{Big-Step} \textbf{begin}

\textbf{10.1 Hoare Logic for Partial Correctness}

\textbf{types} \textit{assn} = \textit{state} \Rightarrow \textit{bool}

\textbf{abbreviation} \textit{state-subst} :: \textit{state} \Rightarrow \textit{aexp} \Rightarrow \textit{name} \Rightarrow \textit{state}
\texttt{([-/-] [1000,0,0] 999)}
\textbf{where} \textit{s}[a/x] == \textit{s}(x := \text{aval } a s)

\textbf{inductive}
\textit{hoare} :: \textit{assn} \Rightarrow \textit{com} \Rightarrow \textit{assn} \Rightarrow \textit{bool} (\vdash ((1\cdot)/ (\cdot)/ \{(1\cdot)/\}) 50)

53
where

Skip: \vdash \{ P \} \text{SKIP} \{ P \} |

Assign: \vdash \{ \lambda s. P(s[a/x]) \} \ x::=a \ \{ P \} |

Semi: \vdash \{ P \} \ c_1 \ \{ Q \} ; \vdash \{ Q \} \ c_2 \ \{ R \} \ \Rightarrow \ \vdash \{ P \} \ c_1;c_2 \ \{ R \} |

If: \vdash \{ \lambda s. P \ s \land bval \ b \ s \} \ c_1 \ \{ Q \} ; \vdash \{ \lambda s. P \ s \land \neg bval \ b \ s \} \ c_2 \ \{ Q \} \ \Rightarrow \ \vdash \{ P \} \ IF b THEN c_1 ELSE c_2 \ \{ Q \} |

While: \vdash \{ \lambda s. P \ s \land bval \ b \ s \} \ c \ \{ P \} \ \Rightarrow \ \vdash \{ P \} \ WHILE b DO c \ \{ \lambda s. P \ s \land \neg bval \ b \ s \} |

conseq: [ \forall s. P' \ s \to P \ s ; \vdash \{ P \} \ c \ \{ Q \} ; \forall s. Q \ s \to Q' \ s ] \ \Rightarrow \ \vdash \{ P' \} \ c \ \{ Q' \}

lemmas [simp] = hoare.Skip hoare.Assign hoare.Semi If

lemmas [intro!] = hoare.Skip hoare.Assign hoare.Semi hoare.If

lemma strengthen-pre:
[ \forall s. P' \ s \to P \ s ; \vdash \{ P \} \ c \ \{ Q \} ] \Rightarrow \vdash \{ P' \} \ c \ \{ Q \}
by (blast intro: conseq)

lemma weaken-post:
[ \vdash \{ P \} \ c \ \{ Q \} ; \forall s. Q \ s \to Q' \ s ] \Rightarrow \vdash \{ P \} \ c \ \{ Q' \}
by (blast intro: conseq)

The assignment and While rule are awkward to use in actual proofs because their pre and postcondition are of a very special form and the actual goal would have to match this form exactly. Therefore we derive two variants with arbitrary pre and postconditions.

lemma Assign': \forall s. P \ s \to Q(s[a/x]) \Rightarrow \vdash \{ P \} \ x::=a \ \{ Q \}
by (simp add: strengthen-pre[OF - Assign])

lemma While':
assumes \vdash \{ \lambda s. P \ s \land bval \ b \ s \} \ c \ \{ P \} \ \text{and} \ \forall s. P \ s \land \neg bval \ b \ s \to Q \ s
shows \vdash \{ P \} \ WHILE b DO c \ \{ Q \}
by (rule weaken-post[OF While[OF assms(1)] assms(2)])

end
10.2 Example: Sums

Summing up the first \( n \) natural numbers. The sum is accumulated in variable 0, the loop counter is variable 1.

abbreviation \( wn == \)
\[
\text{WHILE Less (V 1) (N n)} \text{ DO (} 1 ::= \text{Plus (V 1) (N 1); } 0 ::= \text{Plus (V 0) (V 1)} \text{)}
\]

For this example we make use of some predefined functions. Function \( \text{Setsum} \), also written \( \sum \), sums up the elements of a set. The set of numbers from \( m \) to \( n \) is written \( \{m..n\} \).

10.2.1 Proof by Operational Semantics

The behaviour of the loop is proved by induction:

lemma \( \text{while-sum} \):
\[
(w n, s) \Rightarrow t \quad \Rightarrow \quad t 0 = s 0 + \text{Setsum} \{s 1 + 1 \ldots n\}
\]
apply(induct \( w n s t \) rule: \text{big-step-induct})
apply(auto simp add: \text{setsum-head-Suc})
done

We were lucky that the proof was practically automatic, except for the induction. In general, such proofs will not be so easy. The automation is partly due to the right inversion rules that we set up as automatic elimination rules that decompose big-step premises.

Now we prefix the loop with the necessary initialization:

lemma \( \text{sum-via-bigstep} \):
assumes \( (0 ::= N 0; 1 ::= N 0; w n, s) \Rightarrow t \)
shows \( t 0 = \text{Setsum} \{1 \ldots n\} \)
proof -
from assms have \( (w n, s(0:=0,1:=0)) \Rightarrow t \) by auto
from \( \text{while-sum}[OF this] \) show ?thesis by simp
qed

10.2.2 Proof by Hoare Logic

Note that we deal with sequences of commands from right to left, pulling back the postcondition towards the precondition.

lemma \( \{\lambda s. \text{True}\} 0 ::= N 0; 1 ::= N 0; w n \{\lambda s. \text{s 0 = Setsum} \{1 \ldots n\}\} \)
apply(rule \text{hoare.Semi})
prefer 2

55
The proof is intentionally an apply skript because it merely composes the rules of Hoare logic. Of course, in a few places side conditions have to be proved. But since those proofs are 1-liners, a structured proof is overkill. In fact, we shall learn later that the application of the Hoare rules can be automated completely and all that is left for the user is to provide the loop invariants and prove the side-conditions.

end
thus unfolding hoare-valid-def by blast
qed (auto simp: hoare-valid-def)

10.4 Weakest Precondition

definition wp :: com ⇒ assn ⇒ assn where
wp c Q = (λs. ∀ t. (c,s) ⇒ t ⇒ Q t)

lemma wp-SKIP[simp]: wp SKIP Q = Q
by (rule ext) (auto simp: wp-def)

lemma wp-Ass[simp]: wp (x::=a) Q = (λs. Q(s[a/x]))
by (rule ext) (auto simp: wp-def)

lemma wp-Semi[simp]: wp (c1;c2) Q = wp c1 (wp c2 Q)
by (rule ext) (auto simp: wp-def)

lemma wp-If[simp]:
wp (IF b THEN c1 ELSE c2) Q =
(λs. (bval b s ⇒ wp c1 Q s) ∧ (¬ bval b s ⇒ wp c2 Q s))
by (rule ext) (auto simp: wp-def)

lemma wp-While-If:
wp (WHILE b DO c) Q s =
wp (IF b THEN c;WHILE b DO c ELSE SKIP) Q s
unfolding wp-def by (metis unfold-while)

lemma wp-While-True[simp]: bval b s ⇒
wp (WHILE b DO c) Q s = wp (c; WHILE b DO c) Q s
by(simp add: wp-While-If)

lemma wp-While-False[simp]: ¬ bval b s ⇒ wp (WHILE b DO c) Q s = Q s
by(simp add: wp-While-If)

10.5 Completeness

lemma wp-is-pre: ⊢ {wp c Q} c {Q}
proof (induct c arbitrary: Q)
case Semi thus ?case by (auto intro: Semi)
next
case (If b c1 c2)
let ?If = IF b THEN c1 ELSE c2
show \texttt{?case}

\textbf{proof} (rule \textit{hoare}.\textit{If})

show \vdash \{\lambda s\ldotp \text{wp } ?\text{If } Q s \land \text{bval } b s\} \ c1 \ \{Q\}

\textbf{proof} (rule \textit{strengthen-pre}[OF - \textit{If}(1)])

show \forall s. \text{wp } ?\text{If } Q s \land \text{bval } b s \rightarrow \text{wp } c1 \ Q s \ \text{by auto}

\texttt{qed}

show \vdash \{\lambda s\ldotp \text{wp } ?\text{If } Q s \land \neg \text{bval } b s\} \ c2 \ \{Q\}

\textbf{proof} (rule \textit{strengthen-pre}[OF - \textit{If}(2)])

show \forall s. \text{wp } ?\text{If } Q s \land \neg \text{bval } b s \rightarrow \text{wp } c2 \ Q s \ \text{by auto}

\texttt{qed}

\texttt{qed}

\texttt{next}

\texttt{case (While } b \ c)\texttt{)

let \texttt{?w = WHILE } b \ \texttt{DO } c

have \vdash \{\text{wp } ?w \ Q\} \ ?w \ \{\lambda s\ldotp \text{wp } ?w \ Q s \land \neg \text{bval } b s\}

\textbf{proof} (rule \textit{hoare}.\textit{While})

show \vdash \{\lambda s\ldotp \text{wp } ?w \ Q s \land \text{bval } b s\} \ c \ \{wp \ ?w \ Q\}

\textbf{proof} (rule \textit{strengthen-pre}[OF - \textit{While}(1)])

show \forall s. \text{wp } ?w \ Q s \land \text{bval } b s \rightarrow \text{wp } c \ (\text{wp } ?w \ Q) \ s \ \text{by auto}

\texttt{qed}

\texttt{qed}

\texttt{thus } \texttt{?case}

\textbf{proof} (rule \textit{weaken-post})

show \forall s. \text{wp } ?w \ Q s \land \neg \text{bval } b s \rightarrow Q s \ \text{by auto}

\texttt{qed}

\texttt{qed auto}

\textbf{lemma} \textit{hoare-relative-complete}: \texttt{assumes } \vdash \{P\} c \{Q\} \texttt{ shows } \vdash \{P\} c \{Q\}

\textbf{proof} (rule \textit{strengthen-pre})

\textbf{show} \forall s. \text{P } s \rightarrow \text{wp } c \ Q s \texttt{ using } \texttt{assms}

\texttt{by (auto simp: hoare-valid-def wp-def)}

\textbf{show} \vdash \{\text{wp } c \ Q\} \ c \ \{Q\} \texttt{ by (rule wp-is-pre)}

\texttt{qed}

\texttt{end}

\section{Verification Conditions}

\textbf{theory} \textit{VC} \textbf{imports} \textit{Hoare} \textbf{begin}
11.1 VCG via Weakest Preconditions

Annotated commands: commands where loops are annotated with invariants.

datatype acom = Askip | Aassign name aexp | Asemi acom acom | Aif bexp acom acom | Awhile bexp assn acom

Weakest precondition from annotated commands:

fun pre :: acom ⇒ assn ⇒ assn where
pre Askip Q = Q |
pre (Aassign x a) Q = (λs. Q(s(x := aval a s))) |
pre (Asemi c1 c2) Q = pre c1 (pre c2 Q) |
preat (Aif b c1 c2) Q = (λs. (bval b s → pre c1 Q s) ∧
   (¬ bval b s → pre c2 Q s)) |
preat (Awhile b I c) Q = I

Verification condition:

fun vc :: acom ⇒ assn ⇒ assn where
vc Askip Q = (λs. True) |
vc (Aassign x a) Q = (λs. True) |
vc (Asemi c1 c2) Q = (λs. vc c1 Q s ∧ vc c2 Q s) |
vc (Aif b c1 c2) Q = (λs. vc c1 Q s ∧ vc c2 Q s) |
vc (Awhile b I c) Q =
   (λs. (I s ∧ ¬ bval b s → Q s) ∧
   (I s ∧ bval b s → pre c I s) ∧
   vc c I s)

Strip annotations:

fun astrip :: acom ⇒ com where
astrip Askip = SKIP |
astrip (Aassign x a) = (x::=a) |
astrip (Asemi c1 c2) = (astrip c1; astrip c2) |
astrip (Aif b c1 c2) = (IF b THEN astrip c1 ELSE astrip c2) |
astrip (Awhile b I c) = (WHILE b DO astrip c)

11.2 Soundness

lemma vc-sound: ∀ s. vc c Q s → {pre c Q} astrip c {Q}
proof (induct c arbitrary: Q)
case (Awhile b I c)
show ?case
proof\(\) (simp, rule While')
from \(\forall s. vc\ (Awhile b I c) Q s\)
have \(vc\ :: \forall s. vc c I s\) and \(IQ\ :: \forall s. I s \land \neg bval b s \rightarrow Q s\) and
pre: \(\forall s. I s \land bval b s \rightarrow pre c I s\) by simp-all
have \(\vdash\ \{pre c I\}\) astrip c \(\{I\}\) by (rule Awhile.hyps[\(OF\ vc\])
with pre show \(\vdash\ \{\lambda s. I s \land bval b s\}\) astrip c \(\{I\}\)
by (rule strengthen-pre)
show \(\forall s. I s \land \neg bval b s \rightarrow Q s\) by (rule IQ)
qed

\[\text{corollary} \quad \text{vc-sound}':\]
\[\forall s. vc c Q s \land \forall s. P s \rightarrow pre c Q s \implies \vdash \{P\}\ astrip c \{Q\}\]
by (metis strengthen-pre vc-sound)

11.3 Completeness

lemma pre-mono:
\(\forall s. P s \rightarrow P' s \implies pre c P s \implies pre c P' s\)
proof (induct c arbitrary: \(P P' s\))
\[\text{case} \quad \text{Asemi} \quad \text{thus} \quad ?\text{case} \quad \text{by simp metis}\]
qed simp-all

lemma vc-mono:
\(\forall s. P s \rightarrow P' s \implies vc c P s \implies vc c P' s\)
proof (induct c arbitrary: \(P P'\))
\[\text{case} \quad \text{Asemi} \quad \text{thus} \quad ?\text{case} \quad \text{by simp (metis pre-mono)}\]
qed simp-all

lemma vc-complete:
\(\vdash \{P\} c\{Q\} \implies \exists c'. astrip c' = c \land (\forall s. vc c' Q s) \land (\forall s. P s \rightarrow pre c' Q s)\)
(is - \(\implies\) \(\exists c'. ?G P c Q c'\))
proof (induct rule: hoare.induct)
\[\text{case} \quad \text{Skip} \quad \text{show} \quad ?\text{case} \quad (is \ \exists ac. \ ?C ac)\]
proof show \(?C Askip\) by simp qed
next
\[\text{case} \quad (\text{Assign } P a x) \quad \text{show} \quad ?\text{case} \quad (is \ \exists ac. \ ?C ac)\]
proof show \(?C(Aassign x a)\) by simp qed
next
\[\text{case} \quad (\text{Semi } P c1 Q c2 R)\]
from Semi.hyps obtain \(ac1\) where \(ih1: ?G P c1 Q ac1\) by blast
from Semi.hyps obtain ac2 where ih2: ?G Q c2 R ac2 by blast

show ?case (is ∃ ac. ?C ac)

proof
  show ?C(Asemi ac1 ac2)
    using ih1 ih2 by (fastsimp elim!: pre-mono vc-mono)

qed

next

  case (If P b c1 Q c2)
  from If.hyps obtain ac1 where ih1: ?G (λs. P s ∧ bval b s) c1 Q ac1 by blast
  from If.hyps obtain ac2 where ih2: ?G (λs. P s ∧ ¬bval b s) c2 Q ac2 by blast
  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: ?G (λs. P s ∧ bval b s) c P ac by blast
  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!: pre-mono vc-mono)

qed

11.4 An Optimization

fun vcpre :: acom ⇒ assn ⇒ assn × assn where
vcpre Askip Q = (λs. True, Q) |
vcre (Aassign x a) Q = (λs. True, λs. Q(s[a/x])) |
vcre (Asemi c1 c2) Q =
  (let (vc2,wp2) = vcpre c2 Q;
   (vc1,wp1) = vcpre c1 wp2
   in (λs. vc1 s ∧ vc2 s, wp1)) |
vcre (Aif b c1 c2) Q =
  (let (vc2,wp2) = vcpre c2 Q;
   (vc1,wp1) = vcpre c1 Q
   in (λs. vc1 s ∧ vc2 s, λs. (bval b s → wp1 s) ∧ (¬bval b s → wp2 s))) |
vcre (Awhile b I c) Q =
  (let (vcc,wpc) = vcpre c I
   in (λs. (I s ∧ ¬ bval b s → Q s) ∧
      (I s ∧ bval b s → wpc s) ∧ vcc s, I))
lemma vcpre-vc-pre: vcpre c Q = (vc c Q, pre c Q)
by (induct c arbitrary: Q) (simp-all add: Let-def)
end

12 Hoare Logic for Total Correctness

definition hoare-tvalid :: assn ⇒ com ⇒ assn ⇒ bool
  (|=t {{(I-)}/ (-)/ {{(I-)}} 50) where
  |=t {{P} c{Q}} ≡ ∀ s. P s → (∃ t. (c,s) ⇒ t ∧ Q t)

  Proveability of Hoare triples in the proof system for total correctness is
  written ⊢t {{P} c{Q}} and defined inductively. The rules for ⊢t differ from
  those for ⊢ only in the one place where nontermination can arise: the While-
  rule.

  inductive hoaret :: assn ⇒ com ⇒ assn ⇒ bool (⊢t {{(I-)}/ (-)/ {{(I-)}} 50)
  where
  Skip: ⊢t {{P} SKIP {P}} |
  Assign: ⊢t {λs. P(s[a/x])} x:=a {P} |
  Semi: [ ⊢t {P3} c1 {P2}; ⊢t {P2} c2 {P3} ] ⇒ ⊢t {c1;c2} {P3} |
  If: [ ⊢t {λs. P s ∧ bval b s} c1 {Q}; ⊢t {λs. P s ∧ ¬ bval b s} c2 {Q} ]
  ⇒ ⊢t {IP b THEN c1 ELSE c2} {Q} |
  While:
  [ ∀ n::nat. ⊢t {λs. P s ∧ bval b s ∧ f s = n} c {λs. P s ∧ f s < n}]
  ⇒ ⊢t {P} WHILE b DO c {λs. P s ∧ ¬bval b s} |
  conseq: [ ∀ s. P' s → P s; ⊢t {P}c{Q}; ∀ s. Q s → Q' s ]
  ⇒ ⊢t {P'}c{Q'}

  The While-rule is like the one for partial correctness but it requires addition-
  ally that with every execution of the loop body some measure function
  f :: state ⇒ nat decreases.

lemma strengthen-pre:
  [ ∀ s. P' s → P s; ⊢t {P} c {Q} ] ⇒ ⊢t {P'} c {Q}
by (metis conseq)

lemma weaken-post:
  [ ⊢t {P} c {Q}; ∀ s. Q s → Q' s ] ⇒ ⊢t {P} c {Q'}
by (metis conseq)

lemma Assign': \( \forall s. \ P(s) \rightarrow Q(s[a/x]) \implies \vdash t \ {x := a \ \{Q\}} \)
by (simp add: strengthen-pre[OF - Assign])

lemma While':
assumes \( \bigwedge n::\text{nat}. \ \vdash \{P_s \land bval \ b \ s \land f \ s = n\} \ c \ \{\lambda s. \ P(s) \land f < n\} \)
and \( \forall s. \ P(s) \land \neg bval \ b \ s \rightarrow Q(s) \)
shows \( \vdash \{P\} \ \text{WHILE} \ b \ \text{DO} \ c \ \{Q\} \)
by (blast intro: assms(1) weaken-post[OF While assms(2)])

Our standard example:

abbreviation w n ==
\text{WHILE} \ \text{Less} (V 1) (N n)
\text{DO} \ (1 ::\text{= Plus} (V 1) (N 1); 0 ::= \text{Plus} (V 0) (V 1))

lemma \( \vdash t \ {\lambda s. \ True} \ 0 ::= N 0; 1 ::= N 0; w \ n \ \{\lambda s. s 0 = \sum \{1 .. n\}\} \)
apply(rule Semi)
prefer 2
apply(rule While')
[where \( P = \lambda s. s 0 = \sum \{1 .. s 1\} \land s 1 \leq n \)
and \( f = \lambda s. n - s 1\)]
apply(rule Semi)
prefer 2
apply(rule Assign)
apply(rule Assign')
apply simp
apply arith
apply fastsimp
apply(rule Semi)
prefer 2
apply(rule Assign)
apply(rule Assign')
apply simp

The soundness theorem:

theorem hoaret-sound: \( \vdash t \ {P \ c\{Q\}} \implies |= t \ {P \ c\{Q\}} \)
proof (unfold hoare-tvalid-def, induct rule: hoaret.induct)
case (While P b f c)
show ?case
proof
fix s
show \( P \ s \rightarrow (\exists t. \ (\text{WHILE} \ b \ \text{DO} \ c, \ s) \Rightarrow t \land P \ t \land \neg bval \ b \ t) \)

63
proof (induct f s arbitrary: s rule: less-induct)
  case (less n)
  thus ?case by (metis While(2) WhileFalse WhileTrue)
qed

next
  case If thus ?case by auto blast
qed fastsimp+

The completeness proof proceeds along the same lines as the one for partial correctness. First we have to strengthen our notion of weakest precondition to take termination into account:

definition wpt :: com ⇒ assn ⇒ assn (wpt)
where
wp t c Q ≡ λs. ∃t. (c,s) ⇒ t ∧ Q t

lemma [simp]: wp t SKIP Q = Q
by (auto intro!: ext simp: wpt-def)

lemma [simp]: wp t (x ::= e) Q = (λs. Q(s(x := aval e s)))
by (auto intro!: ext simp: wpt-def)

lemma [simp]: wp t (c1;c2) Q = wp t c1 (wp t c2 Q)
unfolding wpt-def
apply (rule ext)
apply auto
done

lemma [simp]:
wp t (IF b THEN c1 ELSE c2) Q = (λs. wp t (if bval b s then c1 else c2) Q s)
apply (unfold wpt-def)
apply (rule ext)
apply auto
done

Now we define the number of iterations WHILE b DO c needs to terminate when started in state s. Because this is a truly partial function, we define it as an (inductive) relation first:

inductive Its :: bexp ⇒ com ⇒ state ⇒ nat ⇒ bool where
Its-0: ¬ bval b s ⇒ Its b c s 0 |
Its-Suc: [ bval b s; (c,s) ⇒ s′; Its b c s′ n ] ⇒ Its b c s (Suc n)

The relation is in fact a function:

lemma Its-fun: Its b c s n ⇒ Its b c s n′ ⇒ n=n′
proof (induct arbitrary: \( n' \) rule: Its.induct)

  case Its-0
  from this(1) Its.cases[OF this(2)] show \(?case by metis \\
  next
  case (Its-Suc b s c s' n n')
  note \( C = \) this
  from this(5) show \(?case \\
  proof cases \\
    case Its-0 with Its-Suc(1) show \(?thesis by blast \\
  next \\
    case Its-Suc with C show \(?thesis by (metis big-step-determ)
  qed
  qed

For all terminating loops, Its yields a result:

lemma WHILE-Its: (WHILE b DO c, s) \( \Rightarrow t \) \( \Rightarrow \exists n. \) Its b c s n
proof (induct WHILE b DO c s t rule: big-step-induct)
  case WhileFalse thus \(?case by (metis Its-0)
  next \\
  case WhileTrue thus \(?case by (metis Its-Suc)
  qed

Now the relation is turned into a function with the help of the description operator \( \text{THE} \):

definition its :: bexp \( \Rightarrow \) com \( \Rightarrow \) state \( \Rightarrow \) nat where
its b c s = (THE n. Its b c s n)

The key property: every loop iteration increases its by 1.

lemma its-Suc: \( \begin{array}{l}
[ \text{bval b s}; (c, s) \Rightarrow s'; (\text{WHILE b DO c, s'}) \Rightarrow t] \\
\Rightarrow its b c s = \text{Suc}(its b c s')
\end{array} \)
by (metis its-def WHILE-Its Its.intro(2) Its.fun the-equality)

lemma wpt-is-pre: \( \vdash t \{ \text{wp}_t c Q \} c \{ Q \}
proof (induct c arbitrary: Q)
  case SKIP show \(?case by simp (blast intro:hoaret.Skip)
  next
  case Assign show \(?case by simp (blast intro:hoaret.Assign)
  next
  case Semi thus \(?case by simp (blast intro:hoaret.Semi)
  next
  case If thus \(?case by simp (blast intro:hoaret.If hoaret.conseq)
  next
  case (While b c)
let \( \exists w = \text{WHILE} \, b \, \text{DO} \, c \)

\{ fix \, n \\
have \( \forall s. \, \wp_t \, \exists w \, Q \, s \wedge \text{bval} \, b \, s \wedge \text{its} \, c \, s = n \rightarrow \wp_t \, c \, (\lambda s'. \, \wp_t \, \exists w \, Q \, s' \wedge \text{its} \, b \, c \, s' < n) \, s \)

unfolding \( \wp_t \)-def by \((\text{metis WhileE its-Suc lessI})\)

note strengthen-pre[OF this While]

moreover have \( \forall s. \, \wp_t \, \exists w \, Q \, s \wedge \neg \text{bval} \, b \, s \rightarrow Q \, s \) by \((\text{auto simp add:wp_t-def})\)

ultimately show \( ?\text{case} \) by \((\text{rule weaken-post})\)

qed

In the \( \text{While} \)-case, \( \text{its} \) provides the obvious termination argument.

The actual completeness theorem follows directly, in the same manner as for partial correctness:

theorem hoaret-complete: \( \vdash_t \{ P \} c \{ Q \} \Rightarrow \vdash_t \{ P \} c \{ Q \} \)

apply \((\text{rule strengthen-pre[OF - wp_t-is-pre]})\)

apply \((\text{auto simp: hoare-tvalid-def hoare-valid-def wp_t-def})\)

done

end

13 Extensions and Variations of IMP

theory Procs imports BExp begin

13.1 Procedures and Local Variables

datatype com = SKIP

| Assign name aexp (\( \cdot \ ::= \cdot \) | [1000, 61] 61)

| Semi com com (\( \cdot \ ::= \cdot \) | [60, 61] 60)

| If bexp com com ((IF \( \cdot \apsed \) THEN \( \cdot \apsed \) ELSE \( \cdot \apsed \)) | [0, 0, 61] 61)

| While bexp com ((WHILE \( \cdot \apsed \) DO \( \cdot \apsed \)) | [0, 61] 61)

| Var name com ((1{VAR \apsed \apsed \apsed})

| Proc name com com ((1{PROC \apsed \apsed \apsed})

| CALL name

definition test-com =

\{ VAR 0; \} 0 ::= N 0;

\{PROC 0 = 0 ::= \text{Plus} \, (V \, 0) \, (V \, 0);\}

\{PROC 1 = \text{CALL} \, 0;\} 66
VAR 0::
0 ::= N 5;
{PROC 0 = 0 ::= Plus (V 0) (N 1);;
CALL 1; 1 ::= V 0}}}

end

theory Procs-Dyn-Vars-Dyn imports Util Procs
begin

13.1.1 Dynamic Scoping of Procedures and Variables

types penv = name ⇒ com

inductive big-step :: penv ⇒ com × state ⇒ state ⇒ bool (- ⊢ - ⇒ - [60,0,60] 55)
where
Skip: pe ⊢ (SKIP, s) ⇒ s |
Assign: pe ⊢ (x ::= a,s) ⇒ s(x ::= aval a s) |
Semi: [ pe ⊢ (c1,s1) ⇒ s2; pe ⊢ (c2,s2) ⇒ s3 ] ⇒
      pe ⊢ (c1;c2, s1) ⇒ s3 |
IfTrue: [ bval b s; pe ⊢ (c1,s) ⇒ t ] ⇒
        pe ⊢ (IF b THEN c1 ELSE c2, s) ⇒ t |
IfFalse: [ ¬bval b s; pe ⊢ (c2,s) ⇒ t ] ⇒
         pe ⊢ (IF b THEN c1 ELSE c2, s) ⇒ t |
WhileFalse: ¬bval b s ⇒ pe ⊢ (WHILE b DO c,s) ⇒ s |
WhileTrue:
 [ bval b s1; pe ⊢ (c,s1) ⇒ s2; pe ⊢ (WHILE b DO c, s2) ⇒ s3 ] ⇒
    pe ⊢ (WHILE b DO c, s1) ⇒ s3 |
Var: pe ⊢ (c,s) ⇒ t ⇒
      pe ⊢ ({VAR x; c}, s) ⇒ t(x ::= s x) |
Call: pe ⊢ (pe p, s) ⇒ t ⇒
      pe ⊢ (CALL p, s) ⇒ t |
Proc: pe(p := cp) ⊢ (c,s) ⇒ t ⇒
      pe ⊢ ({PROC p = cp; c}, s) ⇒ t

code-pred big-step .

inductive exec where
(λp. SKIP) ⊢ (c,nth ns) ⇒ s ⇒ exec c ns (list s (length ns))

code-pred exec .
values \{ ns. \text{exec} (CALL 0) [42,43] ns \}

values \{ ns. \text{exec test-com} [0,0] ns \}

end

theory \textit{Procs-Stat-Vars-Dyn} imports \textit{Util Procs}
begin

13.1.2 Static Scoping of Procedures, Dynamic of Variables

types \textit{penv} = (name \times \text{com}) \text{list}

inductive \textit{big-step} :: \textit{penv} \Rightarrow \text{com} \times \text{state} \Rightarrow \text{state} \Rightarrow \text{bool} (- \vdash - \Rightarrow - [60,0,60] 55)

where

\textbf{Skip}: \quad \textit{pe} \vdash (\text{SKIP}, s) \Rightarrow s |

\textbf{Assign}: \quad \textit{pe} \vdash (x ::= a, s) \Rightarrow s(x := \text{aval} a s) |

\textbf{Semi}: \quad [ \textit{pe} \vdash (c_1, s_1) \Rightarrow s_2; \textit{pe} \vdash (c_2, s_2) \Rightarrow s_3 ] \Rightarrow

\quad \textit{pe} \vdash (c_1;c_2, s_1) \Rightarrow s_3 |

\textbf{IfTrue}: \quad [ \text{bval} b s; \textit{pe} \vdash (c_1, s) \Rightarrow t ] \Rightarrow

\quad \textit{pe} \vdash (IF b \text{ THEN } c_1 \text{ ELSE } c_2, s) \Rightarrow t |

\textbf{IfFalse}: \quad [ \neg \text{bval} b s; \textit{pe} \vdash (c_2, s) \Rightarrow t ] \Rightarrow

\quad \textit{pe} \vdash (IF b \text{ THEN } c_1 \text{ ELSE } c_2, s) \Rightarrow t |

\textbf{WhileFalse}: \quad \neg \text{bval} b s \Rightarrow \textit{pe} \vdash (\text{WHILE } b \text{ DO } c, s) \Rightarrow s |

\textbf{WhileTrue}: \quad [ \text{bval} b s_1; \textit{pe} \vdash (c, s_1) \Rightarrow s_2; \textit{pe} \vdash (\text{WHILE } b \text{ DO } c, s_2) \Rightarrow s_3 ] \Rightarrow

\quad \textit{pe} \vdash (\text{WHILE } b \text{ DO } c, s_1) \Rightarrow s_3 |

\textbf{Var}: \quad \textit{pe} \vdash (c, s) \Rightarrow t \quad \Rightarrow \quad \textit{pe} \vdash (\{ \text{VAR } x; c \}, s) \Rightarrow t(x := s x) |

\textbf{Call1}: \quad (p,c)\#\textit{pe} \vdash (c, s) \Rightarrow t \quad \Rightarrow \quad (p,c)\#\textit{pe} \vdash (\text{CALL } p, s) \Rightarrow t |

\textbf{Call2}: \quad [ p' \neq p; \textit{pe} \vdash (\text{CALL } p, s) \Rightarrow t ] \Rightarrow

\quad (p',c)\#\textit{pe} \vdash (\text{CALL } p, s) \Rightarrow t |

\textbf{Proc}: \quad (p,cp)\#\textit{pe} \vdash (c,s) \Rightarrow t \quad \Rightarrow \quad \textit{pe} \vdash (\{ \text{PROC } p = cp; c \}, s) \Rightarrow t |

code-pred \textit{big-step} .

inductive \textit{exec} where

\[ \vdash \textit{c,nth} \text{ ns} \Rightarrow s \Rightarrow \textit{exec} \text{ c ns (list s (length ns))} \]
code-pred exec .

values { ns. exec (CALL 0) [42,43] ns }
values { ns. exec test-com [0,0] ns }
end

theory Procs-Stat-Vars-Stat imports Util Procs
begin

13.1.3 Static Scoping of Procedures and Variables

types

addr = nat
venv = name ⇒ addr
store = addr ⇒ nat
penv = (name × com × venv) list

fun venv :: penv × venv × nat ⇒ venv where
venv(-,-,→) = ve

inductive

big-step :: penv × venv × nat ⇒ com × store ⇒ store ⇒ bool
(- ⊢ - ⇒ - [60,0,60] 55)

where

Skip: e ⊢ (SKIP,s) ⇒ s |
Assign: (pe,ve,f) ⊢ (x ::= a,s) ⇒ s(ve x := aval a (s o ve)) |
Semi: e ⊢ (c1,s1) ⇒ s2; e ⊢ (c2,s2) ⇒ s3 ⇒
        e ⊢ (c1;c2, s1) ⇒ s3 |
IfTrue: bval b (s o venv e); e ⊢ (c1,s) ⇒ t ⇒
        e ⊢ (IF b THEN c1 ELSE c2, s) ⇒ t |
IfFalse: ¬ bval b (s o venv e); e ⊢ (c2,s) ⇒ t ⇒
         e ⊢ (IF b THEN c1 ELSE c2, s) ⇒ t |
WhileFalse: ¬ bval b (s o venv e) ⇒ e ⊢ (WHILE b DO c,s) ⇒ s |
WhileTrue:
        [ bval b (s1 o venv e); e ⊢ (c,s1) ⇒ s2;
          e ⊢ (WHILE b DO c, s2) ⇒ s3 ] ⇒
         e ⊢ (WHILE b DO c, s1) ⇒ s3 |

Var: (pe,ve(x:=f),f+1) ⊢ (c,s) ⇒ t ⇒

69
\[(pe, ve, f) \vdash \{ \text{VAR } x ; c \}, s \Rightarrow t(x := s x) \mid \]

**Call1:** \[((p, c, ve) # pe, ve, f) \vdash (c, s) \Rightarrow t \Rightarrow ((p, c, ve) # pe, ve', f) \vdash (\text{CALL } p, s) \Rightarrow t \mid ((p', c, ve') # pe, ve, f) \vdash (\text{CALL } p, s) \Rightarrow t \mid \]

**Call2:** \[[ p' \neq p; (pe, ve, f) \vdash (\text{CALL } p, s) \Rightarrow t \] \Rightarrow ((p', c, ve') # pe, ve, f) \vdash (\text{CALL } p, s) \Rightarrow t \mid \]

**Proc:** \[((p, cp, ve) # pe, ve, f) \vdash (c, s) \Rightarrow t \Rightarrow (pe, ve, f) \vdash (\{ \text{PROC } p = cp ; c \}, s) \Rightarrow t \mid \]

*code-pred* big-step.

**inductive** exec where
\[(\emptyset, \lambda n. n, 0) \vdash (c, \text{nth } ns) \Rightarrow s \Rightarrow \text{exec } c \text{ ns (list } s \text{ (length } ns)) \]

*code-pred* exec.

**values** \{ ns. exec (CALL 0) [42,43] ns \}

**values** \{ ns. exec test-com [0,0] ns \}

*end*

**theory** C-like imports Util begin

### 13.2 A C-like Language

**types** state = nat \Rightarrow nat

**datatype** aexp = N nat | Deref aexp (!) | Plus aexp aexp

**fun** aval :: aexp \Rightarrow state \Rightarrow nat where
\[\text{aval} (N n) s = n \mid \text{aval} (!a) s = s(\text{aval } a s) \mid \text{aval} (\text{Plus } a_1 a_2) s = \text{aval } a_1 s + \text{aval } a_2 s \]

**datatype** bexp = B bool | Not bexp | And bexp bexp | Less aexp aexp

**primrec** bval :: bexp \Rightarrow state \Rightarrow bool where
\[\text{bval} (B bv) = bv \mid \text{bval} (\text{Not } b) s = (\neg \text{bval } b s) \mid \text{bval} (\text{And } b_1 b_2) s = (\text{if } \text{bval } b_1 s \text{ then } \text{bval } b_2 s \text{ else False}) \mid \text{bval} (\text{Less } a_1 a_2) s = (\text{aval } a_1 s < \text{aval } a_2 s) \]
datatype
  com = SKIP
  | Assign aexp aexp (- ::= - [61, 61] 61)
  | New aexp aexp
  | Semi com com (::/- [60, 61] 60)
  | If bexp com com ((IF -/ THEN -/ ELSE -) [0, 0, 61] 61)
  | While bexp com ((WHILE -/ DO -) [0, 61] 61)

inductive
  big-step :: com × state × nat ⇒ state × nat ⇒ bool (infix ⇒ 55)
where
  Skip: (SKIP, sn) ⇒ sn |
  Assign: (lhs ::= a, s, n) ⇒ (s(aval lhs s := aval a), n) |
  New: (New lhs a, s, n) ⇒ (s(aval lhs s := n), n + aval a) |
  Semi: [[ (c1, sn1) ⇒ sn2; (c2, sn2) ⇒ sn3 ] ⇒
          (c1;c2, sn1) ⇒ sn3 ]|
  IfTrue: [ bval b s; (c1, s, n) ⇒ tn ] ⇒
          (IF b THEN c1 ELSE c2, s, n) ⇒ tn |
  IfFalse: [ ¬bval b s; (c2, s, n) ⇒ tn ] ⇒
          (IF b THEN c1 ELSE c2, s, n) ⇒ tn |
  WhileFalse: ¬bval b s ⇒ (WHILE b DO c, s, n) ⇒ (s, n) |
  WhileTrue: [ bval b s1; (c, s1, n) ⇒ sn2; (WHILE b DO c, sn2) ⇒ sn3 ] ⇒
             (WHILE b DO c, s1, n) ⇒ sn3

code-pred big-step .

inductive exec :: com ⇒ nat list ⇒ nat list ⇒ bool where
(c,nth sl,length sl) ⇒ (s’,n) ⇒⇒ exec c sl (list s’n)

code-pred exec .

Examples:

definition
  array-sum =
  WHILE Less (!((N 0))) (Plus (!((N 1))) (N 1))
  DO ( N 2 ::= Plus (!((N 2))) (!((N 0)));
       N 0 ::= Plus (!((N 0))) (N 1) )

values { sl. exec array-sum [3,4,0,3,7] sl }
definition
linked-list-sum =
WHILE Less (N 0) (! (N 0))
DO (N 1 ::= Plus (! (N 1)) (! (N 0)));
    N 0 ::= !(Plus (! (N 0)) (N 1));
values { sl. exec linked-list-sum [4,0,3,0,7,2] sl }

definition
array-init =
New (N 0) (! (N 1)); N 2 ::= !(N 0);
WHILE Less (! (N 2)) (Plus (! (N 0)) (! (N 1)))
DO ( !(N 2) ::= !(N 2);
    N 2 ::= Plus (! (N 2)) (N 1))
values { sl. exec array-init [5,2,7] sl }

definition
linked-list-init =
WHILE Less (! (N 1)) (! (N 0))
DO (New (N 3) (N 2);
    N 1 ::= Plus (! (N 1)) (N 1);
    !(N 3) ::= !(N 1);
    Plus (! (N 3)) (N 1) ::= !(N 2);
    N 2 ::= !(N 3))
values { sl. exec linked-list-init [2,0,0,0] sl }

definitions
end

theory OO imports Util begin

13.3 Towards an OO Language: A Language of Records

abbreviation fun-upd2 :: ('a ⇒ 'b ⇒ 'c) ⇒ 'a ⇒ 'b ⇒ 'c ⇒ 'c
    (-/((2,- := /)') [1000,0,0,0] 900)
where f(x,y := z) == f(x := (f x)(y := z))

types addr = nat
datatype ref = null | Ref addr

types
    obj = string ⇒ ref
tenv = string ⇒ ref
store = addr ⇒ obj

datatype exp =
    Null |
    New |
    V string |
    Faccess exp string (⋯ [63,1000] 63) |
    Vassign string exp (⋯ ::= -) [1000,61] 62 |
    Fassign exp string exp ((⋯ ::= -) [63,0,62] 62) |
    Mcall exp string exp (⋯<-> [63,0,0] 63) |
    Semi exp exp (⋯/ - [61,60] 60) |
If bexp exp exp (IF - THEN / ELSE - [0,0,61] 61)
and bexp = B bool | Not bexp | And bexp bexp | Eq exp exp

types
    menv = string ⇒ exp
    config = venv × store × addr

inductive
    big-step :: menv ⇒ exp × config ⇒ ref × config ⇒ bool
    ((⋯ ⊢ / (⋯ - / -)) [60,0,60] 55) and
    bval :: menv ⇒ bexp × config ⇒ bool × config ⇒ bool
    ((⋯ - - - [60,0,60] 55)
where
    Null:
        me ⊢ (Null,c) ⇒ (null,c) |
    New:
        me ⊢ (New,ve,s,n) ⇒ (Ref n,ve,s(n := (λf. null)),n+1) |
        Vaccess:
            me ⊢ (V x,ve,sn) ⇒ (ve x,ve,sn) |
        Faccess:
            me ⊢ (e,c) ⇒ (Ref a,ve',s',n') ⇒⇒
            me ⊢ (e·f,c) ⇒ (s' a f,ve',s',n') |
        Vassign:
            me ⊢ (e,c) ⇒ (r,ve',sn') ⇒⇒
            me ⊢ (x ::= e,c) ⇒ (r,ve'(x:=r),sn') |
        Fassign:
            me ⊢ (oe,c1) ⇒ (Ref a,c2); me ⊢ (e,c2) ⇒ (r,ve3,s3,n3) ] ⇒⇒
            me ⊢ (oe·f ::= e,c1) ⇒ (r,ve3,s3(a,f := r),n3) |
        Mcall:
            me ⊢ (oe,c1) ⇒ (or,c2); me ⊢ (pe,c2) ⇒ (pr,ve3,sn3);
            ve = (λx. null)("this" := or, "param" := pr);
            me ⊢ (me m,ve,sn3) ⇒ (r,ve',sn4) ]
⇒⇒

73
me ⊢ (oe·m<pe>,c₁) ⇒ (r,ve₃,sn₄) |

Semi:
[ me ⊢ (e₁,c₁) ⇒ (r,c₂); me ⊢ (e₂,c₂) ⇒ c₃ ] ===
me ⊢ (e₁; e₂,c₁) ⇒ c₃ |

IfTrue:
[ me ⊢ (b,c₁) → (True,c₂); me ⊢ (e₁,c₂) ⇒ c₃ ] ===
me ⊢ (IF b THEN e₁ ELSE e₂,c₁) ⇒ c₃ |

IfFalse:
[ me ⊢ (b,c₁) → (False,c₂); me ⊢ (e₂,c₂) ⇒ c₃ ] ===
me ⊢ (IF b THEN e₁ ELSE e₂,c₁) ⇒ c₃ |

me ⊢ (B bv,c) → (bv,c) |

me ⊢ (b,c₁) → (bv,c₂) === me ⊢ (Not b,c₁) → (¬bv,c₂) |

[ me ⊢ (b₁,c₁) → (bv₁,c₂); me ⊢ (b₂,c₂) → (bv₂,c₃) ] ===
me ⊢ (And b₁ b₂,c₁) → (bv₁∧bv₂,c₃) |

[ me ⊢ (e₁,c₁) ⇒ (r₁,c₂); me ⊢ (e₂,c₂) ⇒ (r₂,c₃) ] ===
me ⊢ (Eq e₁ e₂,c₁) → (r₁=r₂,c₃)

code-pred (modes: i => i => o => bool) big-step .

Execution of semantics. The final variable environment and store are
transformed into lists of references based on given lists of variable and field
names to extract.

inductive exec :: menv ⇒ exp ⇒ string list ⇒ string list
⇒ ref ⇒ ref list ⇒ ref list list ⇒ bool where
me ⊢ (e,(λx. null),nth[],0) ⇒ (r,ve′,s′,n) ===
exec me e xs fs r (map ve′ xs) (map (λn. map (s′ n) fs) [0..<n])

code-pred exec .

Example: natural numbers encoded as objects with a predecessor field.
Null is zero. Method succ adds an object in front, method add adds as many
objects in front as the parameter specifies.

First, the method bodies:

definition
m-succ = ("s" ::= New)."pred" ::= V "this"; V "s"

definition m-add =
IF Eq (V "param") Null
THEN V "this"
ELSE V "this"."succ"<Null>."add"<V "param"."pred">

The method environment:

**definition**

\[ \text{menv} = (\lambda m. \text{Null})("\text{succ} := m-\text{succ}, \text{add} := m-\text{add}) \]

The main code, adding 1 and 2:

**definition** \[ \text{main} = \]

\[ "1" := \text{Null}.;"\text{succ}"<\text{Null}>; \]
\[ "2" := V "1"."\text{succ}"<\text{Null}>; \]
\[ V "2" \cdot "\text{add}" < V "1" \]

**values** \{(r,vl,ol). exec menv main ["1","2"] ["pred"] r vl ol\}

**end**