# Semantics

TN

January 28, 2011

## Contents

# 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* $e_1$ $e_2$) *st* = *aval* $e_1$ *st* + *aval* $e_2$ *st*

Subscripts are for visual beauty only!

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

## 1.2 Optimization

Evaluate constant subsexpressions:

**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

end

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
```

**end**

# 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-acomp*[*simp*]: *aexec* (*acomp 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* × *state* ⇒ *state* ⇒ *bool* (**infix** ⇒ *55*)
**where**
*Skip*: (*SKIP*,*s*) ⇒ *s* |
*Assign*: ($x$ ::= $a$,$s$) ⇒ $s$($x$ := *aval* $a$ $s$) |
*Semi*: ⟦ ($c_1$,$s_1$) ⇒ $s_2$; ($c_2$,$s_2$) ⇒ $s_3$ ⟧ ⟹
  ($c_1$;$c_2$, $s_1$) ⇒ $s_3$ |

*IfTrue*: ⟦ *bval* $b$ $s$; ($c_1$,$s$) ⇒ $t$ ⟧ ⟹
  (*IF* $b$ *THEN* $c_1$ *ELSE* $c_2$, $s$) ⇒ $t$ |
*IfFalse*: ⟦ ¬*bval* $b$ $s$; ($c_2$,$s$) ⇒ $t$ ⟧ ⟹
  (*IF* $b$ *THEN* $c_1$ *ELSE* $c_2$, $s$) ⇒ $t$ |

*WhileFalse*: ¬*bval* $b$ $s$ ⟹ (*WHILE* $b$ *DO* $c$,$s$) ⇒ $s$ |
*WhileTrue*: ⟦ *bval* $b$ $s_1$; ($c$,$s_1$) ⇒ $s_2$; (*WHILE* $b$ *DO* $c$, $s_2$) ⇒ $s_3$ ⟧ ⟹
  (*WHILE* $b$ *DO* $c$, $s_1$) ⇒ $s_3$

**schematic-lemma** *ex*: (*0* ::= *N 5*; *2* ::= *V 0*, *s*) ⇒ *?t*
**apply**(*rule Semi*)
**apply**(*rule Assign*)
**apply** *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*]) ⇒ *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*) ⇒ $s$ ⟹ *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** $\{ns.$
 $exec$
  $(WHILE\ Less\ (V\ 0)\ (V\ 1)\ DO\ (0\ ::=\ Plus\ (V\ 0)\ (N\ 5)))$
  $[0,13]\ ns\}$

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

Proof automation:

**declare** *big-step.intros* [*intro*]

The standard induction rule

$\llbracket x1 \Rightarrow x2;\ \bigwedge s.\ P\ (SKIP,\ s)\ s;\ \bigwedge x\ a\ s.\ P\ (x\ ::=\ a,\ s)\ (s(x\ :=\ aval\ a\ s));$
$\bigwedge c_1\ s_1\ s_2\ c_2\ s_3.$
$\llbracket (c_1,\ s_1) \Rightarrow s_2;\ P\ (c_1,\ s_1)\ s_2;\ (c_2,\ s_2) \Rightarrow s_3;\ P\ (c_2,\ s_2)\ s_3 \rrbracket$
$\Longrightarrow P\ (c_1;\ c_2,\ s_1)\ s_3;$
$\bigwedge b\ s\ c_1\ t\ c_2.$
$\llbracket bval\ b\ s;\ (c_1,\ s) \Rightarrow t;\ P\ (c_1,\ s)\ t \rrbracket \Longrightarrow P\ (IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s)$
$t;$
$\bigwedge b\ s\ c_2\ t\ c_1.$
$\llbracket \neg\ bval\ b\ s;\ (c_2,\ s) \Rightarrow t;\ P\ (c_2,\ s)\ t \rrbracket \Longrightarrow P\ (IF\ b\ THEN\ c_1\ ELSE\ c_2,$
$s)\ t;$
$\bigwedge b\ s\ c.\ \neg\ bval\ b\ s \Longrightarrow P\ (WHILE\ b\ DO\ c,\ s)\ s;$
$\bigwedge b\ s_1\ c\ s_2\ s_3.$
$\llbracket bval\ b\ s_1;\ (c,\ s_1) \Rightarrow s_2;\ P\ (c,\ s_1)\ s_2;\ (WHILE\ b\ DO\ c,\ s_2) \Rightarrow s_3;$
$P\ (WHILE\ b\ DO\ c,\ s_2)\ s_3 \rrbracket$
$\Longrightarrow P\ (WHILE\ b\ DO\ c,\ s_1)\ s_3 \rrbracket$
$\Longrightarrow P\ x1\ x2$

**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*

$\llbracket (x1a,\ x1b) \Rightarrow x2a;\ \bigwedge s.\ P\ SKIP\ s\ s;\ \bigwedge x\ a\ s.\ P\ (x\ ::=\ a)\ s\ (s(x\ :=\ aval\ a$
$s));$
$\bigwedge c_1\ s_1\ s_2\ c_2\ s_3.$
$\llbracket (c_1,\ s_1) \Rightarrow s_2;\ P\ c_1\ s_1\ s_2;\ (c_2,\ s_2) \Rightarrow s_3;\ P\ c_2\ s_2\ s_3 \rrbracket$
$\Longrightarrow P\ (c_1;\ c_2)\ s_1\ s_3;$
$\bigwedge b\ s\ c_1\ t\ c_2.$
$\llbracket bval\ b\ s;\ (c_1,\ s) \Rightarrow t;\ P\ c_1\ s\ t \rrbracket \Longrightarrow P\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ s\ t;$
$\bigwedge b\ s\ c_2\ t\ c_1.$
$\llbracket \neg\ bval\ b\ s;\ (c_2,\ s) \Rightarrow t;\ P\ c_2\ s\ t \rrbracket \Longrightarrow P\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ s\ t;$
$\bigwedge b\ s\ c.\ \neg\ bval\ b\ s \Longrightarrow P\ (WHILE\ b\ DO\ c)\ s\ s;$

$\bigwedge b\ s_1\ c\ s_2\ s_3.$
$[\![bval\ b\ s_1;\ (c,\ s_1) \Rightarrow s_2;\ P\ c\ s_1\ s_2;\ (WHILE\ b\ DO\ c,\ s_2) \Rightarrow s_3;$
$P\ (WHILE\ b\ DO\ c)\ s_2\ s_3]\!]$
$\Longrightarrow P\ (WHILE\ b\ DO\ c)\ s_1\ s_3]\!]$
$\Longrightarrow P\ x1a\ x1b\ x2a$

### 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*) ⇒ *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*) ⇒ *t*
**thm** *AssignE*
**inductive-cases** *SemiE*[*elim*!]: (*c1*;*c2*,*s1*) ⇒ *s3*
**thm** *SemiE*
**inductive-cases** *IfE*[*elim*!]: (*IF b THEN c1 ELSE c2*,*s*) ⇒ *t*
**thm** *IfE*

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

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

An automatic example:

**lemma** (*IF b THEN SKIP ELSE SKIP*, *s*) ⇒ *t* ⟹ *t* = *s*
**by** *blast*

Rule inversion by hand via the "cases" method:

**lemma assumes** (*IF b THEN SKIP ELSE SKIP*, *s*) ⇒ *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* ⇒ *com* ⇒ *bool* (**infix** ∼ *50*) **where**
  $c \sim c' == (\forall s\ t.\ (c,s) \Rightarrow t\ =\ (c',s) \Rightarrow t)$

Warning: ∼ 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*) ∼ (*IF b THEN c*; *WHILE b DO c ELSE SKIP*) (**is** *?w* ∼ *?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*) ⇒ *t*
    — as a first thing we note that, if *b* is *False* in state *s*,
    — then both statements do nothing:
    **{ assume** ¬*bval b s*
      **hence** *t* = *s* **using** ‹(*?w*,*s*) ⇒ *t*› **by** *blast*
      **hence** (*?iw*, *s*) ⇒ *t* **using** ‹¬*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*) ⇒ *t*
    **{ assume** *bval b s*
      **with** ‹(*?w*, *s*) ⇒ *t*› **obtain** *s′* **where**
        (*c*, *s*) ⇒ *s′* **and** (*?w*, *s′*) ⇒ *t* **by** *auto*
      — now we can build a derivation tree for the *IF*
      — first, the body of the True-branch:
      **hence** (*c*; *?w*, *s*) ⇒ *t* **by** (*rule Semi*)
      — then the whole *IF*
      **with** ‹*bval b s*› **have** (*?iw*, *s*) ⇒ *t* **by** (*rule IfTrue*)
    **}**
    **ultimately**
    — both cases together give us what we want:
    **have** (*?iw*, *s*) ⇒ *t* **by** *blast*
  **}**
  **moreover**
  — now the other direction:

**{ fix** *s t* **assume** (*?iw*, *s*) ⇒ *t*
— again, if *b* is *False* in state *s*, then the False-branch
— of the *IF* is executed, and both statements do nothing:
**{ assume** ¬*bval b s*
**hence** *s* = *t* **using** ⟨(*?iw*, *s*) ⇒ *t*⟩ **by** *blast*
**hence** (*?w*, *s*) ⇒ *t* **using** ⟨¬*bval b s*⟩ **by** *blast*
**}**
**moreover**
— on the other hand, if *b* is *True* in state *s*,
— then this time only the *IfTrue* rule can have be used
**{ assume** *bval b s*
**with** ⟨(*?iw*, *s*) ⇒ *t*⟩ **have** (*c*; *?w*, *s*) ⇒ *t* **by** *auto*
— and for this, only the Semi-rule is applicable:
**then obtain** *s'* **where**
(*c*, *s*) ⇒ *s'* **and** (*?w*, *s'*) ⇒ *t* **by** *auto*
— with this information, we can build a derivation tree for the *WHILE*

**with** ⟨*bval b s*⟩
**have** (*?w*, *s*) ⇒ *t* **by** (*rule WhileTrue*)
**}**
**ultimately**
— both cases together again give us what we want:
**have** (*?w*, *s*) ⇒ *t* **by** *blast*
**}**
**ultimately**
**show** *?thesis* **by** *blast*
**qed**

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

**lemma**
(*WHILE b DO c*) ∼ (*IF b THEN c*; *WHILE b DO c ELSE SKIP*)
**by** *blast*

**lemma** *triv-if*:
(*IF b THEN c ELSE c*) ∼ *c*
**by** *blast*

**lemma** *commute-if*:
(*IF b1 THEN* (*IF b2 THEN c11 ELSE c12*) *ELSE c2*)
∼
(*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** *big-step-determ*: $[\![\ (c,s) \Rightarrow t;\ (c,s) \Rightarrow u\ ]\!] \Longrightarrow u = t$
**apply** (*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**
  $(c,s) \Rightarrow t \implies (c,s) \Rightarrow t' \implies t' = t$
**proof** (*induct arbitrary*: $t'$ *rule*: *big-step.induct*)
  — the only interesting case, *WhileTrue*:
  **fix** $b\ c\ s\ s1\ t\ t'$
  — The assumptions of the rule:
  **assume** *bval b s* **and** $(c,s) \Rightarrow s1$ **and** $(WHILE\ b\ DO\ c,s1) \Rightarrow t$
  — Ind.Hyp; note the $\bigwedge$ because of arbitrary:
  **assume** *IHc*: $\bigwedge t'.\ (c,s) \Rightarrow t' \Longrightarrow t' = s1$
  **assume** *IHw*: $\bigwedge t'.\ (WHILE\ b\ DO\ c,s1) \Rightarrow t' \Longrightarrow t' = t$
  — Premise of implication:
  **assume** $(WHILE\ b\ DO\ c,s) \Rightarrow t'$
  **with** ⟨*bval b s*⟩ **obtain** $s1'$ **where**
      *c*: $(c,s) \Rightarrow s1'$ **and**
      *w*: $(WHILE\ b\ DO\ c,s1') \Rightarrow t'$
    **by** *auto*
  **from** *c IHc* **have** $s1' = s1$ **by** *blast*
  **with** *w IHw* **show** $t' = t$ **by** *blast*
**qed** *blast+* — prove the rest automatically

**end**

# 4 Small-Step Semantics of Commands

**theory** *Small-Step* **imports** *Big-Step* **begin**

### 4.1 The transition relation

**inductive**
  *small-step* :: $com * state \Rightarrow com * state \Rightarrow bool$ (**infix** $\to$ *55*)
**where**
*Assign*: $(x ::= a,\ s) \to (SKIP,\ s(x := aval\ a\ s))$ |

*Semi1*: $(SKIP;c_2,s) \to (c_2,s)$ |
*Semi2*: $(c_1,s) \to (c_1',s') \Longrightarrow (c_1;c_2,s) \to (c_1';c_2,s')$ |

*IfTrue*: $bval\ b\ s \Longrightarrow (IF\ b\ THEN\ c_1\ ELSE\ c_2,s) \to (c_1,s)$ |
*IfFalse*: $\neg bval\ b\ s \Longrightarrow (IF\ b\ THEN\ c_1\ ELSE\ c_2,s) \to (c_2,s)$ |

*While*: $(WHILE\ b\ DO\ c,s) \to (IF\ b\ THEN\ c;\ WHILE\ b\ DO\ c\ ELSE\ SKIP,s)$

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

## 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 $a \to b \Longrightarrow \ldots$ 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 → into pairs:

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

Similarly for →∗:

**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*]

**lemma** *step1*[*simp*, *intro*]: $cs \to cs' \Longrightarrow cs \to\!* \ cs'$
**by**(*metis refl step*)

So called transitivity rules. See below.

**declare** *step*[*trans*] *step1*[*trans*]

**lemma** *step2*[*trans*]:
$cs \to cs' \Longrightarrow cs' \to cs'' \Longrightarrow cs \to\!* \ cs''$
**by**(*metis refl step*)

**lemma** *small-steps-trans*[*trans*]:
$cs \to\!* \ cs' \Longrightarrow cs' \to\!* \ cs'' \Longrightarrow cs \to\!* \ cs''$
**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*) → *ct*
**thm** *SkipE*
**inductive-cases** *AssignE*[*elim*!]: (*x*::=*a*,*s*) → *ct*
**thm** *AssignE*
**inductive-cases** *SemiE*[*elim*]: (*c1*;*c2*,*s*) → *ct*
**thm** *SemiE*
**inductive-cases** *IfE*[*elim*!]: (*IF b THEN c1 ELSE c2*,*s*) → *ct*
**inductive-cases** *WhileE*[*elim*]: (*WHILE b DO c*, *s*) → *ct*

A simple property:

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

## 4.4 Equivalence with big-step semantics

**lemma** *rtrancl-semi2*: $(c1,s) \to\!* \ (c1',s') \Longrightarrow (c1;c2,s) \to\!* \ (c1';c2,s')$
**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*:
 $\llbracket$ (*c1*,*s1*) →∗ (*SKIP*,*s2*); (*c2*,*s2*) →∗ (*SKIP*,*s3*) $\rrbracket$
 ⟹ (*c1*;*c2*, *s1*) →∗ (*SKIP*,*s3*)
**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 → and →∗ 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*:
 *cs* ⇒ *t* ⟹ *cs* →∗ (*SKIP*,*t*)
**proof** (*induct rule*: *big-step.induct*)
 **fix** *s* **show** (*SKIP*,*s*) →∗ (*SKIP*,*s*) **by** *simp*
**next**
 **fix** *x a s* **show** (*x* ::= *a*,*s*) →∗ (*SKIP*, *s*(*x* := *aval a s*)) **by** *auto*
**next**
 **fix** *c1 c2 s1 s2 s3*
 **assume** (*c1*,*s1*) →∗ (*SKIP*,*s2*) **and** (*c2*,*s2*) →∗ (*SKIP*,*s3*)
 **thus** (*c1*;*c2*, *s1*) →∗ (*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*) → (*c0*,*s*) **by** *simp*
 **also assume** (*c0*,*s*) →∗ (*SKIP*,*t*)
 **finally show** (*IF b THEN c0 ELSE c1*,*s*) →∗ (*SKIP*,*t*) **.** — = by assumption
**next**
 **fix** *s*::*state* **and** *b c0 c1 t*
 **assume** ¬*bval b s*
 **hence** (*IF b THEN c0 ELSE c1*,*s*) → (*c1*,*s*) **by** *simp*
 **also assume** (*c1*,*s*) →∗ (*SKIP*,*t*)
 **finally show** (*IF b THEN c0 ELSE c1*,*s*) →∗ (*SKIP*,*t*) **.**
**next**
 **fix** *b c* **and** *s*::*state*
 **assume** *b*: ¬*bval b s*
 **let** *?if* = *IF b THEN c*; *WHILE b DO c ELSE SKIP*
 **have** (*WHILE b DO c*,*s*) → (*?if*, *s*) **by** *blast*
 **also have** (*?if*,*s*) → (*SKIP*, *s*) **by** (*simp add*: *b*)
 **finally show** (*WHILE b DO c*,*s*) →∗ (*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: (?w,s') →* (SKIP,t)
  assume c: (c,s) →* (SKIP,s')
  assume b: bval b s
  have (?w,s) → (?if, s) by blast
  also have (?if, s) → (c; ?w, s) by (simp add: b)
  also have (c; ?w,s) →* (SKIP,t) by(rule semi-comp[OF c w])
  finally show (WHILE b DO c,s) →* (SKIP,t) by auto
qed
```

Each case of the induction can be proved automatically:

```
lemma cs ⇒ t ⟹ cs →* (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 → cs' ⟹ cs' ⇒ t ⟹ cs ⇒ t
apply (induct arbitrary: t rule: small-step.induct)
apply auto
done

lemma small-big-continue:
  cs →* cs' ⟹ cs' ⇒ t ⟹ cs ⇒ t
apply (induct rule: small-steps.induct)
apply (auto intro: small1-big-continue)
done

lemma small-to-big: cs →* (SKIP,t) ⟹ cs ⇒ t
```

**by** (*metis small-big-continue Skip*)

Finally, the equivalence theorem:

**theorem** *big-iff-small*:
 $cs \Rightarrow t = cs \rightarrow* (SKIP,t)$
**by**(*metis big-to-small small-to-big*)

### 4.5 Final configurations and infinite reductions

**definition** *final cs* $\longleftrightarrow \neg(EX\ cs'.\ cs \rightarrow cs')$

**lemma** *finalD*: *final* $(c,s) \Longrightarrow c = SKIP$
**apply**(*simp add*: *final-def*)
**apply**(*induct c*)
**apply** *blast+*
**done**

**lemma** *final-iff-SKIP*: *final* $(c,s) = (c = SKIP)$
**by** (*metis SkipE finalD final-def*)

Now we can show that ⇒ yields a final state iff → terminates:

**lemma** *big-iff-small-termination*:
 $(EX\ t.\ cs \Rightarrow t) \longleftrightarrow (EX\ cs'.\ cs \rightarrow* cs' \wedge final\ cs')$
**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 → is determininistic, 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

**datatype** *instr* =
 *PUSH-N nat* | *PUSH-V nat* | *ADD* |
 *STORE nat* |
 *JMPF nat* |
 *JMPB nat* |
 *JMPFLESS 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**
⟦ *i* < *size P*; *P*!*i* = *PUSH-N n* ⟧ ⟹
*P* ⊢ (*i*,*s*,*stk*) → (*i*+*1*,*s*, *n*#*stk*) |
⟦ *i* < *size P*; *P*!*i* = *PUSH-V x* ⟧ ⟹
*P* ⊢ (*i*,*s*,*stk*) → (*i*+*1*,*s*, *s x* # *stk*) |
⟦ *i* < *size P*; *P*!*i* = *ADD* ⟧ ⟹
*P* ⊢ (*i*,*s*,*stk*) → (*i*+*1*,*s*, (*hd2 stk* + *hd stk*) # *tl2 stk*) |
⟦ *i* < *size P*; *P*!*i* = *STORE n* ⟧ ⟹
*P* ⊢ (*i*,*s*,*stk*) → (*i*+*1*,*s*(*n* := *hd stk*),*tl stk*) |
⟦ *i* < *size P*; *P*!*i* = *JMPF n* ⟧ ⟹
*P* ⊢ (*i*,*s*,*stk*) → (*i*+*1*+*n*,*s*,*stk*) |
⟦ *i* < *size P*; *P*!*i* = *JMPB n*; *n* ≤ *i*+*1* ⟧ ⟹
*P* ⊢ (*i*,*s*,*stk*) → (*i*+*1*−*n*,*s*,*stk*) |
⟦ *i* < *size P*; *P*!*i* = *JMPFLESS n* ⟧ ⟹
*P* ⊢ (*i*,*s*,*stk*) → (*if hd2 stk* < *hd stk then i*+*1*+*n else i*+*1*,*s*,*tl2 stk*) |
⟦ *i* < *size P*; *P*!*i* = *JMPFGE n* ⟧ ⟹
*P* ⊢ (*i*,*s*,*stk*) → (*if hd2 stk* >= *hd stk then i*+*1*+*n else i*+*1*,*s*,*tl2 stk*)

**code-pred** *exec1* **.**

**declare** *exec1*.*intros*[*intro*]

**inductive** *exec* :: *instr list* ⇒ *config* ⇒ *config* ⇒ *bool* (-/ ⊢ (- →∗/ -) *50*)
**where**
*refl*: *P* ⊢ *c* →∗ *c* |
*step*: *P* ⊢ *c* → *c*′ ⟹ *P* ⊢ *c*′ →∗ *c*′′ ⟹ *P* ⊢ *c* →∗ *c*′′

**declare** *exec*.*intros*[*intro*]

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

**code-pred** *exec* **.**

Integrating the state to list transformation:

**inductive** *execl* :: *instr list* ⇒ *nat* ⇒ *nat list* ⇒ *stack*
 ⇒ *nat* ⇒ *nat list* ⇒ *stack* ⇒ *bool* **where**
$P \vdash (i,nth\ ns,stk) \rightarrow* (i',s',stk') \Longrightarrow$
 $execl\ P\ i\ ns\ stk\ i'\ (list\ s'\ (size\ ns))\ stk'$

**code-pred** *execl* **.**

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

## 5.2 Verification infrastructure

**lemma** *exec-trans*: $P \vdash c \rightarrow* c' \Longrightarrow P \vdash c' \rightarrow* c'' \Longrightarrow P \vdash c \rightarrow* c''$
**apply**(*induct rule*: *exec.induct*)
 **apply** *blast*
**by** (*metis exec.step*)

**lemma** *exec1-subst*: $P \vdash c \rightarrow c' \Longrightarrow c' = c'' \Longrightarrow P \vdash c \rightarrow c''$
**by** *auto*

**lemmas** *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.

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

**lemma** *exec-appendR*: $P \vdash c \rightarrow* c' \Longrightarrow P@P' \vdash c \rightarrow* c'$
**apply**(*induct rule*: *exec.induct*)
 **apply** *blast*
**by** (*metis exec1-appendR exec.step*)

**lemma** *exec1-appendL*:
**assumes** $P \vdash (i,s,stk) \rightarrow (i',s',stk')$
**shows** $P'\ @\ P \vdash (size(P')+i,s,stk) \rightarrow (size(P')+i',s',stk')$
**proof**−
 **from** *assms* **show** *?thesis*
 **by** *cases* (*simp-all add*: *exec1-simps*)
**qed**

**lemma** *exec-appendL*:
$P \vdash (i,s,stk) \rightarrow* (i',s',stk') \implies$
$P' @ P \vdash (size(P')+i,s,stk) \rightarrow* (size(P')+i',s',stk')$
**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*]:
$P \vdash (i,s,stk) \rightarrow* (j,t,stk') \implies$
$instr\#P \vdash (Suc\ i,s,stk) \rightarrow* (Suc\ j,t,stk')$
**apply**(*drule exec-appendL*[**where** $P'$=[*instr*]])
**apply** *simp*
**done**

**lemma** *exec-appendL-if*[*intro*]:
$size\ P' <= i$
$\implies P \vdash (i - size\ P',s,stk) \rightarrow* (i',s',stk')$
$\implies P' @ P \vdash (i,s,stk) \rightarrow* (size\ P' + i',s',stk')$
**apply**(*drule exec-appendL*[**where** $P'=P'$])
**apply** *simp*
**done**

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

**lemma** *exec-append-trans*[*intro*]:
$P \vdash (0,s,stk) \rightarrow* (i',s',stk') \implies$
$size\ P \leq i' \implies$
$P' \vdash (i' - size\ P,s',stk') \rightarrow* (i'',s'',stk'') \implies$
$j'' = size\ P + i''$
$\implies$
$P @ P' \vdash (0,s,stk) \rightarrow* (j'',s'',stk'')$
**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
acomp (N n) = [PUSH-N n] |
acomp (V n) = [PUSH-V n] |
acomp (Plus a1 a2) = acomp a1 @ acomp a2 @ [ADD]

lemma acomp-correct[intro]:
  acomp a ⊢ (0,s,stk) →* (size(acomp a),s,aval a s#stk)
apply(induct a arbitrary: stk)
apply(fastsimp)+
done

fun bcomp :: bexp ⇒ bool ⇒ nat ⇒ instr list where
bcomp (B v) c n = (if v=c then [JMPF n] else []) |
bcomp (Not b) c n = bcomp b (¬c) n |
bcomp (And b1 b2) c n =
 (let cb2 = bcomp b2 c n;
      m = (if c then size cb2 else size cb2+n);
      cb1 = bcomp b1 False m
  in cb1 @ cb2) |
bcomp (Less a1 a2) c n =
 acomp a1 @ acomp a2 @ (if c then [JMPFLESS n] else [JMPFGE n])

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

lemma bcomp-correct[intro]:
 bcomp b c n ⊢
 (0,s,stk)  →*  (size(bcomp b c n) + (if c = bval b s then n else 0),s,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
ccomp SKIP = [] |
ccomp (x ::= a) = acomp a @ [STORE x] |
ccomp (c1;c2) = ccomp c1 @ ccomp c2 |
ccomp (IF b THEN c1 ELSE c2) =
  (let cc1 = ccomp c1; cc2 = ccomp c2; cb = bcomp b False (size cc1 + 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 1)))
```

### 5.4 Preservation of sematics

```
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 (fastsimp 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 (fastsimp)
  moreover
  have ?cc1 @ ?cc2 ⊢ (size ?cc1,s2,stk) →* (size(?cc1 @ ?cc2),s3,stk)
    using Semi.hyps(4) by (fastsimp)
  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 fastsimp
  moreover
  have ?cw ⊢ (size ?cb + size ?cc,s2,stk) →* (0,s2,stk)
    by (fastsimp)
  moreover
  have ?cw ⊢ (0,s2,stk) →* (size ?cw,s3,stk) by(rule WhileTrue(5))
  ultimately show ?case by(blast intro: exec-trans)
qed fastsimp+

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* $a_1$ *s* (*Iv* $i_1$) ⟹ *taval* $a_2$ *s* (*Iv* $i_2$)
⟹ *taval* (*Plus* $a_1$ $a_2$) *s* (*Iv*($i_1$+$i_2$)) |
*taval* $a_1$ *s* (*Rv* $r_1$) ⟹ *taval* $a_2$ *s* (*Rv* $r_2$)
⟹ *taval* (*Plus* $a_1$ $a_2$) *s* (*Rv*($r_1$+$r_2$))

**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* $b_1$ *s* $bv_1$ ⟹ *tbval* $b_2$ *s* $bv_2$ ⟹ *tbval* (*And* $b_1$ $b_2$) *s* ($bv_1$ & $bv_2$) |
*taval* $a_1$ *s* (*Iv* $i_1$) ⟹ *taval* $a_2$ *s* (*Iv* $i_2$) ⟹ *tbval* (*Less* $a_1$ $a_2$) *s* ($i_1 < i_2$) |
*taval* $a_1$ *s* (*Rv* $r_1$) ⟹ *taval* $a_2$ *s* (*Rv* $r_2$) ⟹ *tbval* (*Less* $a_1$ $a_2$) *s* ($r_1 <$ $r_2$)

## 6.3 Syntax of Commands

**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*)

## 6.4 Small-Step Semantics of Commands

**inductive**
*small-step* :: (*com* × *state*) ⇒ (*com* × *state*) ⇒ *bool* (**infix** → *55*)
**where**
*Assign*: *taval a s v* ⟹ (*x* ::= *a*, *s*) → (*SKIP*, *s*(*x* := *v*)) |

*Semi1*: (*SKIP*;*c*,*s*) → (*c*,*s*) |
*Semi2*: ($c_1$,*s*) → ($c_1'$,$s'$) ⟹ ($c_1$;$c_2$,*s*) → ($c_1'$;$c_2$,$s'$) |

*IfTrue*: *tbval b s True* ⟹ (*IF b THEN* $c_1$ *ELSE* $c_2$,*s*) → ($c_1$,*s*) |
*IfFalse*: *tbval b s False* ⟹ (*IF b THEN* $c_1$ *ELSE* $c_2$,*s*) → ($c_2$,*s*) |

*While*: (*WHILE b DO c*,*s*) → (*IF b THEN c*; *WHILE b DO c ELSE SKIP*,*s*)

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

## 6.5 The Type System

**datatype** *ty* = *Ity* | *Rty*

**types** *tyenv* = *name* ⇒ *ty*

**inductive** *atyping* :: *tyenv* ⇒ *aexp* ⇒ *ty* ⇒ *bool*
((*1*-/ ⊢/ (- :/ -)) [*50*,*0*,*50*] *50*)
**where**
*Ic-ty*: Γ ⊢ *Ic i* : *Ity* |
*Rc-ty*: Γ ⊢ *Rc r* : *Rty* |
*V-ty*: Γ ⊢ *V x* : Γ *x* |
*Plus-ty*: Γ ⊢ $a_1$ : τ ⟹ Γ ⊢ $a_2$ : τ ⟹ Γ ⊢ *Plus* $a_1$ $a_2$ : τ

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.

**inductive** *btyping* :: *tyenv* ⇒ *bexp* ⇒ *bool* (**infix** ⊢ *50*)
**where**
*B-ty*: Γ ⊢ *B bv* |
*Not-ty*: Γ ⊢ *b* ⟹ Γ ⊢ *Not b* |
*And-ty*: Γ ⊢ $b_1$ ⟹ Γ ⊢ $b_2$ ⟹ Γ ⊢ *And* $b_1$ $b_2$ |

*Less-ty*: $\Gamma \vdash a_1 : \tau \Longrightarrow \Gamma \vdash a_2 : \tau \Longrightarrow \Gamma \vdash \mathit{Less}\ a_1\ a_2$

**inductive** *ctyping* :: *tyenv* ⇒ *com* ⇒ *bool* (**infix** ⊢ *50*) **where**
*Skip-ty*: $\Gamma \vdash \mathit{SKIP}$ |
*Assign-ty*: $\Gamma \vdash a : \Gamma(x) \Longrightarrow \Gamma \vdash x ::= a$ |
*Semi-ty*: $\Gamma \vdash c_1 \Longrightarrow \Gamma \vdash c_2 \Longrightarrow \Gamma \vdash c_1;c_2$ |
*If-ty*: $\Gamma \vdash b \Longrightarrow \Gamma \vdash c_1 \Longrightarrow \Gamma \vdash c_2 \Longrightarrow \Gamma \vdash \mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2$ |
*While-ty*: $\Gamma \vdash b \Longrightarrow \Gamma \vdash c \Longrightarrow \Gamma \vdash \mathit{WHILE}\ b\ \mathit{DO}\ c$

**inductive-cases** [*elim*!]:
  $\Gamma \vdash x ::= a$  $\Gamma \vdash c1;c2$
  $\Gamma \vdash \mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2$
  $\Gamma \vdash \mathit{WHILE}\ b\ \mathit{DO}\ c$

## 6.6 Well-typed Programs Do Not Get Stuck

**fun** *type* :: *val* ⇒ *ty* **where**
*type* (*Iv i*) = *Ity* |
*type* (*Rv r*) = *Rty*

**lemma** [*simp*]: *type v* = *Ity* ⟷ (∃ *i*. *v* = *Iv i*)
**by** (*cases v*) *simp-all*

**lemma** [*simp*]: *type v* = *Rty* ⟷ (∃ *r*. *v* = *Rv r*)
**by** (*cases v*) *simp-all*

**definition** *styping* :: *tyenv* ⇒ *state* ⇒ *bool* (**infix** ⊢ *50*)
**where** $\Gamma \vdash s \longleftrightarrow (\forall x.\ \mathit{type}\ (s\ x) = \Gamma\ x)$

**lemma** *apreservation*:
  $\Gamma \vdash a : \tau \Longrightarrow \mathit{taval}\ a\ s\ v \Longrightarrow \Gamma \vdash s \Longrightarrow \mathit{type}\ v = \tau$
**apply**(*induct arbitrary*: *v rule*: *atyping.induct*)
**apply** (*fastsimp simp*: *styping-def*)+
**done**

**lemma** *aprogress*: $\Gamma \vdash a : \tau \Longrightarrow \Gamma \vdash s \Longrightarrow \exists v.\ \mathit{taval}\ a\ s\ v$
**proof**(*induct rule*: *atyping.induct*)
  **case** (*Plus-ty* Γ *a1 t a2*)
  **then obtain** *v1 v2* **where** *v*: *taval a1 s v1 taval a2 s v2* **by** *blast*
  **show** *?case*
  **proof** (*cases v1*)
    **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
qed (auto intro: taval.intros)

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!: tbval.intros(4) dest!:apreservation)
  next
    case Rv
    with Less-ty v show ?thesis
      by (fastsimp intro!: tbval.intros(5) dest!:apreservation)
  qed
qed (auto intro: tbval.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
```

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

**theorem** *styping-preservation*:
  $(c,s) \to (c',s') \Longrightarrow \Gamma \vdash c \Longrightarrow \Gamma \vdash s \Longrightarrow \Gamma \vdash s'$
**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*:
  $(c,s) \to (c',s') \Longrightarrow \Gamma \vdash c \Longrightarrow \Gamma \vdash c'$
**by** (*induct rule*: *small-step-induct*) (*auto simp*: *ctyping.intros*)

**inductive**
  *small-steps* :: *com* $*$ *state* $\Rightarrow$ *com* $*$ *state* $\Rightarrow$ *bool* (**infix** $\to*$ *55*) **where**
*refl*: $cs \to* cs$ |
*step*: $cs \to cs' \Longrightarrow cs' \to* cs'' \Longrightarrow cs \to* cs''$

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

**theorem** *type-sound*:
  $(c,s) \to* (c',s') \Longrightarrow \Gamma \vdash c \Longrightarrow \Gamma \vdash s \Longrightarrow c' \neq SKIP$
   $\Longrightarrow \exists\, cs''.\ (c',s') \to cs''$
**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* = *Ity* | *Rty* | *TV nat*

Everything else remains the same.

**types** *tyenv* = *name* $\Rightarrow$ *ty*

**inductive** *atyping* :: *tyenv* $\Rightarrow$ *aexp* $\Rightarrow$ *ty* $\Rightarrow$ *bool*
  ((*1*-/ $\vdash$*p*/ (- :/ -)) [*50*,*0*,*50*] *50*)
**where**
$\Gamma \vdash p$ *Ic i* : *Ity* |

$\Gamma \vdash p\ Rc\ r : Rty$ |
$\Gamma \vdash p\ V\ x : \Gamma\ x$ |
$\Gamma \vdash p\ a_1 : \tau \Longrightarrow \Gamma \vdash p\ a_2 : \tau \Longrightarrow \Gamma \vdash p\ Plus\ a_1\ a_2 : \tau$

**inductive** *btyping* :: *tyenv* ⇒ *bexp* ⇒ *bool* (**infix** ⊢*p* *50*)
**where**
$\Gamma \vdash p\ B\ bv$ |
$\Gamma \vdash p\ b \Longrightarrow \Gamma \vdash p\ Not\ b$ |
$\Gamma \vdash p\ b_1 \Longrightarrow \Gamma \vdash p\ b_2 \Longrightarrow \Gamma \vdash p\ And\ b_1\ b_2$ |
$\Gamma \vdash p\ a_1 : \tau \Longrightarrow \Gamma \vdash p\ a_2 : \tau \Longrightarrow \Gamma \vdash p\ Less\ a_1\ a_2$

**inductive** *ctyping* :: *tyenv* ⇒ *com* ⇒ *bool* (**infix** ⊢*p* *50*) **where**
$\Gamma \vdash p\ SKIP$ |
$\Gamma \vdash p\ a : \Gamma(x) \Longrightarrow \Gamma \vdash p\ x ::= a$ |
$\Gamma \vdash p\ c_1 \Longrightarrow \Gamma \vdash p\ c_2 \Longrightarrow \Gamma \vdash p\ c_1;c_2$ |
$\Gamma \vdash p\ b \Longrightarrow \Gamma \vdash p\ c_1 \Longrightarrow \Gamma \vdash p\ c_2 \Longrightarrow \Gamma \vdash p\ IF\ b\ THEN\ c_1\ ELSE\ c_2$ |
$\Gamma \vdash p\ b \Longrightarrow \Gamma \vdash p\ c \Longrightarrow \Gamma \vdash p\ WHILE\ b\ DO\ c$

**fun** *type* :: *val* ⇒ *ty* **where**
*type* (*Iv i*) = *Ity* |
*type* (*Rv r*) = *Rty*

**definition** *styping* :: *tyenv* ⇒ *state* ⇒ *bool* (**infix** ⊢*p* *50*)
**where** $\Gamma \vdash p\ s \longleftrightarrow (\forall x.\ type\ (s\ x) = \Gamma\ x)$

**fun** *tsubst* :: (*nat* ⇒ *ty*) ⇒ *ty* ⇒ *ty* **where**
*tsubst S* (*TV n*) = *S n* |
*tsubst S t* = *t*

## 6.8 Typing is Preserved by Substitution

**lemma** *subst-atyping*: $E \vdash p\ a : t \Longrightarrow tsubst\ S \circ E \vdash p\ a : tsubst\ S\ t$
**apply**(*induct rule*: *atyping.induct*)
**apply**(*auto intro*: *atyping.intros*)
**done**

**lemma** *subst-btyping*: $E \vdash p\ (b{::}bexp) \Longrightarrow tsubst\ S \circ E \vdash p\ b$
**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*: $E \vdash p\ (c{::}com) \Longrightarrow tsubst\ S \circ E \vdash p\ c$
**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*)
**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. *avars* and *bvars*, but we overload the name *vars* via a *type class*, a device that originated with Haskell:

**class** *vars* =
**fixes** $vars :: 'a \Rightarrow name\ set$

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 *aexp* and one for *bexp*:

**instantiation** *aexp* :: *vars*
**begin**

**fun** $vars\text{-}aexp :: aexp \Rightarrow name\ set$ **where**
$vars\text{-}aexp\ (N\ n) = \{\}\ |$
$vars\text{-}aexp\ (V\ x) = \{x\}\ |$
$vars\text{-}aexp\ (Plus\ a_1\ a_2) = vars\text{-}aexp\ a_1 \cup vars\text{-}aexp\ a_2$

**instance** ..

**end**

```
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

**inductive** $D$ :: *name set* ⇒ *com* ⇒ *name set* ⇒ *bool* **where**
*Skip*: $D\ A\ SKIP\ A$ |
*Assign*: $vars\ a \subseteq A \Longrightarrow D\ A\ (x ::= a)\ (insert\ x\ A)$ |
*Semi*: $[\![\ D\ A_1\ c_1\ A_2;\ \ D\ A_2\ c_2\ A_3\ ]\!] \Longrightarrow D\ A_1\ (c_1;\ c_2)\ A_3$ |
*If*: $[\![\ vars\ b \subseteq A;\ \ D\ A\ c_1\ A_1;\ \ D\ A\ c_2\ A_2\ ]\!] \Longrightarrow$
  $D\ A\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ (A_1\ Int\ A_2)$ |
*While*: $[\![\ vars\ b \subseteq A;\ \ D\ A\ c\ A'\ ]\!] \Longrightarrow D\ A\ (WHILE\ b\ DO\ c)\ A$

**inductive-cases** [*elim*!]:
$D\ A\ SKIP\ A'$
$D\ A\ (x ::= a)\ A'$
$D\ A\ (c1;c2)\ A'$
$D\ A\ (IF\ b\ THEN\ c1\ ELSE\ c2)\ A'$
$D\ A\ (WHILE\ b\ DO\ c)\ A'$

**lemma** *D-incr*:
  $D\ A\ c\ A' \Longrightarrow A \subseteq A'$
**by** (*induct rule*: *D.induct*) *auto*

**end**

**theory** *Def-Ass-Exp* **imports** *Vars*
**begin**

## 7.3 Initialization-Sensitive Expressions Evaluation

**types**
  *val* = *nat*
  *state* = *name* ⇒ *val option*

**fun** *aval* :: *aexp* ⇒ *state* ⇒ *val option* **where**
$aval\ (N\ i)\ s = Some\ i$ |
$aval\ (V\ x)\ s = s\ x$ |
$aval\ (Plus\ a_1\ a_2)\ s =$
  $(case\ (aval\ a_1\ s,\ aval\ a_2\ s)\ of$
    $(Some\ i_1, Some\ i_2) \Rightarrow Some(i_1+i_2)$ | - ⇒ *None*)

**fun** *bval* :: *bexp* ⇒ *state* ⇒ *bool option* **where**

*bval* (*B bv*) *s* = *Some bv* |
*bval* (*Not b*) *s* = (*case bval b s of None* ⇒ *None* | *Some bv* ⇒ *Some*(¬ *bv*))
|
*bval* (*And* $b_1$ $b_2$) *s* = (*case* (*bval* $b_1$ *s*, *bval* $b_2$ *s*) *of*
(*Some* $bv_1$, *Some* $bv_2$) ⇒ *Some*($bv_1$ & $bv_2$) | - ⇒ *None*) |
*bval* (*Less* $a_1$ $a_2$) *s* = (*case* (*aval* $a_1$ *s*, *aval* $a_2$ *s*) *of*
(*Some* $i_1$, *Some* $i_2$) ⇒ *Some*($i_1$ < $i_2$) | - ⇒ *None*)

**lemma** *aval-Some*: *vars a* ⊆ *dom s* ⟹ ∃ *i*. *aval a s* = *Some i*
**by** (*induct a*) *auto*

**lemma** *bval-Some*: *vars b* ⊆ *dom s* ⟹ ∃ *bv*. *bval b s* = *Some 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**
*big-step* :: (*com* × *state option*) ⇒ *state option* ⇒ *bool* (**infix** ⇒ *55*)
**where**
*None*: (*c*,*None*) ⇒ *None* |
*Skip*: (*SKIP*,*s*) ⇒ *s* |
*AssignNone*: *aval a s* = *None* ⟹ (*x* ::= *a*, *Some s*) ⇒ *None* |
*Assign*: *aval a s* = *Some i* ⟹ (*x* ::= *a*, *Some s*) ⇒ *Some*(*s*(*x* := *Some i*))
|
*Semi*: ($c_1$,$s_1$) ⇒ $s_2$ ⟹ ($c_2$,$s_2$) ⇒ $s_3$ ⟹ ($c_1$;$c_2$,$s_1$) ⇒ $s_3$ |

*IfNone*: *bval b s* = *None* ⟹ (*IF b THEN* $c_1$ *ELSE* $c_2$,*Some s*) ⇒ *None* |
*IfTrue*: ⟦ *bval b s* = *Some True*; ($c_1$,*Some s*) ⇒ *s'* ⟧ ⟹
(*IF b THEN* $c_1$ *ELSE* $c_2$,*Some s*) ⇒ *s'* |
*IfFalse*: ⟦ *bval b s* = *Some False*; ($c_2$,*Some s*) ⇒ *s'* ⟧ ⟹
(*IF b THEN* $c_1$ *ELSE* $c_2$,*Some s*) ⇒ *s'* |

*WhileNone*: *bval b s* = *None* ⟹ (*WHILE b DO c*,*Some s*) ⇒ *None* |
*WhileFalse*: *bval b s* = *Some False* ⟹ (*WHILE b DO c*,*Some s*) ⇒ *Some*
*s* |
*WhileTrue*:

$\llbracket$ *bval b s = Some True*; (*c*,*Some s*) $\Rightarrow$ *s'*; (*WHILE b DO c*,*s'*) $\Rightarrow$ *s''* $\rrbracket$
$\Longrightarrow$
(*WHILE b DO c*,*Some s*) $\Rightarrow$ *s''*

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

**end**

**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*:
$\llbracket$ (*c*,*Some s*) $\Rightarrow$ *s'*; *D A c A'*; *A* $\subseteq$ *dom s* $\rrbracket$
$\Longrightarrow$ $\exists$ *t*. *s'* = *Some t* $\wedge$ *A'* $\subseteq$ *dom t*
**proof** (*induct c Some s s' arbitrary*: *s A A' rule*:*big-step-induct*)
  **case** *AssignNone* **thus** *?case*
    **by** *auto* (*metis aval-Some option.simps*(*3*) *subset-trans*)
**next**
  **case** *Semi* **thus** *?case* **by** *auto metis*
**next**
  **case** *IfTrue* **thus** *?case* **by** *auto blast*
**next**
  **case** *IfFalse* **thus** *?case* **by** *auto blast*
**next**
  **case** *IfNone* **thus** *?case*
    **by** *auto* (*metis bval-Some option.simps*(*3*) *order-trans*)
**next**
  **case** *WhileNone* **thus** *?case*
    **by** *auto* (*metis bval-Some option.simps*(*3*) *order-trans*)
**next**
  **case** (*WhileTrue b s c s' s''*)
  **from** ‹*D A* (*WHILE b DO c*) *A'*› **obtain** *A'* **where** *D A c A'* **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* (*dom s*) *c A'*; (*c*,*Some s*) $\Rightarrow$ *s'* $\rrbracket$ $\Longrightarrow$ *s'* $\neq$ *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** $L :: com \Rightarrow name\ set \Rightarrow name\ set$ **where**
$L\ SKIP\ X = X\ |$
$L\ (x ::= a)\ X = X - \{x\} \cup vars\ a\ |$
$L\ (c_1;\ c_2)\ X = (L\ c_1 \circ L\ c_2)\ X\ |$
$L\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ X = vars\ b \cup L\ c_1\ X \cup L\ c_2\ X\ |$
$L\ (WHILE\ b\ DO\ c)\ X = vars\ b \cup X \cup L\ c\ X$

**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** $kill :: com \Rightarrow name\ set$ **where**
$kill\ SKIP = \{\}\ |$
$kill\ (x ::= a) = \{x\}\ |$
$kill\ (c_1;\ c_2) = kill\ c_1 \cup kill\ c_2\ |$
$kill\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) = kill\ c_1 \cap kill\ c_2\ |$
$kill\ (WHILE\ b\ DO\ c) = \{\}$

**fun** $gen :: com \Rightarrow name\ set$ **where**
$gen\ SKIP = \{\}\ |$
$gen\ (x ::= a) = vars\ a\ |$
$gen\ (c_1;\ c_2) = gen\ c_1 \cup (gen\ c_2 - kill\ c_1)\ |$
$gen\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) = vars\ b \cup gen\ c_1 \cup gen\ c_2\ |$
$gen\ (WHILE\ b\ DO\ c) = vars\ b \cup gen\ c$

**lemma** *L-gen-kill*: $L\ c\ X = (X - kill\ c) \cup gen\ c$
**by**(*induct c arbitrary*:*X*) *auto*

**lemma** *L-While-subset*: $L\ c\ (L\ (WHILE\ b\ DO\ c)\ X) \subseteq L\ (WHILE\ b\ DO\ c)\ X$
**by**(*auto simp add*:*L-gen-kill*)

## 8.2 Soundness

```
theorem L-sound:
  (c,s) ⇒ s' ⟹ s = t on L c X ⟹
  ∃ t'. (c,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: (c1, t1) ⇒ t2 and s2t2: s2 = t2 on L c2 X
    by simp blast
  from Semi(4)[OF s2t2] obtain t3 where
    t23: (c2, 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
    (c1, t) ⇒ t' s' = t' on X by auto
  thus ?case using ‹bval b t› by auto
next
  case (IfFalse b s c2 s' c1)
  hence s = t on vars b s = t on L c2 X by auto
  from  bval-eq-if-eq-on-vars[OF this(1)] IfFalse(1) have ~bval b t by simp
  from IfFalse(3)[OF ‹s = t on L c2 X›] obtain t' where
    (c2, t) ⇒ t' s' = t' on X by auto
  thus ?case using ‹~bval b t› by auto
next
  case (WhileFalse b s c)
  hence ~ bval b t by (auto 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 = WHILE b 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* $(c_1;\ c_2)$ *X = (bury* $c_1$ *(L* $c_2$ *X); bury* $c_2$ *X)* |
*bury (IF b THEN* $c_1$ *ELSE* $c_2$*) X = IF b THEN bury* $c_1$ *X ELSE bury* $c_2$ *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*

**thus** *?case* **using** ⟨*bval b t*⟩ **by** *auto*
**next**
**case** (*IfFalse b s c2 s′ c1*)
**hence** *s = t on vars b s = t on L c2 X* **by** *auto*
**from** *bval-eq-if-eq-on-vars*[*OF this*(*1*)] *IfFalse*(*1*) **have** ~*bval b t* **by** *simp*
**from** *IfFalse*(*3*)[*OF* ⟨*s = t on L c2 X*⟩] **obtain** *t′* **where**
(*bury c2 X*, *t*) ⇒ *t′ s′ = t′ on X* **by** *auto*
**thus** *?case* **using** ⟨~*bval b t*⟩ **by** *auto*
**next**
**case** (*WhileFalse b s c*)
**hence** ~ *bval b t* **by** (*auto 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 = WHILE b 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**
(*bury c* (*L ?w X*), *t1*) ⇒ *t2 s2 = t2 on L ?w X* **by** *auto*
**from** *WhileTrue*(*5*)[*OF this*(*2*)] **obtain** *t3*
**where** (*bury ?w X*,*t2*) ⇒ *t3 s3 = t3 on X*
**by** *auto*
**with** ⟨*bval b t1*⟩ ⟨(*bury c* (*L ?w X*), *t1*) ⇒ *t2*⟩ **show** *?case* **by** *auto*

**qed**

**corollary** *final-bury-sound*: (*c*,*s*) ⇒ *s′* ⟹ (*bury c UNIV*,*s*) ⇒ *s′*
**using** *bury-sound*[*of c s s′ UNIV*]
**by** (*auto simp*: *expand-fun-eq*[*symmetric*])

Now the opposite direction.

**lemma** *SKIP-bury*[*simp*]:
*SKIP = bury c X* ⟷ *c = SKIP* | (*EX x a. c = x::=a & x* ∉ *X*)
**by** (*cases c*) *auto*

**lemma** *Assign-bury*[*simp*]: *x::=a = bury c X* ⟷ *c = x::=a & x : X*
**by** (*cases c*) *auto*

**lemma** *Semi-bury*[*simp*]: $bc_1;bc_2$ = *bury c X* ⟷
(*EX* $c_1$ $c_2$. *c* = $c_1;c_2$ & $bc_2$ = *bury* $c_2$ *X* & $bc_1$ = *bury* $c_1$ (*L* $c_2$ *X*))
**by** (*cases c*) *auto*

```
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)
  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
```

**thus** *?case* **using** *c* ‹~*bval b t*› **by** *auto*
**next**
**case** (*WhileFalse b s c*)
**hence** ~ *bval b t* **by** (*auto simp*: *ball-Un dest*: *bval-eq-if-eq-on-vars*)
**thus** *?case* **using** *WhileFalse* **by** *auto*
**next**
**case** (*WhileTrue b s1 bc′ s2 s3 c X t1*)
**then obtain** *c′* **where** *c*: *c* = *WHILE b DO c′*
**and** *bc′*: *bc′* = *bury c′* (*vars b* ∪ *X* ∪ *L c′ X*) **by** *auto*
**let** *?w* = *WHILE b DO c′*
**from** ‹*bval b s1*› *WhileTrue.prems c* **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 c* **by** *blast*
**with** *WhileTrue*(*3*)[*OF bc′*, *of t1*] **obtain** *t2* **where**
(*c′*, *t1*) ⇒ *t2 s2* = *t2 on L ?w X* **by** *auto*
**from** *WhileTrue*(*5*)[*OF WhileTrue*(*6*), *of t2*] *c this*(*2*) **obtain** *t3*
**where** (*?w*,*t2*) ⇒ *t3 s3* = *t3 on X*
**by** *auto*
**with** ‹*bval b t1*› ‹(*c′*, *t1*) ⇒ *t2*› *c* **show** *?case* **by** *auto*
**qed**

**corollary** *final-bury-sound2*: (*bury c UNIV*,*s*) ⇒ *s′* ⟹ (*c*,*s*) ⇒ *s′*
**using** *bury-sound2*[*of c UNIV*]
**by** (*auto simp*: *expand-fun-eq*[*symmetric*])

**corollary** *bury-iff*: (*bury c UNIV*,*s*) ⇒ *s′* ⟷ (*c*,*s*) ⇒ *s′*
**by**(*metis final-bury-sound final-bury-sound2*)

**end**

# 9 Security Type Systems

**theory** *Sec-Type-Expr* **imports** *Big-Step*
**begin**

## 9.1 Security Levels and Expressions

**types** *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 $n$ has security level $n$:

**class** *sec* = **fixes** *sec* :: $'a \Rightarrow level$

**instantiation** *nat* :: *sec*
**begin**

**definition** *sec-nat* :: $name \Rightarrow level$ **where** *sec n = n*

**instance** ..

**end**

**instantiation** *aexp* :: *sec*
**begin**

**fun** *sec-aexp* :: $aexp \Rightarrow level$ **where**
*sec-aexp (N n) = 0* |
*sec-aexp (V x) = sec x* |
*sec-aexp* (*Plus* $a_1$ $a_2$) = *max* (*sec-aexp* $a_1$) (*sec-aexp* $a_2$)

**instance** ..

**end**

**instantiation** *bexp* :: *sec*
**begin**

**fun** *sec-bexp* :: $bexp \Rightarrow level$ **where**
*sec-bexp (B bv) = 0* |
*sec-bexp (Not b) = sec-bexp b* |
*sec-bexp* (*And* $b_1$ $b_2$) = *max* (*sec-bexp* $b_1$) (*sec-bexp* $b_2$) |
*sec-bexp* (*Less* $a_1$ $a_2$) = *max* (*sec* $a_1$) (*sec* $a_2$)

**instance** ..

**end**

**abbreviation** *eq-le* :: $state \Rightarrow state \Rightarrow level \Rightarrow bool$
$((\text{-} = \text{-}\ '(\leq \text{-}'))\ [51,51,0]\ 50)$ **where**
$s = s'\ (\leq l) == (\forall\ x.\ sec\ x \leq l \longrightarrow s\ x = s'\ x)$

**abbreviation** *eq-less* :: $state \Rightarrow state \Rightarrow level \Rightarrow bool$
$((\text{-} = \text{-}\ '(< \text{-}'))\ [51,51,0]\ 50)$ **where**
$s = s'\ (< l) == (\forall\ x.\ sec\ x < l \longrightarrow s\ x = s'\ x)$

**lemma** *aval-eq-if-eq-le*:
  $[\![ s_1 = s_2 \ (\leq l); \ \ sec\ a \leq l \ ]\!] \Longrightarrow aval\ a\ s_1 = aval\ a\ s_2$
**by** (*induct a*) *auto*

**lemma** *bval-eq-if-eq-le*:
  $[\![ s_1 = s_2 \ (\leq l); \ \ sec\ b \leq l \ ]\!] \Longrightarrow bval\ b\ s_1 = bval\ b\ s_2$
**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* :: $nat \Rightarrow com \Rightarrow bool$ ((-/ ⊢ -) [*0*,*0*] *50*) **where**
*Skip*:
  $l \vdash SKIP$ |
*Assign*:
  $[\![ sec\ x \geq sec\ a; \ \ sec\ x \geq l \ ]\!] \Longrightarrow l \vdash x ::= a$ |
*Semi*:
  $[\![ l \vdash c_1; \ \ l \vdash c_2 \ ]\!] \Longrightarrow l \vdash c_1;c_2$ |
*If*:
  $[\![ max\ (sec\ b)\ l \vdash c_1; \ \ max\ (sec\ b)\ l \vdash c_2 \ ]\!] \Longrightarrow l \vdash IF\ b\ THEN\ c_1\ ELSE$
$c_2$ |
*While*:
  $max\ (sec\ b)\ l \vdash c \Longrightarrow l \vdash WHILE\ b\ DO\ c$

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

**value** *0* ⊢ *IF Less* (*V 1*) (*V 0*) *THEN 1* ::= *N 0 ELSE SKIP*
**value** *1* ⊢ *IF Less* (*V 1*) (*V 0*) *THEN 1* ::= *N 0 ELSE SKIP*
**value** *2* ⊢ *IF Less* (*V 1*) (*V 0*) *THEN 1* ::= *N 0 ELSE SKIP*

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

An important property: anti-monotonicity.

**lemma** *anti-mono*: $[\![ l \vdash c; \ \ l' \leq l \ ]\!] \Longrightarrow l' \vdash c$
**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*)
**apply** (*metis While le-refl sup-mono sup-nat-def*)
**done**

**lemma** *confinement*: ⟦ (*c*,*s*) ⇒ *t*;  *l* ⊢ *c* ⟧ ⟹ *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* ⊢ *c1* **by** *auto*
  **hence** *l* ⊢ *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** *max* (*sec b*) *l* ⊢ *c* **by** *auto*
  **hence** *l* ⊢ *c* **by** (*metis le-maxI2 anti-mono*)
  **thus** *?case* **using** *WhileTrue* **by** *metis*
**qed**

**theorem** *noninterference*:
  ⟦ (*c*,*s*) ⇒ *s'*; (*c*,*t*) ⇒ *t'*;  *0* ⊢ *c*;  *s* = *t* (≤ *l*) ⟧
   ⟹ *s'* = *t'* (≤ *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* := *aval a t*) **using** *Assign* **by** *auto*
  **have** *sec x* >= *sec a* **using** ⟨*0* ⊢ *x* ::= *a*⟩ **by** *auto*
  **show** *?case*
  **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
next
  case Semi thus ?case by blast
next
  case (IfTrue b s c1 s′ c2)
  have sec b ⊢ c1 sec b ⊢ c2 using IfTrue.prems(2) by auto
  show ?case
  proof cases
    assume sec b ≤ l
    hence s = t (≤ sec b) using ‹s = t (≤ l)› by auto
    hence bval b t using ‹bval b s› by(simp add: bval-eq-if-eq-le)
    with IfTrue.hyps(3) IfTrue.prems(1,3) ‹sec b ⊢ c1› anti-mono
    show ?thesis by auto
  next
    assume ¬ sec b ≤ l
    have 1: sec b ⊢ IF b THEN c1 ELSE c2
      by(rule sec-type.intros)(simp-all add: ‹sec b ⊢ c1› ‹sec b ⊢ c2›)
    from confinement[OF big-step.IfTrue[OF IfTrue(1,2)] 1] ‹¬ sec b ≤ l›
    have s = s′ (≤ l) by auto
    moreover
    from confinement[OF IfTrue.prems(1) 1] ‹¬ sec b ≤ l›
    have t = t′ (≤ l) by auto
    ultimately show s′ = t′ (≤ l) using ‹s = t (≤ l)› by auto
  qed
next
  case (IfFalse b s c2 s′ c1)
  have sec b ⊢ c1 sec b ⊢ c2 using IfFalse.prems(2) by auto
  show ?case
  proof cases
    assume sec b ≤ l
    hence s = t (≤ sec b) using ‹s = t (≤ l)› by auto
    hence ¬ bval b t using ‹¬ bval b s› by(simp add: bval-eq-if-eq-le)
    with IfFalse.hyps(3) IfFalse.prems(1,3) ‹sec b ⊢ c2› anti-mono
    show ?thesis by auto
  next
    assume ¬ sec b ≤ l
    have 1: sec b ⊢ IF b THEN c1 ELSE c2
      by(rule sec-type.intros)(simp-all add: ‹sec b ⊢ c1› ‹sec b ⊢ c2›)
```

```
    from confinement[OF big-step.IfFalse[OF IfFalse(1,2)] 1] ‹¬ sec b ≤ l›
    have s = s' (≤ l) by auto
    moreover
    from confinement[OF IfFalse.prems(1) 1] ‹¬ sec b ≤ l›
    have t = t' (≤ l) by auto
    ultimately show s' = t' (≤ l) using ‹s = t (≤ l)› by auto
  qed
next
  case (WhileFalse b s c)
  have sec b ⊢ c using WhileFalse.prems(2) by auto
  show ?case
  proof cases
    assume sec b ≤ l
    hence s = t (≤ sec b) using ‹s = t (≤ l)› by auto
    hence ¬ bval b t using ‹¬ bval b s› by(simp add: bval-eq-if-eq-le)
    with WhileFalse.prems(1,3) show ?thesis by auto
  next
    assume ¬ sec b ≤ l
    have 1: sec b ⊢ WHILE b DO c
      by(rule sec-type.intros)(simp-all add: ‹sec b ⊢ c›)
    from confinement[OF WhileFalse.prems(1) 1] ‹¬ sec b ≤ l›
    have t = t' (≤ l) by auto
    thus s = t' (≤ l) using ‹s = t (≤ l)› by auto
  qed
next
  case (WhileTrue b s1 c s2 s3 t1 t3)
  let ?w = WHILE b DO c
  have sec b ⊢ c using WhileTrue.prems(2) by auto
  show ?case
  proof cases
    assume sec b ≤ l
    hence s1 = t1 (≤ sec b) using ‹s1 = t1 (≤ l)› by auto
    hence bval b t1
      using ‹bval b s1› by(simp add: bval-eq-if-eq-le)
    then obtain t2 where (c,t1) ⇒ t2 (?w,t2) ⇒ t3
      using ‹(?w,t1) ⇒ t3› by auto
    from WhileTrue.hyps(5)[OF ‹(?w,t2) ⇒ t3› ‹0 ⊢ ?w›
      WhileTrue.hyps(3)[OF ‹(c,t1) ⇒ t2› anti-mono[OF ‹sec b ⊢ c›]
        ‹s1 = t1 (≤ l)›]]
    show ?thesis by simp
  next
    assume ¬ sec b ≤ l
    have 1: sec b ⊢ ?w by(rule sec-type.intros)(simp-all add: ‹sec b ⊢ c›)
    from confinement[OF big-step.WhileTrue[OF WhileTrue(1,2,4)] 1] ‹¬
```

*sec b* ≤ *l*›
  **have** *s1* = *s3* (≤ *l*) **by** *auto*
  **moreover**
  **from** *confinement*[*OF WhileTrue.prems*(*1*) *1*] ‹¬ *sec b* ≤ *l*›
  **have** *t1* = *t3* (≤ *l*) **by** *auto*
  **ultimately show** *s3* = *t3* (≤ *l*) **using** ‹*s1* = *t1* (≤ *l*)› **by** *auto*
 **qed**
**qed**

### 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.

**inductive** *sec-type′* :: *nat* ⇒ *com* ⇒ *bool* ((-/ ⊢″ -) [*0*,*0*] *50*) **where**
*Skip′*:
 $l \vdash' SKIP$ |
*Assign′*:
 $[\![\ sec\ x \geq sec\ a;\ sec\ x \geq l\ ]\!] \Longrightarrow l \vdash' x ::= a$ |
*Semi′*:
 $[\![\ l \vdash' c_1;\ \ l \vdash' c_2\ ]\!] \Longrightarrow l \vdash' c_1;c_2$ |
*If′*:
 $[\![\ sec\ b \leq l;\ \ l \vdash' c_1;\ \ l \vdash' c_2\ ]\!] \Longrightarrow l \vdash' IF\ b\ THEN\ c_1\ ELSE\ c_2$ |
*While′*:
 $[\![\ sec\ b \leq l;\ \ l \vdash' c\ ]\!] \Longrightarrow l \vdash' WHILE\ b\ DO\ c$ |
*anti-mono′*:
 $[\![\ l \vdash' c;\ \ l' \leq l\ ]\!] \Longrightarrow l' \vdash' c$

**lemma** *sec-type-sec-type′*: $l \vdash c \Longrightarrow l \vdash' c$
**apply**(*induct rule*: *sec-type.induct*)
**apply** (*metis Skip′*)
**apply** (*metis Assign′*)
**apply** (*metis Semi′*)
**apply** (*metis min-max.inf-sup-ord*(*3*) *min-max.sup-absorb2 nat-le-linear If′ anti-mono′*)
**by** (*metis less-or-eq-imp-le min-max.sup-absorb1 min-max.sup-absorb2 nat-le-linear While′ anti-mono′*)

**lemma** *sec-type′-sec-type*: $l \vdash' c \Longrightarrow l \vdash c$
**apply**(*induct rule*: *sec-type′.induct*)
**apply** (*metis Skip*)
**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*:
⟦ ⊢ $c_1$ : $l_1$; ⊢ $c_2$ : $l_2$ ⟧ ⟹ ⊢ $c_1$;$c_2$ : *min* $l_1$ $l_2$ |
*If2*:
⟦ *sec b* ≤ *min* $l_1$ $l_2$; ⊢ $c_1$ : $l_1$; ⊢ $c_2$ : $l_2$ ⟧
⟹ ⊢ *IF b THEN* $c_1$ *ELSE* $c_2$ : *min* $l_1$ $l_2$ |
*While2*:
⟦ *sec b* ≤ *l*; ⊢ *c* : *l* ⟧ ⟹ ⊢ *WHILE b DO c* : *l*

**lemma** *sec-type2-sec-type'*: ⊢ *c* : *l* ⟹ *l* ⊢' *c*
**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*:
  ⟦ *sec x* ≥ *sec a*; *sec x* ≥ *l* ⟧ ⟹ *l* ⊢ *x* ::= *a* |
*Semi*:
  *l* ⊢ $c_1$ ⟹ *l* ⊢ $c_2$ ⟹ *l* ⊢ $c_1$;$c_2$ |
*If*:
  ⟦ *max* (*sec b*) *l* ⊢ $c_1$; *max* (*sec b*) *l* ⊢ $c_2$ ⟧
  ⟹ *l* ⊢ *IF b THEN* $c_1$ *ELSE* $c_2$ |
*While*:
  *sec b* = *0* ⟹ *0* ⊢ *c* ⟹ *0* ⊢ *WHILE b DO c*

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

**inductive-cases** [*elim*!]:
  *l* ⊢ *x* ::= *a*  *l* ⊢ $c_1$;$c_2$  *l* ⊢ *IF b THEN* $c_1$ *ELSE* $c_2$  *l* ⊢ *WHILE b DO c*

**lemma** *anti-mono*: *l* ⊢ *c* ⟹ *l'* ≤ *l* ⟹ *l'* ⊢ *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*) ⇒ *t* ⟹ *l* ⊢ *c* ⟹ *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* ⊢ *c1* **by** *auto*
  **hence** *l* ⊢ *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** *sec b* ⊢ *c1 sec b* ⊢ *c2* **using** *IfTrue.prems* **by** *auto*
**obtain** *t′* **where** *t′*: (*c1*, *t*) ⇒ *t′ s′* = *t′* (≤ *l*)
**using** *IfTrue*(*3*)[*OF anti-mono*[*OF* ‹*sec b* ⊢ *c1*›] *IfTrue.prems*(*2*)] **by** *blast*
**show** *?case*
**proof** *cases*
**assume** *sec b* ≤ *l*
**hence** *s* = *t* (≤ *sec b*) **using** ‹*s* = *t* (≤ *l*)› **by** *auto*
**hence** *bval b t* **using** ‹*bval b s*› **by**(*simp add*: *bval-eq-if-eq-le*)
**thus** *?thesis* **by** (*metis t′ big-step.IfTrue*)
**next**
**assume** ¬ *sec b* ≤ *l*
**hence** *0*: *sec b* ≠ *0* **by** *arith*
**have** *1*: *sec b* ⊢ *IF b THEN c1 ELSE c2*
**by**(*rule sec-type.intros*)(*simp-all add*: ‹*sec b* ⊢ *c1*› ‹*sec b* ⊢ *c2*›)
**from** *confinement*[*OF big-step.IfTrue*[*OF IfTrue*(*1*,*2*)] *1*] ‹¬ *sec b* ≤ *l*›
**have** *s* = *s′* (≤ *l*) **by** *auto*
**moreover**
**from** *termi-if-non0*[*OF 1 0*, *of t*] **obtain** *t′* **where**
(*IF b THEN c1 ELSE c2*,*t*) ⇒ *t′* **..**
**moreover**
**from** *confinement*[*OF this 1*] ‹¬ *sec b* ≤ *l*›
**have** *t* = *t′* (≤ *l*) **by** *auto*
**ultimately**
**show** *?case* **using** ‹*s* = *t* (≤ *l*)› **by** *auto*
**qed**
**next**
**case** (*IfFalse b s c2 s′ c1*)
**have** *sec b* ⊢ *c1 sec b* ⊢ *c2* **using** *IfFalse.prems* **by** *auto*
**obtain** *t′* **where** *t′*: (*c2*, *t*) ⇒ *t′ s′* = *t′* (≤ *l*)
**using** *IfFalse*(*3*)[*OF anti-mono*[*OF* ‹*sec b* ⊢ *c2*›] *IfFalse.prems*(*2*)] **by** *blast*
**show** *?case*
**proof** *cases*
**assume** *sec b* ≤ *l*
**hence** *s* = *t* (≤ *sec b*) **using** ‹*s* = *t* (≤ *l*)› **by** *auto*
**hence** ¬ *bval b t* **using** ‹¬ *bval b s*› **by**(*simp add*: *bval-eq-if-eq-le*)
**thus** *?thesis* **by** (*metis t′ big-step.IfFalse*)
**next**
**assume** ¬ *sec b* ≤ *l*
**hence** *0*: *sec b* ≠ *0* **by** *arith*
**have** *1*: *sec b* ⊢ *IF b THEN c1 ELSE c2*
**by**(*rule sec-type.intros*)(*simp-all add*: ‹*sec b* ⊢ *c1*› ‹*sec b* ⊢ *c2*›)
**from** *confinement*[*OF big-step.IfFalse*[*OF IfFalse*(*1*,*2*)] *1*] ‹¬ *sec b* ≤ *l*›

```
    have s = s' (≤ l) by auto
    moreover
    from termi-if-non0[OF 1 0, of t] obtain t' where
      (IF b THEN c1 ELSE c2,t) ⇒ t' ..
    moreover
    from confinement[OF this 1] ‹¬ sec b ≤ l›
    have t = t' (≤ l) by auto
    ultimately
    show ?case using ‹s = t (≤ l)› by auto
  qed
next
  case (WhileFalse b s c)
  hence [simp]: sec b = 0 by auto
  have s = t (≤ sec b) using ‹s = t (≤ l)› by auto
  hence ¬ bval b t using ‹¬ bval b s› by (metis bval-eq-if-eq-le le-refl)
  with WhileFalse.prems(2) show ?case by auto
next
  case (WhileTrue b s c s'' s')
  let ?w = WHILE b DO c
  from ‹0 ⊢ ?w› have [simp]: sec b = 0 by auto
  have 0 ⊢ c using WhileTrue.prems(1) by auto
  from WhileTrue(3)[OF this WhileTrue.prems(2)]
  obtain t'' where (c,t) ⇒ t'' and s'' = t'' (≤l) by blast
  from WhileTrue(5)[OF ‹0 ⊢ ?w› this(2)]
  obtain t' where (?w,t'') ⇒ t' and s' = t' (≤l) by blast
  from ‹bval b s› have bval b t
    using bval-eq-if-eq-le[OF ‹s = t (≤l)›] by auto
  show ?case
    using big-step.WhileTrue[OF ‹bval b t› ‹(c,t) ⇒ t''› ‹(?w,t'') ⇒ t'›]
    by (metis ‹s' = t' (≤ l)›)
qed
```

## 9.6 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 antimonotonicity rule. We introduce the standard system now and show the equivalence with our formulation.

```
inductive sec-type' :: nat ⇒ com ⇒ bool ((-/ ⊢'' -) [0,0] 50) where
Skip':
  l ⊢' SKIP  |
Assign':
  ⟦ sec x ≥ sec a;  sec x ≥ l ⟧ ⟹ l ⊢' x ::= a  |
Semi':
```

$l \vdash' c_1 \Longrightarrow l \vdash' c_2 \Longrightarrow l \vdash' c_1;c_2$ |
*If'*:
$[\![$ *sec* $b \le l$; $l \vdash' c_1$; $l \vdash' c_2$ $]\!] \Longrightarrow l \vdash'$ *IF* $b$ *THEN* $c_1$ *ELSE* $c_2$ |
*While'*:
$[\![$ *sec* $b = 0$; $0 \vdash' c$ $]\!] \Longrightarrow 0 \vdash'$ *WHILE* $b$ *DO* $c$ |
*anti-mono'*:
$[\![$ $l \vdash' c$; $l' \le l$ $]\!] \Longrightarrow l' \vdash' c$

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

**lemma** $l \vdash' c \Longrightarrow l \vdash c$
**apply**(*induct rule*: *sec-type'.induct*)
**apply** (*metis Skip*)
**apply** (*metis Assign*)
**apply** (*metis Semi*)
**apply** (*metis min-max.sup-absorb2 If*)
**apply** (*metis While*)
**by** (*metis anti-mono*)

**end**

# 10 Hoare Logic

**theory** *Hoare* **imports** *Big-Step* **begin**

## 10.1 Hoare Logic for Partial Correctness

**types** *assn* = *state* ⇒ *bool*

**abbreviation** *state-subst* :: *state* ⇒ *aexp* ⇒ *name* ⇒ *state*
(-[-'/-] [*1000*,*0*,*0*] *999*)
**where** $s[a/x] == s(x := aval\ a\ s)$

**inductive**
*hoare* :: *assn* ⇒ *com* ⇒ *assn* ⇒ *bool* (⊢ ({(*1*-)}/ (-)/ {(*1*-)}) *50*)

**where**
*Skip*: $\vdash \{P\}\ SKIP\ \{P\}$ |

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

*Semi*: $[\![\ \vdash \{P\}\ c_1\ \{Q\};\ \vdash \{Q\}\ c_2\ \{R\}\ ]\!]$
$\Longrightarrow \vdash \{P\}\ c_1;c_2\ \{R\}$ |

*If*: $[\![\ \vdash \{\lambda s.\ P\ s \wedge bval\ b\ s\}\ c_1\ \{Q\};\ \vdash \{\lambda s.\ P\ s \wedge \neg\ bval\ b\ s\}\ c_2\ \{Q\}\ ]\!]$
$\Longrightarrow \vdash \{P\}\ IF\ b\ THEN\ c_1\ ELSE\ c_2\ \{Q\}$ |

*While*: $\vdash \{\lambda s.\ P\ s \wedge bval\ b\ s\}\ c\ \{P\} \Longrightarrow$
$\vdash \{P\}\ WHILE\ b\ DO\ c\ \{\lambda s.\ P\ s \wedge \neg\ bval\ b\ s\}$ |

*conseq*: $[\![\ \forall s.\ P'\ s \longrightarrow P\ s;\ \vdash \{P\}\ c\ \{Q\};\ \forall s.\ Q\ s \longrightarrow Q'\ s\ ]\!]$
$\Longrightarrow \vdash \{P'\}\ c\ \{Q'\}$

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

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

**lemma** *strengthen-pre*:
$[\![\ \forall s.\ P'\ s \longrightarrow P\ s;\ \vdash \{P\}\ c\ \{Q\}\ ]\!] \Longrightarrow \vdash \{P'\}\ c\ \{Q\}$
**by** (*blast intro*: *conseq*)

**lemma** *weaken-post*:
$[\![\ \vdash \{P\}\ c\ \{Q\};\ \forall s.\ Q\ s \longrightarrow Q'\ s\ ]\!] \Longrightarrow\ \vdash \{P\}\ c\ \{Q'\}$
**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'*: $\forall s.\ P\ s \longrightarrow Q(s[a/x]) \Longrightarrow \vdash \{P\}\ x ::= a\ \{Q\}$
**by** (*simp add*: *strengthen-pre*[*OF* - *Assign*])

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

**end**

**theory** *Hoare-Examples* **imports** *Hoare* **begin**

## 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** $w\ n$ ==
  *WHILE Less* (*V 1*) (*N n*)
  *DO* ( *1* ::= *Plus* (*V 1*) (*N 1*); *0* ::= *Plus* (*V 0*) (*V 1*) )

For this example we make use of some predefined functions. Function *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** *while-sum*:
  $(w\ n,\ s) \Rightarrow t \Longrightarrow t\ 0 = s\ 0 + \sum\ \{s\ 1 + 1\ ..\ n\}$
**apply**(*induct w n s t rule*: *big-step-induct*)
**apply**(*auto simp add*: *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** *sum-via-bigstep*:
**assumes** $(0 ::= N\ 0;\ 1 ::= N\ 0;\ w\ n,\ s) \Rightarrow t$
**shows** $t\ 0 = \sum\ \{1\ ..\ n\}$
**proof** –
  **from** *assms* **have** $(w\ n, s(0{:=}0, 1{:=}0)) \Rightarrow t$ **by** *auto*
  **from** *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** $\vdash \{\lambda s.\ True\}\ 0 ::= N\ 0;\ 1 ::= N\ 0;\ w\ n\ \{\lambda s.\ s\ 0 = \sum\ \{1\ ..\ n\}\}$
**apply**(*rule hoare.Semi*)
**prefer** *2*

**apply**(*rule While'*
  [**where** $P = \lambda s.\ s\ 0 = \sum\ \{1..s\ 1\} \wedge s\ 1 \leq n$])
**apply**(*rule Semi*)
**prefer** *2*
**apply**(*rule Assign*)
**apply**(*rule Assign'*)
**apply**(*fastsimp*)
**apply**(*fastsimp*)
**apply**(*rule Semi*)
**prefer** *2*
**apply**(*rule Assign*)
**apply**(*rule Assign'*)
**apply** *simp*
**done**

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**

**theory** *Hoare-Sound-Complete* **imports** *Hoare* **begin**

## 10.3 Soundness

**definition**
*hoare-valid* :: *assn* ⇒ *com* ⇒ *assn* ⇒ *bool* (⊨ {(*1*-)}/ (-)/ {(*1*-)} *50*) **where**
⊨ {*P*}*c*{*Q*} = (∀ *s t*. (*c*,*s*) ⇒ *t* ⟶ *P s* ⟶ *Q t*)

**lemma** *hoare-sound*: ⊢ {*P*}*c*{*Q*} ⟹ ⊨ {*P*}*c*{*Q*}
**proof**(*induct rule*: *hoare.induct*)
  **case** (*While P b c*)
  **{ fix** *s t*
    **have** (*WHILE b DO c*,*s*) ⇒ *t* ⟹ *P s* ⟶ *P t* ∧ ¬ *bval b t*
    **proof**(*induct WHILE b DO c s t rule*: *big-step-induct*)
      **case** *WhileFalse* **thus** *?case* **by** *blast*
    **next**
      **case** *WhileTrue* **thus** *?case*
        **using** *While*(*2*) **unfolding** *hoare-valid-def* **by** *blast*
    **qed**

}
**thus** *?case* **unfolding** *hoare-valid-def* **by** *blast*
**qed** (*auto simp*: *hoare-valid-def*)

## 10.4 Weakest Precondition

**definition** *wp* :: *com* ⇒ *assn* ⇒ *assn* **where**
$wp\ c\ Q = (\lambda s.\ \forall t.\ (c,s) \Rightarrow t \longrightarrow Q\ t)$

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

**lemma** *wp-Ass*[*simp*]: $wp\ (x{::=}a)\ Q = (\lambda s.\ Q(s[a/x]))$
**by** (*rule ext*) (*auto simp*: *wp-def*)

**lemma** *wp-Semi*[*simp*]: $wp\ (c_1;c_2)\ Q = wp\ c_1\ (wp\ c_2\ Q)$
**by** (*rule ext*) (*auto simp*: *wp-def*)

**lemma** *wp-If*[*simp*]:
$wp\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ Q =$
$(\lambda s.\ (bval\ b\ s \longrightarrow wp\ c_1\ Q\ s) \wedge (\neg\ bval\ b\ s \longrightarrow wp\ c_2\ Q\ s))$
**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 ?case
  proof(rule hoare.If)
    show ⊢ {λs. wp ?If Q s ∧ bval b s} c1 {Q}
    proof(rule strengthen-pre[OF - If(1)])
      show ∀ s. wp ?If Q s ∧ bval b s ⟶ wp c1 Q s by auto
    qed
    show ⊢ {λs. wp ?If Q s ∧ ¬ bval b s} c2 {Q}
    proof(rule strengthen-pre[OF - If(2)])
      show ∀ s. wp ?If Q s ∧ ¬ bval b s ⟶ wp c2 Q s by auto
    qed
  qed
next
  case (While b c)
  let ?w = WHILE b DO c
  have ⊢ {wp ?w Q} ?w {λs. wp ?w Q s ∧ ¬ bval b s}
  proof(rule hoare.While)
    show ⊢ {λs. wp ?w Q s ∧ bval b s} c {wp ?w Q}
    proof(rule strengthen-pre[OF - While(1)])
      show ∀ s. wp ?w Q s ∧ bval b s ⟶ wp c (wp ?w Q) s by auto
    qed
  qed
  thus ?case
  proof(rule weaken-post)
    show ∀ s. wp ?w Q s ∧ ¬ bval b s ⟶ Q s by auto
  qed
qed auto

lemma hoare-relative-complete: assumes ⊨ {P}c{Q} shows ⊢ {P}c{Q}
proof(rule strengthen-pre)
  show ∀ s. P s ⟶ wp c Q s using assms
    by (auto simp: hoare-valid-def wp-def)
  show ⊢ {wp c Q} c {Q} by(rule wp-is-pre)
qed

end
```

## 11 Verification Conditions

**theory** *VC* **imports** *Hoare* **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* = (λ*s*. *Q*(*s*(*x* := *aval a s*))) |
*pre* (*Asemi* $c_1$ $c_2$) *Q* = *pre* $c_1$ (*pre* $c_2$ *Q*) |
*pre* (*Aif b* $c_1$ $c_2$) *Q* =
(λ*s*. (*bval b s* ⟶ *pre* $c_1$ *Q s*) ∧
(¬ *bval b s* ⟶ *pre* $c_2$ *Q s*)) |
*pre* (*Awhile b I c*) *Q* = *I*

Verification condition:

**fun** *vc* :: *acom* ⇒ *assn* ⇒ *assn* **where**
*vc Askip Q* = (λ*s*. *True*) |
*vc* (*Aassign x a*) *Q* = (λ*s*. *True*) |
*vc* (*Asemi* $c_1$ $c_2$) *Q* = (λ*s*. *vc* $c_1$ (*pre* $c_2$ *Q*) *s* ∧ *vc* $c_2$ *Q s*) |
*vc* (*Aif b* $c_1$ $c_2$) *Q* = (λ*s*. *vc* $c_1$ *Q s* ∧ *vc* $c_2$ *Q s*) |
*vc* (*Awhile b I c*) *Q* =
(λ*s*. (*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* $c_1$ $c_2$) = (*astrip* $c_1$; *astrip* $c_2$) |
*astrip* (*Aif b* $c_1$ $c_2$) = (*IF b THEN astrip* $c_1$ *ELSE astrip* $c_2$) |
*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*: $\forall s.\ I\ s \wedge \neg\ bval\ b\ s \longrightarrow Q\ s$ **and**
*pre*: $\forall s.\ I\ s \wedge bval\ b\ s \longrightarrow pre\ c\ I\ s$ **by** *simp-all*
**have** $\vdash \{pre\ c\ I\}\ astrip\ c\ \{I\}$ **by**(*rule Awhile.hyps*[*OF vc*])
**with** *pre* **show** $\vdash \{\lambda s.\ I\ s \wedge bval\ b\ s\}\ astrip\ c\ \{I\}$
**by**(*rule strengthen-pre*)
**show** $\forall s.\ I\ s \wedge \neg bval\ b\ s \longrightarrow Q\ s$ **by**(*rule IQ*)
**qed**
**qed** (*auto intro*: *hoare.conseq*)

**corollary** *vc-sound'*:
$(\forall s.\ vc\ c\ Q\ s) \wedge (\forall s.\ P\ s \longrightarrow pre\ c\ Q\ s) \Longrightarrow \vdash \{P\}\ astrip\ c\ \{Q\}$
**by** (*metis strengthen-pre vc-sound*)

## 11.3 Completeness

**lemma** *pre-mono*:
$\forall s.\ P\ s \longrightarrow P'\ s \Longrightarrow pre\ c\ P\ s \Longrightarrow pre\ c\ P'\ s$
**proof** (*induct c arbitrary*: $P\ P'\ s$)
**case** *Asemi* **thus** *?case* **by** *simp metis*
**qed** *simp-all*

**lemma** *vc-mono*:
$\forall s.\ P\ s \longrightarrow P'\ s \Longrightarrow vc\ c\ P\ s \Longrightarrow vc\ c\ P'\ s$
**proof**(*induct c arbitrary*: $P\ P'$)
**case** *Asemi* **thus** *?case* **by** *simp* (*metis pre-mono*)
**qed** *simp-all*

**lemma** *vc-complete*:
$\vdash \{P\}c\{Q\} \Longrightarrow \exists c'.\ astrip\ c' = c \wedge (\forall s.\ vc\ c'\ Q\ s) \wedge (\forall s.\ P\ s \longrightarrow pre$
$c'\ Q\ s)$
(**is** - $\Longrightarrow \exists c'.\ ?G\ P\ c\ Q\ c'$)
**proof** (*induct rule*: *hoare.induct*)
**case** *Skip*
**show** *?case* (**is** $\exists ac.\ ?C\ ac$)
**proof show** *?C Askip* **by** *simp* **qed**
**next**
**case** (*Assign P a x*)
**show** *?case* (**is** $\exists ac.\ ?C\ ac$)
**proof show** *?C*(*Aassign x a*) **by** *simp* **qed**
**next**
**case** (*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* ($\lambda s.$ *P s* ∧ *bval b s*) *c1 Q ac1*
**by** *blast*
**from** *If.hyps* **obtain** *ac2* **where** *ih2*: *?G* ($\lambda s.$ *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* ($\lambda s.$ *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* = ($\lambda s.$ *True*, *Q*) |
*vcpre* (*Aassign x a*) *Q* = ($\lambda s.$ *True*, $\lambda s.$ *Q*(*s*[*a*/*x*])) |
*vcpre* (*Asemi* $c_1$ $c_2$) *Q* =
(*let* ($vc_2$,$wp_2$) = *vcpre* $c_2$ *Q*;
($vc_1$,$wp_1$) = *vcpre* $c_1$ $wp_2$
*in* ($\lambda s.$ $vc_1$ *s* ∧ $vc_2$ *s*, $wp_1$)) |
*vcpre* (*Aif b* $c_1$ $c_2$) *Q* =
(*let* ($vc_2$,$wp_2$) = *vcpre* $c_2$ *Q*;
($vc_1$,$wp_1$) = *vcpre* $c_1$ *Q*
*in* ($\lambda s.$ $vc_1$ *s* ∧ $vc_2$ *s*, $\lambda s.$ (*bval b s* ⟶ $wp_1$ *s*) ∧ (¬*bval b s* ⟶ $wp_2$ *s*)))
|
*vcpre* (*Awhile b I c*) *Q* =
(*let* (*vcc*,*wpc*) = *vcpre c I*
*in* ($\lambda s.$ (*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

**theory** *HoareT* **imports** *Hoare-Sound-Complete* **begin**

Now that we have termination, we can define total validity, $\models_t$, as partial validity and guaranteed termination:

**definition** *hoare-tvalid* :: *assn* ⇒ *com* ⇒ *assn* ⇒ *bool*
($\models_t$ {(*1*-)}/ (-)/ {(*1*-)} *50*) **where**
$\models_t \{P\}c\{Q\} \ \equiv\ \forall s.\ P\ s \longrightarrow (\exists t.\ (c,s) \Rightarrow t \wedge Q\ t)$

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

**inductive**
*hoaret* :: *assn* ⇒ *com* ⇒ *assn* ⇒ *bool* ($\vdash_t$ ({(*1*-)}/ (-)/ {(*1*-)}) *50*)
**where**
*Skip*: $\vdash_t \{P\}\ SKIP\ \{P\}$ |
*Assign*: $\vdash_t \{\lambda s.\ P(s[a/x])\}\ x{::=}a\ \{P\}$ |
*Semi*: $[\![ \vdash_t \{P_1\}\ c_1\ \{P_2\};\ \vdash_t \{P_2\}\ c_2\ \{P_3\} ]\!] \Longrightarrow \vdash_t \{P_1\}\ c_1;c_2\ \{P_3\}$ |
*If*: $[\![ \vdash_t \{\lambda s.\ P\ s \wedge bval\ b\ s\}\ c_1\ \{Q\};\ \vdash_t \{\lambda s.\ P\ s \wedge \neg\ bval\ b\ s\}\ c_2\ \{Q\} ]\!]$
$\Longrightarrow \vdash_t \{P\}\ IF\ b\ THEN\ c_1\ ELSE\ c_2\ \{Q\}$ |
*While*:
$[\![ \bigwedge n{::}nat.\ \vdash_t \{\lambda s.\ P\ s \wedge bval\ b\ s \wedge f\ s = n\}\ c\ \{\lambda s.\ P\ s \wedge f\ s < n\}]\!]$
$\Longrightarrow \vdash_t \{P\}\ WHILE\ b\ DO\ c\ \{\lambda s.\ P\ s \wedge \neg bval\ b\ s\}$ |
*conseq*: $[\![ \forall s.\ P'\ s \longrightarrow P\ s;\ \vdash_t \{P\}c\{Q\};\ \forall s.\ Q\ s \longrightarrow Q'\ s\ ]\!] \Longrightarrow$
$\vdash_t \{P'\}c\{Q'\}$

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

**lemma** *strengthen-pre*:
$[\![ \forall s.\ P'\ s \longrightarrow P\ s;\ \vdash_t \{P\}\ c\ \{Q\} ]\!] \Longrightarrow \vdash_t \{P'\}\ c\ \{Q\}$
**by** (*metis conseq*)

**lemma** *weaken-post*:
$[\![ \vdash_t \{P\}\ c\ \{Q\};\ \forall s.\ Q\ s \longrightarrow Q'\ s ]\!] \Longrightarrow\ \vdash_t \{P\}\ c\ \{Q'\}$

**by** (*metis conseq*)

**lemma** *Assign'*: $\forall s.\ P\ s \longrightarrow Q(s[a/x]) \Longrightarrow \vdash_t \{P\}\ x ::= a\ \{Q\}$
**by** (*simp add*: *strengthen-pre*[*OF* - *Assign*])

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

Our standard example:

**abbreviation** *w n* ==
  *WHILE Less* (*V 1*) (*N n*)
  *DO* ( *1* ::= *Plus* (*V 1*) (*N 1*); *0* ::= *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** $P = \lambda s.\ s\ 0 = \sum\ \{1..s\ 1\} \wedge s\ 1 \le n$
   **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*
**done**

The soundness theorem:

**theorem** *hoaret-sound*: $\vdash_t \{P\}c\{Q\} \implies\ \models_t \{P\}c\{Q\}$
**proof**(*unfold hoare-tvalid-def*, *induct rule*: *hoaret.induct*)
  **case** (*While P b f c*)
  **show** *?case*
  **proof**
    **fix** *s*
    **show** $P\ s \longrightarrow (\exists t.$ (*WHILE b DO c*, $s) \Rightarrow t \wedge P\ t \wedge \neg\ bval\ b\ t)$

**proof**(*induct f s arbitrary*: *s rule*: *less-induct*)
**case** (*less n*)
**thus** *?case* **by** (*metis While*(*2*) *WhileFalse WhileTrue*)
**qed**
**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* ($wp_t$) **where**
$wp_t$ *c Q* ≡ λ*s*. ∃ *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* = (λ*s*. *Q*(*s*(*x* := *aval e s*)))
**by**(*auto intro*!: *ext simp*: *wpt-def*)

**lemma** [*simp*]: $wp_t$ ($c_1$;$c_2$) *Q* = $wp_t$ $c_1$ ($wp_t$ $c_2$ *Q*)
**unfolding** *wpt-def*
**apply**(*rule ext*)
**apply** *auto*
**done**

**lemma** [*simp*]:
$wp_t$ (*IF b THEN* $c_1$ *ELSE* $c_2$) *Q* = (λ*s*. $wp_t$ (*if bval b s then* $c_1$ *else* $c_2$) *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*: $(WHILE\ b\ DO\ c,s) \Rightarrow t \Longrightarrow \exists\, n.\ Its\ b\ c\ s\ n$
**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 *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*: $[\![\ bval\ b\ s;\ (c,\ s) \Rightarrow s';\ (WHILE\ b\ DO\ c,\ s') \Rightarrow t]\!]$
$\Longrightarrow its\ b\ c\ s = Suc(its\ b\ c\ s')$
**by** (*metis its-def WHILE-Its Its.intros*(*2*) *Its-fun the-equality*)

**lemma** *wpt-is-pre*: $\vdash_t \{wp_t\ c\ Q\}\ c\ \{Q\}$
**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** *?w* = *WHILE b DO c*
**{ fix** *n*
**have** $\forall s.\ wp_t\ ?w\ Q\ s \wedge bval\ b\ s \wedge its\ b\ c\ s = n \longrightarrow$
$wp_t\ c\ (\lambda s'.\ wp_t\ ?w\ Q\ s' \wedge its\ b\ c\ s' < n)\ s$
**unfolding** *wpt-def* **by** (*metis WhileE its-Suc lessI*)
**note** *strengthen-pre*[*OF this While*]
**} note** *hoaret.While*[*OF this*]
**moreover have** $\forall s.\ wp_t\ ?w\ Q\ s \wedge \neg\ bval\ b\ s \longrightarrow Q\ s$ **by** (*auto simp add*:*wpt-def*)
**ultimately show** *?case* **by**(*rule weaken-post*)
**qed**

In the *While*-case, *its* provides the obvious termination argument.

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

**theorem** *hoaret-complete*: $\models_t \{P\}c\{Q\} \Longrightarrow \vdash_t \{P\}c\{Q\}$
**apply**(*rule strengthen-pre*[*OF - wpt-is-pre*])
**apply**(*auto simp*: *hoare-tvalid-def hoare-valid-def wpt-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* (- ::= - [*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*)
| *Var name com* ((*1*{*VAR* -;;/ -}))
| *Proc name com com* ((*1*{*PROC* - = -;;/ -}))
| *CALL name*

**definition** *test-com* =
{*VAR 0*;;
*0* ::= *N 0*;
{*PROC 0* = *0* ::= *Plus* (*V 0*) (*V 0*);;
{*PROC 1* = *CALL 0*;;

```
{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 ⊢ ($c_1$,$s_1$) ⇒ $s_2$; pe ⊢ ($c_2$,$s_2$) ⇒ $s_3$ ⟧ ⟹
  pe ⊢ ($c_1$;$c_2$, $s_1$) ⇒ $s_3$ |

*IfTrue*: ⟦ bval b s; pe ⊢ ($c_1$,s) ⇒ t ⟧ ⟹
  pe ⊢ (IF b THEN $c_1$ ELSE $c_2$, s) ⇒ t |
*IfFalse*: ⟦ ¬bval b s; pe ⊢ ($c_2$,s) ⇒ t ⟧ ⟹
  pe ⊢ (IF b THEN $c_1$ ELSE $c_2$, s) ⇒ t |

*WhileFalse*: ¬bval b s ⟹ pe ⊢ (WHILE b DO c,s) ⇒ s |
*WhileTrue*:
 ⟦ bval b $s_1$; pe ⊢ (c,$s_1$) ⇒ $s_2$; pe ⊢ (WHILE b DO c, $s_2$) ⇒ $s_3$ ⟧ ⟹
  pe ⊢ (WHILE b DO c, $s_1$) ⇒ $s_3$ |

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

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

**end**

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

### 13.1.2 Static Scoping of Procedures, Dynamic of Variables

**types** $penv = (name \times com)\ list$

**inductive**
$big\text{-}step :: penv \Rightarrow com \times state \Rightarrow state \Rightarrow bool\ (\text{-} \vdash \text{-} \Rightarrow \text{-}\ [60,0,60]\ 55)$
**where**
*Skip*: $pe \vdash (SKIP,s) \Rightarrow s$ |
*Assign*: $pe \vdash (x ::= a,s) \Rightarrow s(x := aval\ a\ s)$ |
*Semi*: $[\![\ pe \vdash (c_1,s_1) \Rightarrow s_2;\ \ pe \vdash (c_2,s_2) \Rightarrow s_3\ ]\!] \Longrightarrow$
$pe \vdash (c_1;c_2,\ s_1) \Rightarrow s_3$ |

*IfTrue*: $[\![\ bval\ b\ s;\ \ pe \vdash (c_1,s) \Rightarrow t\ ]\!] \Longrightarrow$
$pe \vdash (IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \Rightarrow t$ |
*IfFalse*: $[\![\ \neg bval\ b\ s;\ \ pe \vdash (c_2,s) \Rightarrow t\ ]\!] \Longrightarrow$
$pe \vdash (IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \Rightarrow t$ |

*WhileFalse*: $\neg bval\ b\ s \Longrightarrow pe \vdash (WHILE\ b\ DO\ c,s) \Rightarrow s$ |
*WhileTrue*:
$[\![\ bval\ b\ s_1;\ \ pe \vdash (c,s_1) \Rightarrow s_2;\ \ pe \vdash (WHILE\ b\ DO\ c,\ s_2) \Rightarrow s_3\ ]\!] \Longrightarrow$
$pe \vdash (WHILE\ b\ DO\ c,\ s_1) \Rightarrow s_3$ |

*Var*: $pe \vdash (c,s) \Rightarrow t \ \Longrightarrow\ pe \vdash (\{VAR\ x;;\ c\},\ s) \Rightarrow t(x := s\ x)$ |

*Call1*: $(p,c)\#pe \vdash (c,\ s) \Rightarrow t \ \Longrightarrow\ (p,c)\#pe \vdash (CALL\ p,\ s) \Rightarrow t$ |
*Call2*: $[\![\ p' \neq p;\ \ pe \vdash (CALL\ p,\ s) \Rightarrow t\ ]\!] \Longrightarrow$
$(p',c)\#pe \vdash (CALL\ p,\ s) \Rightarrow t$ |

*Proc*: $(p,cp)\#pe \vdash (c,s) \Rightarrow t \ \Longrightarrow\ pe \vdash (\{PROC\ p = cp;;\ c\},\ s) \Rightarrow t$

**code-pred** *big-step* **.**

**inductive** *exec* **where**
$[] \vdash (c,nth\ ns) \Rightarrow s \ \Longrightarrow\ exec\ c\ ns\ (list\ s\ (length\ ns))$

**code-pred** *exec* **.**

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

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

**end**

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

### 13.1.3 Static Scoping of Procedures and Variables

**types**
$addr = nat$
$venv = name \Rightarrow addr$
$store = addr \Rightarrow nat$
$penv = (name \times com \times venv)\ list$

**fun** $venv :: penv \times venv \times nat \Rightarrow venv$ **where**
$venv(\text{-},ve,\text{-}) = ve$

**inductive**
$big\text{-}step :: penv \times venv \times nat \Rightarrow com \times store \Rightarrow store \Rightarrow bool$
$(\text{-} \vdash \text{-} \Rightarrow \text{-}\ [60,0,60]\ 55)$
**where**
*Skip*: $e \vdash (SKIP,s) \Rightarrow s$ |
*Assign*: $(pe,ve,f) \vdash (x ::= a,s) \Rightarrow s(ve\ x := aval\ a\ (s\ o\ ve))$ |
*Semi*: $[\![\ e \vdash (c_1,s_1) \Rightarrow s_2;\ \ e \vdash (c_2,s_2) \Rightarrow s_3\ ]\!] \Longrightarrow$
$e \vdash (c_1;c_2,\ s_1) \Rightarrow s_3$ |

*IfTrue*: $[\![\ bval\ b\ (s \circ venv\ e);\ \ e \vdash (c_1,s) \Rightarrow t\ ]\!] \Longrightarrow$
$e \vdash (IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \Rightarrow t$ |
*IfFalse*: $[\![\ \neg bval\ b\ (s \circ venv\ e);\ \ e \vdash (c_2,s) \Rightarrow t\ ]\!] \Longrightarrow$
$e \vdash (IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \Rightarrow t$ |

*WhileFalse*: $\neg bval\ b\ (s \circ venv\ e) \Longrightarrow e \vdash (WHILE\ b\ DO\ c,s) \Rightarrow s$ |
*WhileTrue*:
$[\![\ bval\ b\ (s_1 \circ venv\ e);\ \ e \vdash (c,s_1) \Rightarrow s_2;$
$e \vdash (WHILE\ b\ DO\ c,\ s_2) \Rightarrow s_3\ ]\!] \Longrightarrow$
$e \vdash (WHILE\ b\ DO\ c,\ s_1) \Rightarrow s_3$ |

*Var*: $(pe,ve(x{:=}f),f{+}1) \vdash (c,s) \Rightarrow t \Longrightarrow$

$(pe,ve,f) \vdash (\{VAR\ x;;\ c\},\ s) \Rightarrow t(x := s\ x)$ |

*Call1*: $((p,c,ve)\#pe,ve,f) \vdash (c,\ s) \Rightarrow t \implies$
$((p,c,ve)\#pe,ve',f) \vdash (CALL\ p,\ s) \Rightarrow t$ |
*Call2*: $[\![\ p' \neq p;\ \ (pe,ve,f) \vdash (CALL\ p,\ s) \Rightarrow t\ ]\!] \implies$
$((p',c,ve')\#pe,ve,f) \vdash (CALL\ p,\ s) \Rightarrow t$ |

*Proc*: $((p,cp,ve)\#pe,ve,f) \vdash (c,s) \Rightarrow t$
$\implies (pe,ve,f) \vdash (\{PROC\ p = cp;;\ c\},\ s) \Rightarrow t$

**code-pred** *big-step* **.**

**inductive** *exec* **where**
$([],\ \lambda n.\ n,\ 0) \vdash (c,nth\ ns) \Rightarrow s \implies exec\ c\ ns\ (list\ s\ (length\ ns))$

**code-pred** *exec* **.**

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

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

**end**

**theory** *C-like* **imports** *Util* **begin**

## 13.2 A C-like Language

**types** *state* = *nat* ⇒ *nat*

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

**fun** *aval* :: *aexp* ⇒ *state* ⇒ *nat* **where**
$aval\ (N\ n)\ s = n$ |
$aval\ (!a)\ s = s(aval\ a\ s)$ |
$aval\ (Plus\ a_1\ a_2)\ s = aval\ a_1\ s + aval\ a_2\ s$

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

**primrec** *bval* :: *bexp* ⇒ *state* ⇒ *bool* **where**
$bval\ (B\ bv)\ \text{-} = bv$ |
$bval\ (Not\ b)\ s = (\neg\ bval\ b\ s)$ |
$bval\ (And\ b_1\ b_2)\ s = (if\ bval\ b_1\ s\ then\ bval\ b_2\ s\ else\ False)$ |
$bval\ (Less\ a_1\ a_2)\ s = (aval\ a_1\ s < aval\ a_2\ s)$

```
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 s),n) |
New:     (New lhs a,s,n) ⇒ (s(aval lhs s := n), n+aval a s)  |
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*}

**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* ⇒ '*a* ⇒ '*b* ⇒ '*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*
 *venv* = *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*:
$me \vdash (Null,c) \Rightarrow (null,c)$ |
*New*:
$me \vdash (New,ve,s,n) \Rightarrow (Ref\ n,ve,s(n := (\lambda f.\ null)),n+1)$ |
*Vaccess*:
$me \vdash (V\ x,ve,sn) \Rightarrow (ve\ x,ve,sn)$ |
*Faccess*:
$me \vdash (e,c) \Rightarrow (Ref\ a,ve',s',n') \Longrightarrow$
$me \vdash (e{\bullet}f,c) \Rightarrow (s'\ a\ f,ve',s',n')$ |
*Vassign*:
$me \vdash (e,c) \Rightarrow (r,ve',sn') \Longrightarrow$
$me \vdash (x ::= e,c) \Rightarrow (r,ve'(x{:=}r),sn')$ |
*Fassign*:
$[\![\ me \vdash (oe,c_1) \Rightarrow (Ref\ a,c_2);\ \ me \vdash (e,c_2) \Rightarrow (r,ve_3,s_3,n_3)\ ]\!] \Longrightarrow$
$me \vdash (oe{\bullet}f ::= e,c_1) \Rightarrow (r,ve_3,s_3(a,f := r),n_3)$ |
*Mcall*:
$[\![\ me \vdash (oe,c_1) \Rightarrow (or,c_2);\ \ me \vdash (pe,c_2) \Rightarrow (pr,ve_3,sn_3);$
$ve = (\lambda x.\ null)(''this'' := or,\ ''param'' := pr);$
$me \vdash (me\ m,ve,sn_3) \Rightarrow (r,ve',sn_4)\ ]\!]$
$\Longrightarrow$

$me \vdash (oe{\cdot}m{<}pe{>},c_1) \Rightarrow (r,ve_3,sn_4)$ |
*Semi*:
$[\![ \; me \vdash (e_1,c_1) \Rightarrow (r,c_2); \;\; me \vdash (e_2,c_2) \Rightarrow c_3 \;]\!] \Longrightarrow$
$me \vdash (e_1;\; e_2,c_1) \Rightarrow c_3$ |
*IfTrue*:
$[\![ \; me \vdash (b,c_1) \rightarrow (True,c_2); \;\; me \vdash (e_1,c_2) \Rightarrow c_3 \;]\!] \Longrightarrow$
$me \vdash (IF\ b\ THEN\ e_1\ ELSE\ e_2,c_1) \Rightarrow c_3$ |
*IfFalse*:
$[\![ \; me \vdash (b,c_1) \rightarrow (False,c_2); \;\; me \vdash (e_2,c_2) \Rightarrow c_3 \;]\!] \Longrightarrow$
$me \vdash (IF\ b\ THEN\ e_1\ ELSE\ e_2,c_1) \Rightarrow c_3$ |

$me \vdash (B\ bv,c) \rightarrow (bv,c)$ |

$me \vdash (b,c_1) \rightarrow (bv,c_2) \Longrightarrow me \vdash (Not\ b,c_1) \rightarrow (\neg bv,c_2)$ |

$[\![ \; me \vdash (b_1,c_1) \rightarrow (bv_1,c_2); \;\; me \vdash (b_2,c_2) \rightarrow (bv_2,c_3) \;]\!] \Longrightarrow$
$me \vdash (And\ b_1\ b_2,c_1) \rightarrow (bv_1 \wedge bv_2,c_3)$ |

$[\![ \; me \vdash (e_1,c_1) \Rightarrow (r_1,c_2); \;\; me \vdash (e_2,c_2) \Rightarrow (r_2,c_3) \;]\!] \Longrightarrow$
$me \vdash (Eq\ e_1\ e_2,c_1) \rightarrow (r_1{=}r_2,c_3)$

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

Execution of semantics. The final variable environment and store are converted 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**
$me \vdash (e,(\lambda x.\ null),nth[\,],0) \Rightarrow (r,ve',s',n) \Longrightarrow$
$exec\ me\ e\ xs\ fs\ r\ (map\ ve'\ xs)\ (map\ (\lambda n.\ map\ (s'\ n)\ fs)\ [0..{<}n])$

**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**
*menv* = (λ*m*. *Null*)(''*succ*'' := *m-succ*, ''*add*'' := *m-add*)

The main code, adding 1 and 2:

**definition** *main* =
''*1*'' ::= *Null*•''*succ*''<*Null*>;
''*2*'' ::= *V* ''*1*''•''*succ*''<*Null*>;
*V* ''*2*'' • ''*add*'' <*V* ''*1*''>

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

**end**