

# Concrete Semantics

Tobias Nipkow & Gerwin Klein

August 24, 2018

## Abstract

This document presents formalizations of the semantics of a simple imperative programming language together with a number of applications: a compiler, type systems, various program analyses and abstract interpreters. These theories form the basis of the book *Concrete Semantics with Isabelle/HOL* by Nipkow and Klein [2].

## Contents

<b>1</b>	<b>Arithmetic and Boolean Expressions</b>	<b>4</b>
1.1	Arithmetic Expressions . . . . .	4
1.2	Constant Folding . . . . .	5
1.3	Boolean Expressions . . . . .	6
1.4	Constant Folding . . . . .	6
<b>2</b>	<b>Stack Machine and Compilation</b>	<b>7</b>
2.1	Stack Machine . . . . .	7
2.2	Compilation . . . . .	8
<b>3</b>	<b>IMP — A Simple Imperative Language</b>	<b>9</b>
3.1	Big-Step Semantics of Commands . . . . .	9
3.2	Rule inversion . . . . .	11
3.3	Command Equivalence . . . . .	13
3.4	Execution is deterministic . . . . .	15
<b>4</b>	<b>Small-Step Semantics of Commands</b>	<b>16</b>
4.1	The transition relation . . . . .	16
4.2	Executability . . . . .	16
4.3	Proof infrastructure . . . . .	16
4.4	Equivalence with big-step semantics . . . . .	17
4.5	Final configurations and infinite reductions . . . . .	19

<b>5</b>	<b>Compiler for IMP</b>	<b>20</b>
5.1	List setup . . . . .	20
5.2	Instructions and Stack Machine . . . . .	21
5.3	Verification infrastructure . . . . .	22
5.4	Compilation . . . . .	23
5.5	Preservation of semantics . . . . .	25
<b>6</b>	<b>Compiler Correctness, Reverse Direction</b>	<b>25</b>
6.1	Definitions . . . . .	26
6.2	Basic properties of <i>exec<sub>n</sub></i> . . . . .	26
6.3	Concrete symbolic execution steps . . . . .	27
6.4	Basic properties of <i>succs</i> . . . . .	27
6.5	Splitting up machine executions . . . . .	31
6.6	Correctness theorem . . . . .	35
<b>7</b>	<b>A Typed Language</b>	<b>39</b>
7.1	Arithmetic Expressions . . . . .	40
7.2	Boolean Expressions . . . . .	40
7.3	Syntax of Commands . . . . .	40
7.4	Small-Step Semantics of Commands . . . . .	41
7.5	The Type System . . . . .	41
7.6	Well-typed Programs Do Not Get Stuck . . . . .	42
<b>8</b>	<b>Security Type Systems</b>	<b>44</b>
8.1	Security Levels and Expressions . . . . .	44
8.2	Security Typing of Commands . . . . .	46
8.3	Termination-Sensitive Systems . . . . .	52
<b>9</b>	<b>Definite Initialization Analysis</b>	<b>56</b>
9.1	The Variables in an Expression . . . . .	57
9.2	Initialization-Sensitive Expressions Evaluation . . . . .	59
9.3	Definite Initialization Analysis . . . . .	60
9.4	Initialization-Sensitive Big Step Semantics . . . . .	61
9.5	Soundness wrt Big Steps . . . . .	61
9.6	Initialization-Sensitive Small Step Semantics . . . . .	62
9.7	Soundness wrt Small Steps . . . . .	63
<b>10</b>	<b>Constant Folding</b>	<b>64</b>
10.1	Semantic Equivalence up to a Condition . . . . .	64
10.2	Simple folding of arithmetic expressions . . . . .	67
<b>11</b>	<b>Live Variable Analysis</b>	<b>72</b>
11.1	Liveness Analysis . . . . .	72
11.2	Correctness . . . . .	73
11.3	Program Optimization . . . . .	75

11.4 True Liveness Analysis . . . . .	78
<b>12 Denotational Semantics of Commands</b>	<b>83</b>
12.1 Continuity . . . . .	84
12.2 The denotational semantics is deterministic . . . . .	86
<b>13 Hoare Logic</b>	<b>87</b>
13.1 Hoare Logic for Partial Correctness . . . . .	87
13.2 Examples . . . . .	88
13.3 Soundness and Completeness . . . . .	90
13.4 Verification Condition Generation . . . . .	92
13.5 Hoare Logic for Total Correctness . . . . .	94
<b>14 Abstract Interpretation</b>	<b>99</b>
14.1 Complete Lattice . . . . .	99
14.2 Annotated Commands . . . . .	100
14.3 Collecting Semantics of Commands . . . . .	103
14.4 Collecting Semantics Examples . . . . .	109
14.5 Abstract Interpretation Test Programs . . . . .	110
14.6 Abstract Interpretation . . . . .	112
14.7 Computable State . . . . .	123
14.8 Computable Abstract Interpretation . . . . .	127
14.9 Constant Propagation . . . . .	133
14.10 Parity Analysis . . . . .	137
14.11 Backward Analysis of Expressions . . . . .	140
14.12 Interval Analysis . . . . .	146
14.13 Widening and Narrowing . . . . .	156

# 1 Arithmetic and Boolean Expressions

## 1.1 Arithmetic Expressions

**theory** *AExp* **imports** *Main* **begin**

**type\_synonym** *vname* = *string*

**type\_synonym** *val* = *int*

**type\_synonym** *state* = *vname*  $\Rightarrow$  *val*

**datatype** *aexp* = *N int* | *V vname* | *Plus aexp aexp*

**fun** *aval* :: *aexp*  $\Rightarrow$  *state*  $\Rightarrow$  *val* **where**

*aval* (*N n*) *s* = *n* |

*aval* (*V x*) *s* = *s x* |

*aval* (*Plus a1 a2*) *s* = *aval a1 s* + *aval a2 s*

**value** *aval* (*Plus* (*V "x"*) (*N 5*)) ( $\lambda x.$  *if* *x* = *"x"* *then* 7 *else* 0)

The same state more concisely:

**value** *aval* (*Plus* (*V "x"*) (*N 5*)) (( $\lambda x.$  0) (*"x" := 7*))

A little syntax magic to write larger states compactly:

**definition** *null\_state* (<>) **where**

*null\_state*  $\equiv$   $\lambda x.$  0

**syntax**

*\_State* :: *updbinds*  $\Rightarrow$  'a (<\_>)

**translations**

*\_State ms* == *\_Update* <> *ms*

*\_State* (*\_updbinds b bs*) <= *\_Update* (*\_State b*) *bs*

We can now write a series of updates to the function  $\lambda x.$  0 compactly:

**lemma** <*a := 1, b := 2*> = (<> (*a := 1*)) (*b := (2::int)*)

**by** (*rule refl*)

**value** *aval* (*Plus* (*V "x"*) (*N 5*)) <*"x" := 7*>

In the <*a := b*> syntax, variables that are not mentioned are 0 by default:

**value** *aval* (*Plus* (*V "x"*) (*N 5*)) <*"y" := 7*>

Note that this <...> syntax works for any function space  $\tau_1 \Rightarrow \tau_2$  where  $\tau_2$  has a 0.

## 1.2 Constant Folding

Evaluate constant subexpressions:

```
fun asimp_const :: aexp  $\Rightarrow$  aexp where  
asimp_const (N n) = N n |  
asimp_const (V x) = V x |  
asimp_const (Plus a1 a2) =  
  (case (asimp_const a1, asimp_const a2) of  
    (N n1, N n2)  $\Rightarrow$  N(n1+n2) |  
    (b1,b2)  $\Rightarrow$  Plus b1 b2)
```

**theorem** *aval\_asimp\_const*:

*aval* (*asimp\_const* *a*) *s* = *aval* *a* *s*

**apply**(*induction* *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  $\Rightarrow$  aexp  $\Rightarrow$  aexp where  
plus (N i1) (N i2) = N(i1+i2) |  
plus (N i) a = (if i=0 then a else Plus (N i) a) |  
plus a (N i) = (if i=0 then a else Plus a (N i)) |  
plus a1 a2 = Plus a1 a2
```

**lemma** *aval\_plus[simp]*:

*aval* (*plus* *a*<sub>1</sub> *a*<sub>2</sub>) *s* = *aval* *a*<sub>1</sub> *s* + *aval* *a*<sub>2</sub> *s*

**apply**(*induction* *a*<sub>1</sub> *a*<sub>2</sub> *rule: plus.induct*)

**apply** *simp\_all*

**done**

**fun** *asimp* :: *aexp*  $\Rightarrow$  *aexp* **where**

*asimp* (N *n*) = N *n* |

*asimp* (V *x*) = V *x* |

*asimp* (Plus *a*<sub>1</sub> *a*<sub>2</sub>) = *plus* (*asimp* *a*<sub>1</sub>) (*asimp* *a*<sub>2</sub>)

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 "x") (N 0)))

**theorem** *aval\_asimp[simp]*:

*aval* (*asimp* *a*) *s* = *aval* *a* *s*

**apply**(*induction* *a*)

```

apply simp_all
done

```

```

end

```

### 1.3 Boolean Expressions

```

theory BExp imports AExp begin

```

```

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

```

```

fun bval :: bexp  $\Rightarrow$  state  $\Rightarrow$  bool where

```

```

bval (Bc v) s = v |

```

```

bval (Not b) s = ( $\neg$  bval b s) |

```

```

bval (And b1 b2) s = (bval b1 s  $\wedge$  bval b2 s) |

```

```

bval (Less a1 a2) s = (aval a1 s < aval a2 s)

```

```

value bval (Less (V "x") (Plus (N 3) (V "y")))
  <"x" := 3, "y" := 1>

```

### 1.4 Constant Folding

Optimizing constructors:

```

fun less :: aexp  $\Rightarrow$  aexp  $\Rightarrow$  bexp where

```

```

less (N n1) (N n2) = Bc(n1 < n2) |

```

```

less a1 a2 = Less a1 a2

```

```

lemma [simp]: bval (less a1 a2) s = (aval a1 s < aval a2 s)

```

```

apply(induction a1 a2 rule: less.induct)

```

```

apply simp_all

```

```

done

```

```

fun and :: bexp  $\Rightarrow$  bexp  $\Rightarrow$  bexp where

```

```

and (Bc True) b = b |

```

```

and b (Bc True) = b |

```

```

and (Bc False) b = Bc False |

```

```

and b (Bc False) = Bc False |

```

```

and b1 b2 = And b1 b2

```

```

lemma bval_and[simp]: bval (and b1 b2) s = (bval b1 s  $\wedge$  bval b2 s)

```

```

apply(induction b1 b2 rule: and.induct)

```

```

apply simp_all

```

```

done

```

```

fun not :: bexp  $\Rightarrow$  bexp where

```

```

not (Bc True) = Bc False |
not (Bc False) = Bc True |
not b = Not b

```

```

lemma bval_not[simp]: bval (not b) s = (¬ bval b s)
apply(induction b rule: not.induct)
apply simp_all
done

```

Now the overall optimizer:

```

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

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

value bsimp (And (Less (N 1) (N 0)) (Bc True))

theorem bval (bsimp b) s = bval b s
apply(induction b)
apply simp_all
done

end

```

## 2 Stack Machine and Compilation

```

theory ASM imports AExp begin

```

### 2.1 Stack Machine

```

datatype instr = LOADI val | LOAD vname | ADD

```

```

type_synonym stack = val list

```

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

```

fun exec1 :: instr ⇒ state ⇒ stack ⇒ stack where
exec1 (LOADI n) _ stk = n # stk |
exec1 (LOAD x) s stk = s(x) # stk |

```

$exec1 \text{ ADD } (j \# i \# stk) = (i + j) \# stk$

**fun**  $exec :: instr \ list \Rightarrow state \Rightarrow stack \Rightarrow stack$  **where**  
 $exec [] \_ stk = stk$  |  
 $exec (i\#is) s stk = exec is s (exec1 i s stk)$

**value**  $exec [LOADI 5, LOAD "y", ADD] <"x" := 42, "y" := 43> [50]$

**lemma**  $exec\_append[simp]$ :  
 $exec (is1@is2) s stk = exec is2 s (exec is1 s stk)$   
**apply**( $induction is1 arbitrary: stk$ )  
**apply** ( $auto$ )  
**done**

## 2.2 Compilation

**fun**  $comp :: aexp \Rightarrow instr \ list$  **where**  
 $comp (N n) = [LOADI n]$  |  
 $comp (V x) = [LOAD x]$  |  
 $comp (Plus e_1 e_2) = comp e_1 @ comp e_2 @ [ADD]$

**value**  $comp (Plus (Plus (V "x") (N 1)) (V "z"))$

**theorem**  $exec\_comp: exec (comp a) s stk = aval a s \# stk$   
**apply**( $induction a arbitrary: stk$ )  
**apply** ( $auto$ )  
**done**

**end**  
**theory**  $Star$  **imports**  $Main$   
**begin**

**inductive**  
 $star :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$   
**for**  $r$  **where**  
 $refl: star r x x$  |  
 $step: r x y \Longrightarrow star r y z \Longrightarrow star r x z$

**hide\_fact** (**open**)  $refl step$  — names too generic

**lemma**  $star\_trans$ :  
 $star r x y \Longrightarrow star r y z \Longrightarrow star r x z$   
**proof**( $induction rule: star.induct$ )  
**case**  $refl$  **thus** ? $case$  .



```

next
  case step thus ?case by (metis star.step)
qed

lemmas star_induct =
  star.induct[of r:: 'a*'b  $\Rightarrow$  'a*'b  $\Rightarrow$  bool, split_format(complete)]

declare star.refl[simp,intro]

lemma star_stepI[simp, intro]: r x y  $\Longrightarrow$  star r x y
by(metis star.refl star.step)

code_pred star .

end

```

### 3 IMP — A Simple Imperative Language

```
theory Com imports BExp begin
```

```
datatype
```

```

  com = SKIP
    | Assign vname aexp      (- ::= - [1000, 61] 61)
    | Seq    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
```

#### 3.1 Big-Step Semantics of Commands

```
theory Big_Step imports Com begin
```

The big-step semantics is a straight-forward inductive definition with concrete syntax. Note that the first parameter is a tuple, so the syntax becomes  $(c,s) \Rightarrow s'$ .

```
inductive
```

```
  big_step :: com  $\times$  state  $\Rightarrow$  state  $\Rightarrow$  bool (infix  $\Rightarrow$  55)
```

```
where
```

```
  Skip: (SKIP,s)  $\Rightarrow$  s |
```

```
  Assign: (x ::= a,s)  $\Rightarrow$  s(x := aval a s) |
```

```
  Seq:  $\llbracket (c_1,s_1) \Rightarrow s_2; (c_2,s_2) \Rightarrow s_3 \rrbracket \Longrightarrow (c_1;;c_2, s_1) \Rightarrow s_3$  |
```

```
  IfTrue:  $\llbracket \text{bval } b \text{ s}; (c_1,s) \Rightarrow t \rrbracket \Longrightarrow (\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2, s) \Rightarrow t$  |
```

```
  IfFalse:  $\llbracket \neg \text{bval } b \text{ s}; (c_2,s) \Rightarrow t \rrbracket \Longrightarrow (\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2, s) \Rightarrow t$  |
```

*WhileFalse*:  $\neg \text{bval } b \ s \implies (\text{WHILE } b \ \text{DO } c, s) \Rightarrow s \mid$   
*WhileTrue*:  
 $\llbracket \text{bval } b \ s_1; (c, s_1) \Rightarrow s_2; (\text{WHILE } b \ \text{DO } c, s_2) \Rightarrow s_3 \rrbracket$   
 $\implies (\text{WHILE } b \ \text{DO } c, s_1) \Rightarrow s_3$

**schematic\_goal** *ex*:  $(\text{"x"} ::= N \ 5;; \text{"y"} ::= V \ \text{"x"}, s) \Rightarrow ?t$   
**apply**(*rule Seq*)  
**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**  $\{t. (\text{SKIP}, \lambda_. 0) \Rightarrow t\}$

We need to translate the result state into a list to display it.

**values**  $\{\text{map } t \ [\text{"x"}] \mid t. (\text{SKIP}, \langle \text{"x"} := 42 \rangle) \Rightarrow t\}$

**values**  $\{\text{map } t \ [\text{"x"}] \mid t. (\text{"x"} ::= N \ 2, \langle \text{"x"} := 42 \rangle) \Rightarrow t\}$

**values**  $\{\text{map } t \ [\text{"x"}, \text{"y"}] \mid t.$   
 $(\text{WHILE } \text{Less } (V \ \text{"x"}) \ (V \ \text{"y"}) \ \text{DO } (\text{"x"} ::= \text{Plus } (V \ \text{"x"}) \ (N \ 5)),$   
 $\langle \text{"x"} := 0, \text{"y"} := 13 \rangle) \Rightarrow t\}$

Proof automation:

The introduction rules are good for automatically construction small program executions. The recursive cases may require backtracking, so we declare the set as unsafe intro rules.

**declare** *big\_step.intros* [*intro*]

The standard induction rule

$\llbracket x1 \Rightarrow x2; \bigwedge s. P (\text{SKIP}, s) \ s; \bigwedge x \ a \ s. P (x ::= a, s) (s(x := \text{aval } a \ s));$   
 $\bigwedge c_1 \ s_1 \ s_2 \ c_2 \ s_3.$   
 $\llbracket (c_1, s_1) \Rightarrow s_2; P (c_1, s_1) \ s_2; (c_2, s_2) \Rightarrow s_3; P (c_2, s_2) \ s_3 \rrbracket$   
 $\implies P (c_1;; c_2, s_1) \ s_3;$   
 $\bigwedge b \ s \ c_1 \ t \ c_2.$   
 $\llbracket \text{bval } b \ s; (c_1, s) \Rightarrow t; P (c_1, s) \ t \rrbracket \implies P (\text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2, s)$   
 $t;$   
 $\bigwedge b \ s \ c_2 \ t \ c_1.$

$$\begin{aligned}
& \llbracket \neg \text{bval } b \text{ } s; (c_2, s) \Rightarrow t; P (c_2, s) t \rrbracket \Longrightarrow P (\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2, \\
& s) t; \\
& \wedge b \text{ } s \text{ } c. \neg \text{bval } b \text{ } s \Longrightarrow P (\text{WHILE } b \text{ DO } c, s) s; \\
& \wedge b \text{ } s_1 \text{ } c \text{ } s_2 \text{ } s_3. \\
& \quad \llbracket \text{bval } b \text{ } s_1; (c, s_1) \Rightarrow s_2; P (c, s_1) s_2; (\text{WHILE } b \text{ DO } c, s_2) \Rightarrow s_3; \\
& \quad P (\text{WHILE } b \text{ DO } c, s_2) s_3 \rrbracket \\
& \quad \Longrightarrow P (\text{WHILE } b \text{ DO } c, s_1) s_3 \rrbracket \\
& \Longrightarrow P \text{ } x1 \text{ } x2
\end{aligned}$$

**thm** *big\_step.induct*

This induction schema is almost perfect for our purposes, but our trick for reusing the tuple syntax means that the induction schema has two parameters instead of the  $c$ ,  $s$ , and  $s'$  that we are likely to encounter. Splitting the tuple parameter fixes this:

**lemmas** *big\_step\_induct* = *big\_step.induct*[*split\_format(complete)*]

**thm** *big\_step\_induct*

$$\begin{aligned}
& \llbracket (x1a, x1b) \Rightarrow x2a; \wedge s. P \text{ SKIP } s \text{ } s; \wedge x \text{ } a \text{ } s. P (x ::= a) s (s(x := \text{aval } a \\
& s)); \\
& \wedge c_1 \text{ } s_1 \text{ } s_2 \text{ } c_2 \text{ } s_3. \\
& \quad \llbracket (c_1, s_1) \Rightarrow s_2; P c_1 \text{ } s_1 \text{ } s_2; (c_2, s_2) \Rightarrow s_3; P c_2 \text{ } s_2 \text{ } s_3 \rrbracket \\
& \quad \Longrightarrow P (c_1;; c_2) s_1 \text{ } s_3; \\
& \wedge b \text{ } s \text{ } c_1 \text{ } t \text{ } c_2. \\
& \quad \llbracket \text{bval } b \text{ } s; (c_1, s) \Rightarrow t; P c_1 \text{ } s \text{ } t \rrbracket \Longrightarrow P (\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2) s \text{ } t; \\
& \wedge b \text{ } s \text{ } c_2 \text{ } t \text{ } c_1. \\
& \quad \llbracket \neg \text{bval } b \text{ } s; (c_2, s) \Rightarrow t; P c_2 \text{ } s \text{ } t \rrbracket \Longrightarrow P (\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2) s \text{ } t; \\
& \wedge b \text{ } s \text{ } c. \neg \text{bval } b \text{ } s \Longrightarrow P (\text{WHILE } b \text{ DO } c) s \text{ } s; \\
& \wedge b \text{ } s_1 \text{ } c \text{ } s_2 \text{ } s_3. \\
& \quad \llbracket \text{bval } b \text{ } s_1; (c, s_1) \Rightarrow s_2; P c \text{ } s_1 \text{ } s_2; (\text{WHILE } b \text{ DO } c, s_2) \Rightarrow s_3; \\
& \quad P (\text{WHILE } b \text{ DO } c) s_2 \text{ } s_3 \rrbracket \\
& \quad \Longrightarrow P (\text{WHILE } b \text{ DO } c) s_1 \text{ } s_3 \rrbracket \\
& \Longrightarrow P \text{ } x1a \text{ } x1b \text{ } x2a
\end{aligned}$$

### 3.2 Rule inversion

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

**inductive\_cases** *SkipE*[*elim!*]:  $(\text{SKIP}, s) \Rightarrow t$

**thm** *SkipE*

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

Similarly for the other commands:

```

inductive_cases AssignE[elim!]: (x ::= a, s) ⇒ t
thm AssignE
inductive_cases SeqE[elim!]: (c1;;c2, s1) ⇒ s3
thm SeqE
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

```

**lemma** assign\_simp:

```

(x ::= a, s) ⇒ s' ⇔ (s' = s(x := aval a s))
by auto

```

An example combining rule inversion and derivations

**lemma** Seq\_assoc:

```

(c1;; c2;; c3, s) ⇒ s' ⇔ (c1;; (c2;; c3), s) ⇒ s'

```

**proof**

```

assume (c1;; c2;; c3, s) ⇒ s'
then obtain s1 s2 where
  c1: (c1, s) ⇒ s1 and
  c2: (c2, s1) ⇒ s2 and
  c3: (c3, s2) ⇒ s' by auto

```

**from**  $c2\ c3$   
**have**  $(c2;;\ c3,\ s1) \Rightarrow s'$  **by** (*rule Seq*)  
**with**  $c1$   
**show**  $(c1;;\ (c2;;\ c3),\ s) \Rightarrow s'$  **by** (*rule Seq*)  
**next**  
— The other direction is analogous  
**assume**  $(c1;;\ (c2;;\ c3),\ s) \Rightarrow s'$   
**thus**  $(c1;;\ c2;;\ c3,\ s) \Rightarrow s'$  **by** *auto*  
**qed**

### 3.3 Command Equivalence

We call two statements  $c$  and  $c'$  equivalent wrt. the big-step semantics when  $c$  started in  $s$  terminates in  $s'$  iff  $c'$  started in the same  $s$  also terminates in the same  $s'$ . Formally:

#### abbreviation

$equiv\_c :: com \Rightarrow com \Rightarrow bool$  (**infix**  $\sim 50$ ) **where**  
 $c \sim c' \equiv (\forall s\ t.\ (c,\ s) \Rightarrow t = (c',\ s) \Rightarrow t)$

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

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

#### lemma *unfold\_while*:

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

#### proof –

— to show the equivalence, we look at the derivation tree for

— each side and from that construct a derivation tree for the other side

**have**  $(?iw,\ s) \Rightarrow t$  **if** *asm*:  $(?w,\ s) \Rightarrow t$  **for**  $s\ t$

#### proof –

**from** *asm* **show**  $?thesis$

**proof** *cases* — rule inversion on  $(?w,\ s) \Rightarrow t$

**case** *WhileFalse*

**thus**  $?thesis$  **by** *blast*

#### next

**case** *WhileTrue*

**from**  $\langle bval\ b\ s \rangle \langle (?w,\ s) \Rightarrow t \rangle$  **obtain**  $s'$  **where**

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

— now we can build a derivation tree for the *IF*

— first, the body of the True-branch:

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

— then the whole *IF*

**with**  $\langle bval\ b\ s \rangle$  **show**  $?thesis$  **by** (*rule IfTrue*)

**qed**  
**qed**  
**moreover**  
— now the other direction:  
**have**  $(?w, s) \Rightarrow t$  **if** *assm*:  $(?iw, s) \Rightarrow t$  **for**  $s\ t$   
**proof** —  
  **from** *assm* **show** *?thesis*  
  **proof** *cases* — rule inversion on  $(?iw, s) \Rightarrow t$   
    **case** *IfFalse*  
    **hence**  $s = t$  **using**  $\langle (?iw, s) \Rightarrow t \rangle$  **by** *blast*  
    **thus** *?thesis* **using**  $\langle \neg bval\ b\ s \rangle$  **by** *blast*  
  **next**  
  **case** *IfTrue*  
  — and for this, only the Seq-rule is applicable:  
  **from**  $\langle c;; ?w, s \Rightarrow t \rangle$  **obtain**  $s'$  **where**  
     $(c, s) \Rightarrow s'$  **and**  $(?w, s') \Rightarrow t$  **by** *auto*  
  — with this information, we can build a derivation tree for *WHILE*  
  **with**  $\langle bval\ b\ s \rangle$  **show** *?thesis* **by** (*rule WhileTrue*)  
**qed**  
**qed**  
**ultimately**  
**show** *?thesis* **by** *blast*  
**qed**

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

**lemma** *while\_unfold*:  
 $(WHILE\ b\ DO\ c) \sim (IF\ b\ THEN\ c;;\ WHILE\ b\ DO\ c\ ELSE\ SKIP)$   
**by** *blast*

**lemma** *triv\_if*:  
 $(IF\ b\ THEN\ c\ ELSE\ c) \sim c$   
**by** *blast*

**lemma** *commute\_if*:  
 $(IF\ b1\ THEN\ (IF\ b2\ THEN\ c11\ ELSE\ c12)\ ELSE\ c2)$   
 $\sim$   
 $(IF\ b2\ THEN\ (IF\ b1\ THEN\ c11\ ELSE\ c2)\ ELSE\ (IF\ b1\ THEN\ c12\ ELSE\ c2))$   
**by** *blast*

**lemma** *sim\_while\_cong\_aux*:  
 $(WHILE\ b\ DO\ c, s) \Rightarrow t \implies c \sim c' \implies (WHILE\ b\ DO\ c', s) \Rightarrow t$   
**apply**(*induction WHILE\ b\ DO\ c\ s\ t\ arbitrary: b\ c\ rule: big\_step\_induct*)

**apply** *blast*  
**apply** *blast*  
**done**

**lemma** *sim\_while\_cong*:  $c \sim c' \implies \text{WHILE } b \text{ DO } c \sim \text{WHILE } b \text{ DO } c'$   
**by** (*metis sim\_while\_cong\_aux*)

Command equivalence is an equivalence relation, i.e. it is reflexive, symmetric, and transitive. Because we used an abbreviation above, Isabelle derives this automatically.

**lemma** *sim\_refl*:  $c \sim c$  **by** *simp*

**lemma** *sim\_sym*:  $(c \sim c') = (c' \sim c)$  **by** *auto*

**lemma** *sim\_trans*:  $c \sim c' \implies c' \sim c'' \implies c \sim c''$  **by** *auto*

### 3.4 Execution is deterministic

This proof is automatic.

**theorem** *big\_step\_determ*:  $\llbracket (c,s) \Rightarrow t; (c,s) \Rightarrow u \rrbracket \implies u = t$   
**by** (*induction arbitrary: u rule: big\_step.induct*) *blast+*

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** (*induction arbitrary: t' rule: big\_step.induct*)

— the only interesting case, *WhileTrue*:

**fix**  $b \ c \ s \ s_1 \ t \ t'$

— The assumptions of the rule:

**assume**  $\text{bval } b \ s$  **and**  $(c,s) \Rightarrow s_1$  **and**  $(\text{WHILE } b \text{ DO } c, s_1) \Rightarrow t$

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

**assume**  $IHc: \wedge t'. (c,s) \Rightarrow t' \implies t' = s_1$

**assume**  $IHw: \wedge t'. (\text{WHILE } b \text{ DO } c, s_1) \Rightarrow t' \implies t' = t$

— Premise of implication:

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

**with**  $(\text{bval } b \ s)$  **obtain**  $s_1'$  **where**

$c: (c,s) \Rightarrow s_1'$  **and**

$w: (\text{WHILE } b \text{ DO } c, s_1') \Rightarrow t'$

**by** *auto*

**from**  $c$   $IHc$  **have**  $s_1' = s_1$  **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** *Star Big\_Step* **begin**

### 4.1 The transition relation

**inductive**

*small\_step* :: *com* \* *state*  $\Rightarrow$  *com* \* *state*  $\Rightarrow$  *bool* (**infix**  $\rightarrow$  55)

**where**

*Assign*:  $(x ::= a, s) \rightarrow (SKIP, s(x := aval\ a\ s)) \mid$

*Seq1*:  $(SKIP;;c_2,s) \rightarrow (c_2,s) \mid$

*Seq2*:  $(c_1,s) \rightarrow (c_1',s') \Longrightarrow (c_1;;c_2,s) \rightarrow (c_1';;c_2,s') \mid$

*IfTrue*:  $bval\ b\ s \Longrightarrow (IF\ b\ THEN\ c_1\ ELSE\ c_2,s) \rightarrow (c_1,s) \mid$

*IfFalse*:  $\neg bval\ b\ s \Longrightarrow (IF\ b\ THEN\ c_1\ ELSE\ c_2,s) \rightarrow (c_2,s) \mid$

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

**abbreviation**

*small\_steps* :: *com* \* *state*  $\Rightarrow$  *com* \* *state*  $\Rightarrow$  *bool* (**infix**  $\rightarrow^*$  55)

**where**  $x \rightarrow^* y == star\ small\_step\ x\ y$

### 4.2 Executability

**code\_pred** *small\_step* .

**values**  $\{(c',map\ t\ [\"x\", \"y\", \"z\"]) \mid c'\ t.$   
 $(\"x\" ::= V\ \"z\";;\ \"y\" ::= V\ \"x\",$   
 $\langle \"x\" := 3,\ \"y\" := 7,\ \"z\" := 5 \rangle) \rightarrow^* (c',t)\}$

### 4.3 Proof infrastructure

#### 4.3.1 Induction rules

The default induction rule *small\_step.induct* only works for lemmas of the form  $a \rightarrow b \Longrightarrow \dots$  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  $\rightarrow$  into pairs:

**lemmas** *small\_step\_induct* = *small\_step.induct*[*split\_format*(*complete*)]



### 4.3.2 Proof automation

**declare** *small\_step.intros*[*simp,intro*]

Rule inversion:

**inductive\_cases** *SkipE*[*elim!*]: (*SKIP,s*)  $\rightarrow$  *ct*

**thm** *SkipE*

**inductive\_cases** *AssignE*[*elim!*]: (*x ::= a,s*)  $\rightarrow$  *ct*

**thm** *AssignE*

**inductive\_cases** *SeqE*[*elim!*]: (*c1;;c2,s*)  $\rightarrow$  *ct*

**thm** *SeqE*

**inductive\_cases** *IfE*[*elim!*]: (*IF b THEN c1 ELSE c2,s*)  $\rightarrow$  *ct*

**inductive\_cases** *WhileE*[*elim!*]: (*WHILE b DO c, s*)  $\rightarrow$  *ct*

A simple property:

**lemma** *deterministic*:

$cs \rightarrow cs' \implies cs \rightarrow cs'' \implies cs'' = cs'$

**apply**(*induction arbitrary: cs'' rule: small\_step.induct*)

**apply** *blast+*

**done**

### 4.4 Equivalence with big-step semantics

**lemma** *star\_seq2*: (*c1,s*)  $\rightarrow^*$  (*c1',s'*)  $\implies$  (*c1;;c2,s*)  $\rightarrow^*$  (*c1';c2,s'*)

**proof**(*induction rule: star\_induct*)

**case refl thus ?case by simp**

**next**

**case step**

**thus ?case by (metis Seq2 star.step)**

**qed**

**lemma** *seq\_comp*:

$\llbracket (c1,s1) \rightarrow^* (SKIP,s2); (c2,s2) \rightarrow^* (SKIP,s3) \rrbracket$

$\implies (c1;;c2, s1) \rightarrow^* (SKIP,s3)$

**by**(*blast intro: star.step star\_seq2 star\_trans*)

The following proof corresponds to one on the board where one would show chains of  $\rightarrow$  and  $\rightarrow^*$  steps.

**lemma** *big\_to\_small*:

$cs \Rightarrow t \implies cs \rightarrow^* (SKIP,t)$

**proof** (*induction rule: big\_step.induct*)

**fix s show** (*SKIP,s*)  $\rightarrow^*$  (*SKIP,s*) **by simp**

**next**

**fix x a s show** (*x ::= a,s*)  $\rightarrow^*$  (*SKIP, s(x := aval a s)*) **by auto**

**next**

```

fix  $c1\ c2\ s1\ s2\ s3$ 
assume  $(c1, s1) \rightarrow^* (SKIP, s2)$  and  $(c2, s2) \rightarrow^* (SKIP, s3)$ 
thus  $(c1;;c2, s1) \rightarrow^* (SKIP, s3)$  by  $(rule\ seq\_comp)$ 
next
fix  $s::state$  and  $b\ c0\ c1\ t$ 
assume  $bval\ b\ s$ 
hence  $(IF\ b\ THEN\ c0\ ELSE\ c1, s) \rightarrow (c0, s)$  by  $simp$ 
moreover assume  $(c0, s) \rightarrow^* (SKIP, t)$ 
ultimately
show  $(IF\ b\ THEN\ c0\ ELSE\ c1, s) \rightarrow^* (SKIP, t)$  by  $(metis\ star.simps)$ 
next
fix  $s::state$  and  $b\ c0\ c1\ t$ 
assume  $\neg bval\ b\ s$ 
hence  $(IF\ b\ THEN\ c0\ ELSE\ c1, s) \rightarrow (c1, s)$  by  $simp$ 
moreover assume  $(c1, s) \rightarrow^* (SKIP, t)$ 
ultimately
show  $(IF\ b\ THEN\ c0\ ELSE\ c1, s) \rightarrow^* (SKIP, t)$  by  $(metis\ star.simps)$ 
next
fix  $b\ c$  and  $s::state$ 
assume  $b: \neg bval\ b\ s$ 
let  $?if = IF\ b\ THEN\ c;;\ WHILE\ b\ DO\ c\ ELSE\ SKIP$ 
have  $(WHILE\ b\ DO\ c, s) \rightarrow (?if, s)$  by  $blast$ 
moreover have  $(?if, s) \rightarrow (SKIP, s)$  by  $(simp\ add: b)$ 
ultimately show  $(WHILE\ b\ DO\ c, s) \rightarrow^* (SKIP, s)$  by  $(metis\ star.refl\ star.step)$ 
next
fix  $b\ c\ s\ s'\ t$ 
let  $?w = WHILE\ b\ DO\ c$ 
let  $?if = IF\ b\ THEN\ c;;\ ?w\ ELSE\ SKIP$ 
assume  $w: (?w, s') \rightarrow^* (SKIP, t)$ 
assume  $c: (c, s) \rightarrow^* (SKIP, s')$ 
assume  $b: bval\ b\ s$ 
have  $(?w, s) \rightarrow (?if, s)$  by  $blast$ 
moreover have  $(?if, s) \rightarrow (c;;\ ?w, s)$  by  $(simp\ add: b)$ 
moreover have  $(c;;\ ?w, s) \rightarrow^* (SKIP, t)$  by  $(rule\ seq\_comp[OF\ c\ w])$ 
ultimately show  $(WHILE\ b\ DO\ c, s) \rightarrow^* (SKIP, t)$  by  $(metis\ star.simps)$ 
qed

```

Each case of the induction can be proved automatically:

```

lemma  $cs \Rightarrow t \Longrightarrow cs \rightarrow^* (SKIP, t)$ 
proof  $(induction\ rule: big\_step.induct)$ 
case  $Skip$  show  $?case$  by  $blast$ 
next
case  $Assign$  show  $?case$  by  $blast$ 

```

```

next
  case Seq thus ?case by (blast intro: seq_comp)
next
  case IfTrue thus ?case by (blast intro: star.step)
next
  case IfFalse thus ?case by (blast intro: star.step)
next
  case WhileFalse thus ?case
    by (metis star.step star_step1 small_step.IfFalse small_step.While)
next
  case WhileTrue
  thus ?case
    by (metis While seq_comp small_step.IfTrue star.step[of small_step])
qed

```

```

lemma small1_big_continue:
  cs → cs' ⇒ cs' ⇒ t ⇒ cs ⇒ t
apply (induction arbitrary: t rule: small_step.induct)
apply auto
done

```

```

lemma small_to_big:
  cs →* (SKIP,t) ⇒ cs ⇒ t
apply (induction cs (SKIP,t) rule: star.induct)
apply (auto intro: small1_big_continue)
done

```

Finally, the equivalence theorem:

```

theorem big_iff_small:
  cs ⇒ t = cs →* (SKIP,t)
by (metis big_to_small small_to_big)

```

## 4.5 Final configurations and infinite reductions

```

definition final cs ⇔ ¬(∃ cs'. cs → cs')

```

```

lemma finalD: final (c,s) ⇒ c = SKIP
apply (simp add: final_def)
apply (induction c)
apply blast+
done

```

```

lemma final_iff_SKIP: final (c,s) = (c = SKIP)
by (metis SkipE finalD final_def)

```

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

**lemma** *big\_iff\_small\_termination*:

$$(\exists t. cs \Rightarrow t) \longleftrightarrow (\exists cs'. cs \rightarrow^* cs' \wedge \text{final } cs')$$

**by** (*simp add: big\_iff\_small final\_iff\_SKIP*)

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

**end**

## 5 Compiler for IMP

**theory** *Compiler* **imports** *Big\_Step Star*

**begin**

### 5.1 List setup

In the following, we use the length of lists as integers instead of natural numbers. Instead of converting *nat* to *int* explicitly, we tell Isabelle to coerce *nat* automatically when necessary.

**declare** *[[coercion\_enabled]]*

**declare** *[[coercion int :: nat  $\Rightarrow$  int]]*

Similarly, we will want to access the *i*th element of a list, where *i* is an *int*.

**fun** *inth* :: *'a list  $\Rightarrow$  int  $\Rightarrow$  'a* (**infixl** !! 100) **where**  
*(x # xs) !! i = (if i = 0 then x else xs !! (i - 1))*

The only additional lemma we need about this function is indexing over append:

**lemma** *inth\_append* [*simp*]:

$$0 \leq i \implies$$

$$(xs @ ys) !! i = (\text{if } i < \text{size } xs \text{ then } xs !! i \text{ else } ys !! (i - \text{size } xs))$$

**by** (*induction xs arbitrary: i*) (*auto simp: algebra\_simps*)

We hide coercion *int* applied to *length*:

**abbreviation** (**output**)

$$\text{isize } xs == \text{int } (\text{length } xs)$$

**notation** *isize* (*size*)

## 5.2 Instructions and Stack Machine

**datatype** *instr* =  
*LOADI int* | *LOAD vname* | *ADD* | *STORE vname* |  
*JMP int* | *JMPLESS int* | *JMPGE int*  
**type\_synonym** *stack* = *val list*  
**type\_synonym** *config* = *int* × *state* × *stack*

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

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

**fun** *iexec* :: *instr* ⇒ *config* ⇒ *config* **where**  
*iexec instr (i,s,stk)* = (case *instr* of  
*LOADI n* ⇒ (*i+1,s, n#stk*) |  
*LOAD x* ⇒ (*i+1,s, s x # stk*) |  
*ADD* ⇒ (*i+1,s, (hd2 stk + hd stk) # tl2 stk*) |  
*STORE x* ⇒ (*i+1,s(x := hd stk),tl stk*) |  
*JMP n* ⇒ (*i+1+n,s,stk*) |  
*JMPLESS n* ⇒ (if *hd2 stk* < *hd stk* then *i+1+n* else *i+1,s,tl2 stk*) |  
*JMPGE n* ⇒ (if *hd2 stk* >= *hd stk* then *i+1+n* else *i+1,s,tl2 stk*))

### definition

*exec1* :: *instr list* ⇒ *config* ⇒ *config* ⇒ *bool*  
 ((-/ ⊢ (- →/ -)) [59,0,59] 60)

### where

*P* ⊢ *c* → *c'* =  
 (∃ *i s stk. c = (i,s,stk) ∧ c' = iexec(P!!i) (i,s,stk) ∧ 0 ≤ i ∧ i < size P)*

**lemma** *exec1I* [*intro, code\_pred\_intro*]:

*c' = iexec (P!!i) (i,s,stk) ⇒ 0 ≤ i ⇒ i < size P*  
 ⇒ *P* ⊢ (*i,s,stk*) → *c'*

**by** (*simp add: exec1\_def*)

### abbreviation

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

### where

*exec P* ≡ *star (exec1 P)*

**lemmas** *exec\_induct* = *star.induct* [of *exec1 P, split\_format(complete)*]

**code\_pred** *exec1* **by** (*metis exec1\_def*)

### values

{(*i,map t ["x","y"],stk*) | *i t stk.*}

$[LOAD\ "y",\ STORE\ "x"] \vdash$   
 $(0, \langle "x" := 3, "y" := 4 \rangle, []) \rightarrow^* (i, t, stk)$

### 5.3 Verification infrastructure

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** *iexec\_shift* [*simp*]:

$((n+i', s', stk') = iexec\ x\ (n+i, s, stk)) = ((i', s', stk') = iexec\ x\ (i, s, stk))$

**by** (*auto split:instr.split*)

**lemma** *exec1\_appendR*:  $P \vdash c \rightarrow c' \implies P@P' \vdash c \rightarrow c'$

**by** (*auto simp: exec1\_def*)

**lemma** *exec\_appendR*:  $P \vdash c \rightarrow^* c' \implies P@P' \vdash c \rightarrow^* c'$

**by** (*induction rule: star.induct*) (*fastforce intro: star.step exec1\_appendR*)+

**lemma** *exec1\_appendL*:

**fixes**  $i\ i' :: int$

**shows**

$P \vdash (i, s, stk) \rightarrow (i', s', stk') \implies$

$P' @ P \vdash (size(P') + i, s, stk) \rightarrow (size(P') + i', s', stk')$

**unfolding** *exec1\_def*

**by** (*auto simp del: iexec.simps*)

**lemma** *exec\_appendL*:

**fixes**  $i\ i' :: int$

**shows**

$P \vdash (i, s, stk) \rightarrow^* (i', s', stk') \implies$

$P' @ P \vdash (size(P') + i, s, stk) \rightarrow^* (size(P') + i', s', stk')$

**by** (*induction rule: exec\_induct*) (*blast intro: star.step exec1\_appendL*)+

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 we have just executed the first instruction of the program, drop it:

**lemma** *exec\_Cons\_1* [*intro*]:

$P \vdash (0, s, stk) \rightarrow^* (j, t, stk') \implies$

$instr \# P \vdash (1, s, stk) \rightarrow^* (1+j, t, stk')$

**by** (*drule exec\_appendL[where P'=[instr]]*) *simp*

```

lemma exec_appendL_if[intro]:
  fixes i i' j :: int
  shows
    size P' <= i
     $\implies P \vdash (i - \text{size } P', s, \text{stk}) \rightarrow^* (j, s', \text{stk}')$ 
     $\implies i' = \text{size } P' + j$ 
     $\implies P' @ P \vdash (i, s, \text{stk}) \rightarrow^* (i', s', \text{stk}')$ 
by (drule exec_appendL[where  $P'=P'$ ]) simp

```

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

```

lemma exec_append_trans[intro]:
  fixes i' i'' j'' :: int
  shows
     $P \vdash (0, s, \text{stk}) \rightarrow^* (i', s', \text{stk}') \implies$ 
     $\text{size } P \leq i' \implies$ 
     $P' \vdash (i' - \text{size } P, s', \text{stk}') \rightarrow^* (i'', s'', \text{stk}'') \implies$ 
     $j'' = \text{size } P + i''$ 
     $\implies$ 
     $P @ P' \vdash (0, s, \text{stk}) \rightarrow^* (j'', s'', \text{stk}'')$ 
by(metis star_trans[OF exec_appendR exec_appendL_if])

```

```

declare Let_def[simp]

```

## 5.4 Compilation

```

fun acomp :: aexp  $\Rightarrow$  instr list where
  acomp (N n) = [LOADI n] |
  acomp (V x) = [LOAD x] |
  acomp (Plus a1 a2) = acomp a1 @ acomp a2 @ [ADD]

```

```

lemma acomp_correct[intro]:
   $\text{acomp } a \vdash (0, s, \text{stk}) \rightarrow^* (\text{size}(\text{acomp } a), s, \text{aval } a \text{ } s \# \text{stk})$ 
by (induction a arbitrary: stk) fastforce +

```

```

fun bcomp :: bexp  $\Rightarrow$  bool  $\Rightarrow$  int  $\Rightarrow$  instr list where
  bcomp (Bc v) f n = (if v=f then [JMP n] else []) |
  bcomp (Not b) f n = bcomp b ( $\neg$ f) n |
  bcomp (And b1 b2) f n =
    (let cb2 = bcomp b2 f n;
     m = if f then size cb2 else (size cb2::int)+n;
     cb1 = bcomp b1 False m
     in cb1 @ cb2) |

```

*bcomp* (*Less* *a1 a2*) *f n* =  
*acomp a1 @ acomp a2 @ (if f then [JMPLESS *n*] else [JMPGE *n*])*

**value**

*bcomp* (*And* (*Less* (*V "x"*) (*V "y"*)) (*Not*(*Less* (*V "u"*) (*V "v"*))))  
*False 3*

**lemma** *bcomp\_correct*[*intro*]:

**fixes** *n :: int*

**shows**

$0 \leq n \implies$

*bcomp b f n*  $\vdash$

(*0,s,stk*)  $\rightarrow^*$  (*size(bcomp b f n) + (if f = bval b s then n else 0),s,stk*)

**proof**(*induction b arbitrary: f n*)

**case** *Not*

**from** *Not*(1)[**where** *f = ~f*] *Not*(2) **show** ?*case by fastforce*

**next**

**case** (*And b1 b2*)

**from** *And*(1)[*of if f then size(bcomp b2 f n) else size(bcomp b2 f n) + n*  
*False*]

*And*(2)[*of n f*] *And*(3)

**show** ?*case by fastforce*

**qed** *fastforce+*

**fun** *ccomp* :: *com*  $\Rightarrow$  *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 @ JMP (size cc2) # cc2) |*

*ccomp* (*WHILE b DO c*) =

(*let cc = ccomp c; cb = bcomp b False (size cc + 1)*  
*in cb @ cc @ [JMP (-(size cb + size cc + 1))])*

**value** *ccomp*

(*IF Less* (*V "u"*) (*N 1*) *THEN "u" ::= Plus* (*V "u"*) (*N 1*)

*ELSE "v" ::= V "u"*)

**value** *ccomp* (*WHILE Less* (*V "u"*) (*N 1*) *DO ("u" ::= Plus* (*V "u"*) (*N 1*)))



## 5.5 Preservation of semantics

**lemma** *ccomp\_bigstep*:

```

(c,s) ⇒ t ⇒⇒ ccomp c ⊢ (0,s,stk) →* (size(ccomp c),t,stk)
proof(induction arbitrary: stk rule: big_step_induct)
  case (Assign x a s)
  show ?case by (fastforce simp:fun_upd_def cong: if_cong)
next
  case (Seq c1 s1 s2 c2 s3)
  let ?cc1 = ccomp c1 let ?cc2 = ccomp c2
  have ?cc1 @ ?cc2 ⊢ (0,s1,stk) →* (size ?cc1,s2,stk)
    using Seq.IH(1) by fastforce
  moreover
  have ?cc1 @ ?cc2 ⊢ (size ?cc1,s2,stk) →* (size(?cc1 @ ?cc2),s3,stk)
    using Seq.IH(2) by fastforce
  ultimately show ?case by simp (blast intro: star_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,s1,stk)
    using (bval b s1) by fastforce
  moreover
  have ?cw ⊢ (size ?cb,s1,stk) →* (size ?cb + size ?cc,s2,stk)
    using WhileTrue.IH(1) by fastforce
  moreover
  have ?cw ⊢ (size ?cb + size ?cc,s2,stk) →* (0,s2,stk)
    by fastforce
  moreover
  have ?cw ⊢ (0,s2,stk) →* (size ?cw,s3,stk) by(rule WhileTrue.IH(2))
  ultimately show ?case by(blast intro: star_trans)
qed fastforce+

```

**end**

## 6 Compiler Correctness, Reverse Direction

```

theory Compiler2
imports Compiler
begin

```

The preservation of the source code semantics is already shown in the parent theory *Compiler*. This here shows the second direction.

## 6.1 Definitions

Execution in  $n$  steps for simpler induction

**primrec**

$exec\_n :: instr\ list \Rightarrow config \Rightarrow nat \Rightarrow config \Rightarrow bool$   
 $(-/ \vdash (- \rightarrow \hat{\ } / -) [65,0,1000,55] 55)$

**where**

$P \vdash c \rightarrow \hat{\ } 0\ c' = (c' = c) \mid$   
 $P \vdash c \rightarrow \hat{\ } (Suc\ n)\ c'' = (\exists\ c'. (P \vdash c \rightarrow c') \wedge P \vdash c' \rightarrow \hat{\ } n\ c'')$

The possible successor PCs of an instruction at position  $n$

**definition**  $isuccs :: instr \Rightarrow int \Rightarrow int\ set$  **where**

$isuccs\ i\ n = (case\ i\ of$   
 $\ JMP\ j \Rightarrow \{n + 1 + j\} \mid$   
 $\ JMPLESS\ j \Rightarrow \{n + 1 + j, n + 1\} \mid$   
 $\ JMPGE\ j \Rightarrow \{n + 1 + j, n + 1\} \mid$   
 $\ - \Rightarrow \{n + 1\})$

The possible successors PCs of an instruction list

**definition**  $succs :: instr\ list \Rightarrow int \Rightarrow int\ set$  **where**

$succs\ P\ n = \{s. \exists\ i::int. 0 \leq i \wedge i < size\ P \wedge s \in isuccs\ (P!!i)\ (n+i)\}$

Possible exit PCs of a program

**definition**  $exits :: instr\ list \Rightarrow int\ set$  **where**

$exits\ P = succs\ P\ 0 - \{0..< size\ P\}$

## 6.2 Basic properties of $exec\_n$

**lemma**  $exec\_n\_exec$ :

$P \vdash c \rightarrow \hat{\ } n\ c' \Longrightarrow P \vdash c \rightarrow^* c'$   
**by** ( $induct\ n\ arbitrary: c$ ) ( $auto\ intro: star.step$ )

**lemma**  $exec\_0$  [ $intro!$ ]:  $P \vdash c \rightarrow \hat{\ } 0\ c$  **by**  $simp$

**lemma**  $exec\_Suc$ :

$\llbracket P \vdash c \rightarrow c'; P \vdash c' \rightarrow \hat{\ } n\ c'' \rrbracket \Longrightarrow P \vdash c \rightarrow \hat{\ } (Suc\ n)\ c''$   
**by** ( $fastforce\ simp\ del: split\_paired.Ex$ )

**lemma**  $exec\_exec\_n$ :

$P \vdash c \rightarrow^* c' \Longrightarrow \exists\ n. P \vdash c \rightarrow \hat{\ } n\ c'$   
**by** ( $induct\ rule: star.induct$ ) ( $auto\ intro: exec\_Suc$ )

**lemma**  $exec\_eq\_exec\_n$ :

$(P \vdash c \rightarrow^* c') = (\exists\ n. P \vdash c \rightarrow \hat{\ } n\ c')$

by (*blast intro: exec\_exec\_n exec\_n\_exec*)

**lemma** *exec\_n\_Nil* [*simp*]:

$\square \vdash c \rightarrow \hat{k} c' = (c' = c \wedge k = 0)$

by (*induct k*) (*auto simp: exec1\_def*)

**lemma** *exec1\_exec\_n* [*intro!*]:

$P \vdash c \rightarrow c' \implies P \vdash c \rightarrow \hat{1} c'$

by (*cases c'*) *simp*

### 6.3 Concrete symbolic execution steps

**lemma** *exec\_n\_step*:

$n \neq n' \implies$

$P \vdash (n, stk, s) \rightarrow \hat{k} (n', stk', s') =$

$(\exists c. P \vdash (n, stk, s) \rightarrow c \wedge P \vdash c \rightarrow \hat{(k-1)} (n', stk', s') \wedge 0 < k)$

by (*cases k*) *auto*

**lemma** *exec1\_end*:

$size P \leq fst c \implies \neg P \vdash c \rightarrow c'$

by (*auto simp: exec1\_def*)

**lemma** *exec\_n\_end*:

$size P \leq (n::int) \implies$

$P \vdash (n, s, stk) \rightarrow \hat{k} (n', s', stk') = (n' = n \wedge stk' = stk \wedge s' = s \wedge k = 0)$

by (*cases k*) (*auto simp: exec1\_end*)

**lemmas** *exec\_n\_simps* = *exec\_n\_step exec\_n\_end*

### 6.4 Basic properties of *succs*

**lemma** *succs\_simps* [*simp*]:

*succs* [*ADD*]  $n = \{n + 1\}$

*succs* [*LOADI v*]  $n = \{n + 1\}$

*succs* [*LOAD x*]  $n = \{n + 1\}$

*succs* [*STORE x*]  $n = \{n + 1\}$

*succs* [*JMP i*]  $n = \{n + 1 + i\}$

*succs* [*JMPGE i*]  $n = \{n + 1 + i, n + 1\}$

*succs* [*JMPLESS i*]  $n = \{n + 1 + i, n + 1\}$

by (*auto simp: succs\_def isuccs\_def*)

**lemma** *succs\_empty* [*iff*]: *succs*  $\square$   $n = \{\}$

by (*simp add: succs\_def*)

**lemma** *succs.Cons*:

$\text{succs } (x\#xs) \ n = \text{isuccs } x \ n \cup \text{succs } xs \ (1+n)$  (**is**  $\_ = ?x \cup ?xs$ )

**proof**

**let**  $?isuccs = \lambda p \ P \ n \ i::\text{int}. \ 0 \leq i \wedge i < \text{size } P \wedge p \in \text{isuccs } (P!!i) \ (n+i)$

**have**  $p \in ?x \cup ?xs$  **if** *assm*:  $p \in \text{succs } (x\#xs) \ n$  **for**  $p$

**proof** –

**from** *assm* **obtain**  $i::\text{int}$  **where** *isuccs*:  $?isuccs \ p \ (x\#xs) \ n \ i$

**unfolding** *succs\_def* **by** *auto*

**show** *?thesis*

**proof** *cases*

**assume**  $i = 0$  **with** *isuccs* **show** *?thesis* **by** *simp*

**next**

**assume**  $i \neq 0$

**with** *isuccs*

**have**  $?isuccs \ p \ xs \ (1+n) \ (i - 1)$  **by** *auto*

**hence**  $p \in ?xs$  **unfolding** *succs\_def* **by** *blast*

**thus** *?thesis* ..

**qed**

**qed**

**thus**  $\text{succs } (x\#xs) \ n \subseteq ?x \cup ?xs$  ..

**have**  $p \in \text{succs } (x\#xs) \ n$  **if** *assm*:  $p \in ?x \vee p \in ?xs$  **for**  $p$

**proof** –

**from** *assm* **show** *?thesis*

**proof**

**assume**  $p \in ?x$  **thus** *?thesis* **by** (*fastforce simp: succs\_def*)

**next**

**assume**  $p \in ?xs$

**then obtain**  $i$  **where**  $?isuccs \ p \ xs \ (1+n) \ i$

**unfolding** *succs\_def* **by** *auto*

**hence**  $?isuccs \ p \ (x\#xs) \ n \ (1+i)$

**by** (*simp add: algebra\_simps*)

**thus** *?thesis* **unfolding** *succs\_def* **by** *blast*

**qed**

**qed**

**thus**  $?x \cup ?xs \subseteq \text{succs } (x\#xs) \ n$  **by** *blast*

**qed**

**lemma** *succs\_iexec1*:

**assumes**  $c' = \text{iexec } (P!!i) \ (i,s,stk) \ 0 \leq i \ i < \text{size } P$

**shows**  $\text{fst } c' \in \text{succs } P \ 0$

**using** *assms* **by** (*auto simp: succs\_def isuccs\_def split: instr.split*)

**lemma** *succs\_shift*:

$(p - n \in \text{succs } P \ 0) = (p \in \text{succs } P \ n)$   
**by** (*fastforce simp: succs\_def isuccs\_def split: instr.split*)

**lemma** *inj\_op\_plus* [*simp*]:  
*inj* ((+) (*i::int*))  
**by** (*metis add\_minus\_cancel inj\_on\_inverseI*)

**lemma** *succs\_set\_shift* [*simp*]:  
 (+) *i* ‘ *succs xs 0 = succs xs i*  
**by** (*force simp: succs\_shift [where n=i, symmetric] intro: set\_eqI*)

**lemma** *succs\_append* [*simp*]:  
*succs (xs @ ys) n = succs xs n  $\cup$  succs ys (n + size xs)*  
**by** (*induct xs arbitrary: n*) (*auto simp: succs\_Cons algebra\_simps*)

**lemma** *exits\_append* [*simp*]:  
*exits (xs @ ys) = exits xs  $\cup$  ((+) (size xs)) ‘ exits ys –*  
 $\{0..<size\ xs + size\ ys\}$   
**by** (*auto simp: exits\_def image\_set\_diff*)

**lemma** *exits\_single*:  
*exits [x] = isuccs x 0 – {0}*  
**by** (*auto simp: exits\_def succs\_def*)

**lemma** *exits\_Cons*:  
*exits (x # xs) = (isuccs x 0 – {0})  $\cup$  ((+) 1) ‘ exits xs –*  
 $\{0..<1 + size\ xs\}$   
**using** *exits\_append* [*of [x] xs*]  
**by** (*simp add: exits\_single*)

**lemma** *exits\_empty* [*iff*]: *exits [] = {}* **by** (*simp add: exits\_def*)

**lemma** *exits\_simps* [*simp*]:  
*exits [ADD] = {1}*  
*exits [LOADI v] = {1}*  
*exits [LOAD x] = {1}*  
*exits [STORE x] = {1}*  
*i  $\neq$  -1  $\implies$  exits [JMP i] = {1 + i}*  
*i  $\neq$  -1  $\implies$  exits [JMPGE i] = {1 + i, 1}*  
*i  $\neq$  -1  $\implies$  exits [JMPLESS i] = {1 + i, 1}*  
**by** (*auto simp: exits\_def*)

**lemma** *acompsuccs* [*simp*]:

$succs (acompa a) n = \{n + 1 .. n + size (acompa a)\}$   
**by** (*induct a arbitrary: n*) *auto*

**lemma** *acompa\_size*:  
 $(1::int) \leq size (acompa a)$   
**by** (*induct a*) *auto*

**lemma** *acompa\_exits* [*simp*]:  
 $exits (acompa a) = \{size (acompa a)\}$   
**by** (*auto simp: exits\_def acompa\_size*)

**lemma** *bcomp\_succs*:  
 $0 \leq i \implies$   
 $succs (bcomp b f i) n \subseteq \{n .. n + size (bcomp b f i)\} \cup \{n + i + size (bcomp b f i)\}$

**proof** (*induction b arbitrary: f i n*)  
**case** (*And b1 b2*)  
**from** *And.prem1*  
**show** *?case*  
**by** (*cases f*)  
*(auto dest: And.IH(1) [THEN subsetD, rotated]*  
*And.IH(2) [THEN subsetD, rotated])*

**qed** *auto*

**lemmas** *bcomp\_succsD* [*dest!*] = *bcomp\_succs* [*THEN subsetD, rotated*]

**lemma** *bcomp\_exits*:  
**fixes**  $i :: int$   
**shows**  
 $0 \leq i \implies$   
 $exits (bcomp b f i) \subseteq \{size (bcomp b f i), i + size (bcomp b f i)\}$   
**by** (*auto simp: exits\_def*)

**lemma** *bcomp\_exitsD* [*dest!*]:  
 $p \in exits (bcomp b f i) \implies 0 \leq i \implies$   
 $p = size (bcomp b f i) \vee p = i + size (bcomp b f i)$   
**using** *bcomp\_exits* **by** *auto*

**lemma** *ccomp\_succs*:  
 $succs (ccomp c) n \subseteq \{n..n + size (ccomp c)\}$   
**proof** (*induction c arbitrary: n*)  
**case** *SKIP* **thus** *?case* **by** *simp*  
**next**  
**case** *Assign* **thus** *?case* **by** *simp*

```

next
  case (Seq c1 c2)
  from Seq.premis
  show ?case
    by (fastforce dest: Seq.IH [THEN subsetD])
next
  case (If b c1 c2)
  from If.premis
  show ?case
    by (auto dest!: If.IH [THEN subsetD] simp: isuccs_def succs_Cons)
next
  case (While b c)
  from While.premis
  show ?case by (auto dest!: While.IH [THEN subsetD])
qed

```

```

lemma ccomp_exits:
  exits (ccomp c)  $\subseteq$  {size (ccomp c)}
  using ccomp_succs [of c 0] by (auto simp: exits_def)

```

```

lemma ccomp_exitsD [dest!]:
   $p \in \text{exits } (\text{ccomp } c) \implies p = \text{size } (\text{ccomp } c)$ 
  using ccomp_exits by auto

```

## 6.5 Splitting up machine executions

```

lemma exec1_split:
  fixes i j :: int
  shows
     $P @ c @ P' \vdash (\text{size } P + i, s) \rightarrow (j, s') \implies 0 \leq i \implies i < \text{size } c \implies$ 
     $c \vdash (i, s) \rightarrow (j - \text{size } P, s')$ 
  by (auto split: instr_splits simp: exec1_def)

```

```

lemma exec_n_split:
  fixes i j :: int
  assumes  $P @ c @ P' \vdash (\text{size } P + i, s) \rightarrow \hat{n} (j, s')$ 
     $0 \leq i & i < \text{size } c$ 
     $j \notin \{\text{size } P ..< \text{size } P + \text{size } c\}$ 
  shows  $\exists s'' (i'::int) k m.$ 
     $c \vdash (i, s) \rightarrow \hat{k} (i', s'') \wedge$ 
     $i' \in \text{exits } c \wedge$ 
     $P @ c @ P' \vdash (\text{size } P + i', s'') \rightarrow \hat{m} (j, s') \wedge$ 
     $n = k + m$ 
  using assms proof (induction n arbitrary: i j s)

```

```

case 0
thus ?case by simp
next
case (Suc n)
have i: 0 ≤ i i < size c by fact+
from Suc.prem.s
have j: ¬ (size P ≤ j ∧ j < size P + size c) by simp
from Suc.prem.s
obtain i0 s0 where
  step: P @ c @ P' ⊢ (size P + i, s) → (i0, s0) and
  rest: P @ c @ P' ⊢ (i0, s0) → ^n (j, s')
by clarsimp

from step i
have c: c ⊢ (i, s) → (i0 - size P, s0) by (rule exec1_split)

have i0 = size P + (i0 - size P) by simp
then obtain j0::int where j0: i0 = size P + j0 ..

note split_paired_Ex [simp del]

have ?case if assm: j0 ∈ {0 ..< size c}
proof -
  from assm j0 j rest c show ?case
  by (fastforce dest!: Suc.IH intro!: exec_Suc)
qed
moreover
have ?case if assm: j0 ∉ {0 ..< size c}
proof -
  from c j0 have j0 ∈ succs c 0
  by (auto dest: succs_iexec1 simp: exec1_def simp del: iexec.simps)
  with assm have j0 ∈ exits c by (simp add: exits_def)
  with c j0 rest show ?case by fastforce
qed
ultimately
show ?case by cases
qed

lemma exec_n_drop_right:
fixes j :: int
assumes c @ P' ⊢ (0, s) → ^n (j, s') j ∉ {0..<size c}
shows ∃ s'' i' k m.
  (if c = [] then s'' = s ∧ i' = 0 ∧ k = 0
   else c ⊢ (0, s) → ^k (i', s'') ∧

```



$$i' \in \text{exits } c) \wedge$$

$$c @ P' \vdash (i', s'') \rightarrow \hat{m}(j, s') \wedge$$

$$n = k + m$$

**using** *assms*

**by** (*cases*  $c = []$ )

(*auto dest: exec\_n\_split [where P=[], simplified]*)

Dropping the left context of a potentially incomplete execution of  $c$ .

**lemma** *exec1\_drop\_left*:

**fixes**  $i \ n :: \text{int}$

**assumes**  $P1 @ P2 \vdash (i, s, stk) \rightarrow (n, s', stk')$  **and**  $\text{size } P1 \leq i$

**shows**  $P2 \vdash (i - \text{size } P1, s, stk) \rightarrow (n - \text{size } P1, s', stk')$

**proof** –

**have**  $i = \text{size } P1 + (i - \text{size } P1)$  **by** *simp*

**then obtain**  $i' :: \text{int}$  **where**  $i = \text{size } P1 + i' ..$

**moreover**

**have**  $n = \text{size } P1 + (n - \text{size } P1)$  **by** *simp*

**then obtain**  $n' :: \text{int}$  **where**  $n = \text{size } P1 + n' ..$

**ultimately**

**show** *?thesis* **using** *assms*

**by** (*clarsimp simp: exec1\_def simp del: iexec.simps*)

**qed**

**lemma** *exec\_n\_drop\_left*:

**fixes**  $i \ n :: \text{int}$

**assumes**  $P @ P' \vdash (i, s, stk) \rightarrow \hat{k}(n, s', stk')$

$\text{size } P \leq i$  *exits*  $P' \subseteq \{0..\}$

**shows**  $P' \vdash (i - \text{size } P, s, stk) \rightarrow \hat{k}(n - \text{size } P, s', stk')$

**using** *assms* **proof** (*induction k arbitrary: i s stk*)

**case** 0 **thus** *?case* **by** *simp*

**next**

**case** (*Suc k*)

**from** *Suc.prem*s

**obtain**  $i' \ s'' \ stk''$  **where**

*step*:  $P @ P' \vdash (i, s, stk) \rightarrow (i', s'', stk'')$  **and**

*rest*:  $P @ P' \vdash (i', s'', stk'') \rightarrow \hat{k}(n, s', stk')$

**by** *auto*

**from** *step* ( $\text{size } P \leq i$ )

**have**  $*$ :  $P' \vdash (i - \text{size } P, s, stk) \rightarrow (i' - \text{size } P, s'', stk'')$

**by** (*rule exec1\_drop\_left*)

**then have**  $i' - \text{size } P \in \text{succs } P' \ 0$

**by** (*fastforce dest!: succs\_iexec1 simp: exec1\_def simp del: iexec.simps*)

**with** (*exits*  $P' \subseteq \{0..\}$ )

**have**  $\text{size } P \leq i'$  **by** (*auto simp: exits\_def*)

**from** *rest this*  $\langle \text{exits } P' \subseteq \{0..\} \rangle$   
**have**  $P' \vdash (i' - \text{size } P, s'', \text{stk}'') \rightarrow \hat{k} (n - \text{size } P, s', \text{stk}')$   
**by** (*rule Suc.IH*)  
**with** \* **show** ?*case* **by** *auto*  
**qed**

**lemmas** *exec\_n\_drop\_Cons* =  
*exec\_n\_drop\_left* [**where**  $P=[\text{instr}]$ , *simplified*] **for** *instr*

**definition**  
*closed*  $P \longleftrightarrow \text{exits } P \subseteq \{\text{size } P\}$

**lemma** *ccomp\_closed* [*simp*, *intro!*]: *closed* (*ccomp*  $c$ )  
**using** *ccomp\_exits* **by** (*auto simp: closed\_def*)

**lemma** *acompl\_closed* [*simp*, *intro!*]: *closed* (*acompl*  $c$ )  
**by** (*simp add: closed\_def*)

**lemma** *exec\_n\_split\_full*:  
**fixes**  $j :: \text{int}$   
**assumes** *exec*:  $P @ P' \vdash (0, s, \text{stk}) \rightarrow \hat{k} (j, s', \text{stk}')$   
**assumes**  $P$ :  $\text{size } P \leq j$   
**assumes** *closed*: *closed*  $P$   
**assumes** *exits*:  $\text{exits } P' \subseteq \{0..\}$   
**shows**  $\exists k1 \ k2 \ s'' \ \text{stk}'' . P \vdash (0, s, \text{stk}) \rightarrow \hat{k}1 (\text{size } P, s'', \text{stk}'') \wedge$   
 $P' \vdash (0, s'', \text{stk}'') \rightarrow \hat{k}2 (j - \text{size } P, s', \text{stk}')$

**proof** (*cases*  $P$ )  
**case** *Nil* **with** *exec*  
**show** ?*thesis* **by** *fastforce*  
**next**  
**case** *Cons*  
**hence**  $0 < \text{size } P$  **by** *simp*  
**with** *exec*  $P$  *closed*  
**obtain**  $k1 \ k2 \ s'' \ \text{stk}''$  **where**  
 $1$ :  $P \vdash (0, s, \text{stk}) \rightarrow \hat{k}1 (\text{size } P, s'', \text{stk}'')$  **and**  
 $2$ :  $P @ P' \vdash (\text{size } P, s'', \text{stk}'') \rightarrow \hat{k}2 (j, s', \text{stk}')$   
**by** (*auto dest!: exec\_n\_split* [**where**  $P=[]$  **and**  $i=0$ , *simplified*]  
*simp: closed\_def*)  
**moreover**  
**have**  $j = \text{size } P + (j - \text{size } P)$  **by** *simp*  
**then obtain**  $j0 :: \text{int}$  **where**  $j = \text{size } P + j0$  ..  
**ultimately**  
**show** ?*thesis* **using** *exits*  
**by** (*fastforce dest: exec\_n\_drop\_left*)

qed

## 6.6 Correctness theorem

**lemma** *acom\_neq\_Nil* [*simp*]:

*acom* *a*  $\neq$  []  
**by** (*induct a*) *auto*

**lemma** *acom\_exec\_n* [*dest!*]:

*acom* *a*  $\vdash (0, s, stk) \rightarrow \hat{n} (size (acom a), s', stk') \implies$   
 $s' = s \wedge stk' = aval a s \# stk$

**proof** (*induction a arbitrary: n s' stk stk'*)

**case** (*Plus a1 a2*)

**let** *?sz* = *size (acom a1)* + (*size (acom a2)* + 1)

**from** *Plus.prem*s

**have** *acom a1* @ *acom a2* @ [*ADD*]  $\vdash (0, s, stk) \rightarrow \hat{n} (?sz, s', stk')$   
**by** (*simp add: algebra\_simps*)

**then obtain** *n1 s1 stk1 n2 s2 stk2 n3* **where**

*acom a1*  $\vdash (0, s, stk) \rightarrow \hat{n1} (size (acom a1), s1, stk1)$

*acom a2*  $\vdash (0, s1, stk1) \rightarrow \hat{n2} (size (acom a2), s2, stk2)$

[*ADD*]  $\vdash (0, s2, stk2) \rightarrow \hat{n3} (1, s', stk')$

**by** (*auto dest!: exec\_n\_split\_full*)

**thus** *?case* **by** (*fastforce dest: Plus.IH simp: exec\_n\_simps exec1\_def*)

qed (*auto simp: exec\_n\_simps exec1\_def*)

**lemma** *bcomp\_split*:

**fixes** *i j* :: *int*

**assumes** *bcomp b f i* @ *P'*  $\vdash (0, s, stk) \rightarrow \hat{n} (j, s', stk')$

$j \notin \{0..<size (bcomp b f i)\} \ 0 \leq i$

**shows**  $\exists s'' stk'' (i'::int) k m.$

*bcomp b f i*  $\vdash (0, s, stk) \rightarrow \hat{k} (i', s'', stk'') \wedge$

$(i' = size (bcomp b f i) \vee i' = i + size (bcomp b f i)) \wedge$

*bcomp b f i* @ *P'*  $\vdash (i', s'', stk'') \rightarrow \hat{m} (j, s', stk')$   $\wedge$

$n = k + m$

**using** *assms* **by** (*cases bcomp b f i = []*) (*fastforce dest!: exec\_n\_drop\_right*) $+$

**lemma** *bcomp\_exec\_n* [*dest*]:

**fixes** *i j* :: *int*

**assumes** *bcomp b f j*  $\vdash (0, s, stk) \rightarrow \hat{n} (i, s', stk')$

$size (bcomp b f j) \leq i \ 0 \leq j$

**shows**  $i = size (bcomp b f j) + (if f = bval b s then j else 0) \wedge$

$s' = s \wedge stk' = stk$

```

using assms proof (induction b arbitrary: f j i n s' stk')
  case Bc thus ?case
    by (simp split: if_split_asm add: exec_n_simps exec1_def)
next
  case (Not b)
  from Not.prems show ?case
    by (fastforce dest!: Not.IH)
next
  case (And b1 b2)

  let ?b2 = bcomp b2 f j
  let ?m = if f then size ?b2 else size ?b2 + j
  let ?b1 = bcomp b1 False ?m

  have j: size (bcomp (And b1 b2) f j) ≤ i 0 ≤ j by fact+

  from And.prems
  obtain s'' stk'' and i'::int and k m where
    b1: ?b1 ⊢ (0, s, stk) →^k (i', s'', stk'')
    i' = size ?b1 ∨ i' = ?m + size ?b1 and
    b2: ?b2 ⊢ (i' - size ?b1, s'', stk'') →^m (i - size ?b1, s', stk')
    by (auto dest!: bcomp_split dest: exec_n_drop_left)
  from b1 j
  have i' = size ?b1 + (if ¬bval b1 s then ?m else 0) ∧ s'' = s ∧ stk'' =
stk
    by (auto dest!: And.IH)
  with b2 j
  show ?case
    by (fastforce dest!: And.IH simp: exec_n_end split: if_split_asm)
next
  case Less
  thus ?case by (auto dest!: exec_n_split_full simp: exec_n_simps exec1_def)

```

qed

```

lemma ccomp_empty [elim!]:
  ccomp c = [] ⇒ (c,s) ⇒ s
  by (induct c) auto

```

```

declare assign_simp [simp]

```

```

lemma ccomp_exec_n:
  ccomp c ⊢ (0,s,stk) →^n (size(ccomp c),t,stk')
   $\implies (c,s) \Rightarrow t \wedge stk' = stk$ 

```

```

proof (induction c arbitrary: s t stk stk' n)
  case SKIP
  thus ?case by auto
next
  case (Assign x a)
  thus ?case
    by simp (fastforce dest!: exec_n_split_full simp: exec_n_simps exec1_def)
next
  case (Seq c1 c2)
  thus ?case by (fastforce dest!: exec_n_split_full)
next
  case (If b c1 c2)
  note If.IH [dest!]

  let ?if = IF b THEN c1 ELSE c2
  let ?cs = ccomp ?if
  let ?bcomp = bcomp b False (size (ccomp c1) + 1)

  from (⟨?cs ⊢ (0,s,stk) →^n (size ?cs,t,stk')⟩)
  obtain i' :: int and k m s'' stk'' where
    cs: ?cs ⊢ (i',s'',stk'') →^m (size ?cs,t,stk') and
    ?bcomp ⊢ (0,s,stk) →^k (i', s'', stk'')
    i' = size ?bcomp ∨ i' = size ?bcomp + size (ccomp c1) + 1
  by (auto dest!: bcomp_split)

  hence i':
    s''=s stk'' = stk
    i' = (if bval b s then size ?bcomp else size ?bcomp+size(ccomp c1)+1)
  by auto

  with cs have cs':
    ccomp c1@JMP (size (ccomp c2))#ccomp c2 ⊢
      (if bval b s then 0 else size (ccomp c1)+1, s, stk) →^m
      (1 + size (ccomp c1) + size (ccomp c2), t, stk')
    by (fastforce dest: exec_n_drop_left simp: exits_Cons isuccs_def alge-
bra_simps)

  show ?case
  proof (cases bval b s)
    case True with cs'
    show ?thesis
    by simp
      (fastforce dest: exec_n_drop_right
split: if_split_asm)

```

```

      simp: exec_n_simps exec1_def)
next
  case False with cs'
  show ?thesis
    by (auto dest!: exec_n_drop_Cons exec_n_drop_left
        simp: exits_Cons isucss_def)
qed
next
  case (While b c)

  from While.prems
  show ?case
  proof (induction n arbitrary: s rule: nat_less_induct)
    case (1 n)

    have ?case if assm:  $\neg \text{bval } b \ s$ 
    proof -
      from assm 1.prem
      show ?case
        by simp (fastforce dest!: bcomp_split simp: exec_n_simps)
    qed
  moreover
  have ?case if b:  $\text{bval } b \ s$ 
  proof -
    let ?c0 = WHILE b DO c
    let ?cs = ccomp ?c0
    let ?bs = bcomp b False (size (ccomp c) + 1)
    let ?jmp = [JMP ( $\neg((\text{size } ?bs + \text{size } (\text{ccomp } c) + 1))$ )]

    from 1.prem b
    obtain k where
      cs: ?cs  $\vdash (\text{size } ?bs, s, \text{stk}) \rightarrow \hat{k} (\text{size } ?cs, t, \text{stk}')$  and
      k:  $k \leq n$ 
    by (fastforce dest!: bcomp_split)

    show ?case
  proof cases
    assume ccomp c = []
    with cs k
    obtain m where
      ?cs  $\vdash (0, s, \text{stk}) \rightarrow \hat{m} (\text{size } (\text{ccomp } ?c0), t, \text{stk}')$ 
       $m < n$ 
    by (auto simp: exec_n_step [where  $k=k$ ] exec1_def)
  with 1.IH

```

```

    show ?case by blast
next
  assume ccomp c ≠ []
  with cs
  obtain m m' s'' stk'' where
    c: ccomp c ⊢ (0, s, stk) → ^m' (size (ccomp c), s'', stk'') and
    rest: ?cs ⊢ (size ?bs + size (ccomp c), s'', stk'') → ^m
      (size ?cs, t, stk') and
    m: k = m + m'
  by (auto dest: exec_n_split [where i=0, simplified])
  from c
  have (c,s) ⇒ s'' and stk: stk'' = stk
  by (auto dest!: While.IH)
  moreover
  from rest m k stk
  obtain k' where
    ?cs ⊢ (0, s'', stk) → ^k' (size ?cs, t, stk')
    k' < n
  by (auto simp: exec_n_step [where k=m] exec1_def)
  with 1.IH
  have (?c0, s'') ⇒ t ∧ stk' = stk by blast
  ultimately
  show ?case using b by blast
qed
qed
ultimately show ?case by cases
qed
qed

```

**theorem** *ccomp\_exec*:

$ccomp\ c \vdash (0, s, stk) \rightarrow^* (size(ccomp\ c), t, stk') \implies (c, s) \Rightarrow t$   
 by (auto dest: exec\_exec\_n ccomp\_exec\_n)

**corollary** *ccomp\_sound*:

$ccomp\ c \vdash (0, s, stk) \rightarrow^* (size(ccomp\ c), t, stk) \iff (c, s) \Rightarrow t$   
 by (blast intro!: ccomp\_exec ccomp\_bigstep)

end

## 7 A Typed Language

**theory** *Types* imports *Star Complex\_Main* begin

We build on *Complex\_Main* instead of *Main* to access the real numbers.

## 7.1 Arithmetic Expressions

**datatype** *val* = *Iv int* | *Rv real*

**type\_synonym** *vname* = *string*

**type\_synonym** *state* = *vname*  $\Rightarrow$  *val*  
**datatype** *aexp* = *Ic int* | *Rc real* |  
*V vname* | *Plus aexp aexp*

**inductive** *taval* :: *aexp*  $\Rightarrow$  *state*  $\Rightarrow$  *val*  $\Rightarrow$  *bool* **where**

*taval* (*Ic i*) *s* (*Iv i*) |  
*taval* (*Rc r*) *s* (*Rv r*) |  
*taval* (*V x*) *s* (*s x*) |  
*taval* *a1 s* (*Iv i1*)  $\Longrightarrow$  *taval* *a2 s* (*Iv i2*)  
 $\Longrightarrow$  *taval* (*Plus a1 a2*) *s* (*Iv(i1+i2)*) |  
*taval* *a1 s* (*Rv r1*)  $\Longrightarrow$  *taval* *a2 s* (*Rv r2*)  
 $\Longrightarrow$  *taval* (*Plus a1 a2*) *s* (*Rv(r1+r2)*)

**inductive\_cases** [*elim!*]:

*taval* (*Ic i*) *s v* *taval* (*Rc i*) *s v*  
*taval* (*V x*) *s v*  
*taval* (*Plus a1 a2*) *s v*

## 7.2 Boolean Expressions

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

**inductive** *tbval* :: *bexp*  $\Rightarrow$  *state*  $\Rightarrow$  *bool*  $\Rightarrow$  *bool* **where**

*tbval* (*Bc v*) *s v* |  
*tbval* *b s bv*  $\Longrightarrow$  *tbval* (*Not b*) *s* ( $\neg$  *bv*) |  
*tbval* *b1 s bv1*  $\Longrightarrow$  *tbval* *b2 s bv2*  $\Longrightarrow$  *tbval* (*And b1 b2*) *s* (*bv1* & *bv2*) |  
*taval* *a1 s* (*Iv i1*)  $\Longrightarrow$  *taval* *a2 s* (*Iv i2*)  $\Longrightarrow$  *tbval* (*Less a1 a2*) *s* (*i1* < *i2*)  
|  
*taval* *a1 s* (*Rv r1*)  $\Longrightarrow$  *taval* *a2 s* (*Rv r2*)  $\Longrightarrow$  *tbval* (*Less a1 a2*) *s* (*r1* < *r2*)

## 7.3 Syntax of Commands

**datatype**

*com* = *SKIP*  
| *Assign vname aexp* (*\_ ::= \_* [*1000*, *61*] *61*)  
| *Seq com com* (*\_;; \_* [*60*, *61*] *60*)  
| *If bexp com com* (*IF* - *THEN* - *ELSE* - [*0*, *0*, *61*] *61*)  
| *While bexp com* (*WHILE* - *DO* - [*0*, *61*] *61*)



## 7.4 Small-Step Semantics of Commands

**inductive**

$small\_step :: (com \times state) \Rightarrow (com \times state) \Rightarrow bool$  (**infix**  $\rightarrow 55$ )

**where**

$Assign: taval\ a\ s\ v \Longrightarrow (x ::= a, s) \rightarrow (SKIP, s(x := v)) \mid$

$Seq1: (SKIP;;c,s) \rightarrow (c,s) \mid$

$Seq2: (c1,s) \rightarrow (c1',s') \Longrightarrow (c1;;c2,s) \rightarrow (c1';;c2,s') \mid$

$IfTrue: tval\ b\ s\ True \Longrightarrow (IF\ b\ THEN\ c1\ ELSE\ c2,s) \rightarrow (c1,s) \mid$

$IfFalse: tval\ b\ s\ False \Longrightarrow (IF\ b\ THEN\ c1\ ELSE\ c2,s) \rightarrow (c2,s) \mid$

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

**lemmas**  $small\_step\_induct = small\_step.induct[split\_format(complete)]$

## 7.5 The Type System

**datatype**  $ty = Ity \mid Rty$

**type\_synonym**  $tyenv = vname \Rightarrow ty$

**inductive**  $atyping :: tyenv \Rightarrow aexp \Rightarrow ty \Rightarrow bool$

$((1\_ / \vdash / (- : / -)) [50,0,50] 50)$

**where**

$Ic\_ty: \Gamma \vdash Ic\ i : Ity \mid$

$Rc\_ty: \Gamma \vdash Rc\ r : Rty \mid$

$V\_ty: \Gamma \vdash V\ x : \Gamma\ x \mid$

$Plus\_ty: \Gamma \vdash a1 : \tau \Longrightarrow \Gamma \vdash a2 : \tau \Longrightarrow \Gamma \vdash Plus\ a1\ a2 : \tau$

**declare**  $atyping.intros [intro!]$

**inductive\_cases**  $[elim!]$ :

$\Gamma \vdash V\ x : \tau \ \Gamma \vdash Ic\ i : \tau \ \Gamma \vdash Rc\ r : \tau \ \Gamma \vdash Plus\ a1\ a2 : \tau$

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

**where**

$B\_ty: \Gamma \vdash Bc\ v \mid$

$Not\_ty: \Gamma \vdash b \Longrightarrow \Gamma \vdash Not\ b \mid$

$And\_ty: \Gamma \vdash b1 \Longrightarrow \Gamma \vdash b2 \Longrightarrow \Gamma \vdash And\ b1\ b2 \mid$

*Less\_ty*:  $\Gamma \vdash a1 : \tau \Longrightarrow \Gamma \vdash a2 : \tau \Longrightarrow \Gamma \vdash \text{Less } a1 \ a2$

**declare** *btyping.intros* [intro!]

**inductive\_cases** [elim!]:  $\Gamma \vdash \text{Not } b \ \Gamma \vdash \text{And } b1 \ b2 \ \Gamma \vdash \text{Less } a1 \ a2$

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

*Skip\_ty*:  $\Gamma \vdash \text{SKIP}$  |

*Assign\_ty*:  $\Gamma \vdash a : \Gamma(x) \Longrightarrow \Gamma \vdash x ::= a$  |

*Seq\_ty*:  $\Gamma \vdash c1 \Longrightarrow \Gamma \vdash c2 \Longrightarrow \Gamma \vdash c1;;c2$  |

*If\_ty*:  $\Gamma \vdash b \Longrightarrow \Gamma \vdash c1 \Longrightarrow \Gamma \vdash c2 \Longrightarrow \Gamma \vdash \text{IF } b \ \text{THEN } c1 \ \text{ELSE } c2$  |

*While\_ty*:  $\Gamma \vdash b \Longrightarrow \Gamma \vdash c \Longrightarrow \Gamma \vdash \text{WHILE } b \ \text{DO } c$

**declare** *ctyping.intros* [intro!]

**inductive\_cases** [elim!]:

$\Gamma \vdash x ::= a \ \Gamma \vdash c1;;c2$

$\Gamma \vdash \text{IF } b \ \text{THEN } c1 \ \text{ELSE } c2$

$\Gamma \vdash \text{WHILE } b \ \text{DO } c$

## 7.6 Well-typed Programs Do Not Get Stuck

**fun** *type* :: *val*  $\Rightarrow$  *ty* **where**

*type* (*Iv* *i*) = *Ity* |

*type* (*Rv* *r*) = *Rty*

**lemma** *type\_eq\_Ity*[simp]:  $\text{type } v = \text{Ity} \longleftrightarrow (\exists i. v = \text{Iv } i)$

**by** (*cases* *v*) *simp\_all*

**lemma** *type\_eq\_Rty*[simp]:  $\text{type } v = \text{Rty} \longleftrightarrow (\exists r. v = \text{Rv } r)$

**by** (*cases* *v*) *simp\_all*

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

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

**lemma** *apreservation*:

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

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

**apply** (*fastforce simp*: *styping\_def*)+

**done**

**lemma** *aprogess*:  $\Gamma \vdash a : \tau \Longrightarrow \Gamma \vdash s \Longrightarrow \exists v. \text{taval } a \ s \ v$

**proof**(*induction rule*: *atyping.induct*)

**case** (*Plus\_ty*  $\Gamma \ a1 \ t \ a2$ )

**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 v show ?thesis
    by(fastforce intro: taval.intros(4) dest!: apreservation)
next
  case Rv
  with Plus_ty v show ?thesis
    by(fastforce intro: taval.intros(5) dest!: apreservation)
qed
qed (auto intro: taval.intros)

```

```

lemma bprogress:  $\Gamma \vdash b \implies \Gamma \vdash s \implies \exists v. \text{tbval } b \ s \ v$ 
proof(induction rule: btyping.induct)
  case (Less_ty  $\Gamma \ a1 \ t \ a2$ )
  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 (fastforce intro!: tbval.intros(4) dest!: apreservation)
  next
    case Rv
    with Less_ty v show ?thesis
      by (fastforce intro!: tbval.intros(5) dest!: apreservation)
  qed
qed (auto intro: tbval.intros)

```

```

theorem progress:
   $\Gamma \vdash c \implies \Gamma \vdash s \implies c \neq \text{SKIP} \implies \exists cs'. (c,s) \rightarrow cs'$ 
proof(induction rule: ctyping.induct)
  case Skip_ty thus ?case by simp
next
  case Assign_ty
  thus ?case by (metis Assign aprogress)
next
  case Seq_ty thus ?case by simp (metis Seq1 Seq2)
next
  case (If_ty  $\Gamma \ b \ c1 \ c2$ )
  then obtain bv where tbval b s bv by (metis bprogress)
  show ?case
  proof(cases bv)
    assume bv
    with  $\langle \text{tbval } b \ s \ bv \rangle$  show ?case by simp (metis IfTrue)

```

```

next
  assume  $\neg bv$ 
  with  $\langle tbval\ b\ s\ bv \rangle$  show  $?case$  by simp (metis IfFalse)
qed
next
  case While_ty show  $?case$  by (metis While)
qed

theorem styping_preservation:
   $(c,s) \rightarrow (c',s') \implies \Gamma \vdash c \implies \Gamma \vdash s \implies \Gamma \vdash s'$ 
proof(induction rule: small_step_induct)
  case Assign thus  $?case$ 
  by (auto simp: styping_def) (metis Assign(1,3) apreservation)
qed auto

theorem ctyping_preservation:
   $(c,s) \rightarrow (c',s') \implies \Gamma \vdash c \implies \Gamma \vdash c'$ 
by (induct rule: small_step_induct) (auto simp: ctyping.intros)

abbreviation small_steps ::  $com * state \Rightarrow com * state \Rightarrow bool$  (infix  $\rightarrow^*$ 
55)
where  $x \rightarrow^* y == star\ small\_step\ x\ y$ 

theorem type_sound:
   $(c,s) \rightarrow^* (c',s') \implies \Gamma \vdash c \implies \Gamma \vdash s \implies c' \neq SKIP$ 
 $\implies \exists cs''. (c',s') \rightarrow cs''$ 
apply(induction rule:star_induct)
apply (metis progress)
by (metis styping_preservation ctyping_preservation)

end

```

## 8 Security Type Systems

### 8.1 Security Levels and Expressions

```

theory Sec_Type_Expr imports Big_Step
begin

```

```

type_synonym level = nat

```

```

class sec =
fixes sec ::  $'a \Rightarrow nat$ 

```

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! — a variable of length  $n$  has security level  $n$ :

**instantiation**  $list :: (type)sec$   
**begin**

**definition**  $sec(x :: 'a list) = length\ x$

**instance** ..

**end**

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

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

**instance** ..

**end**

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

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

**instance** ..

**end**

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

**abbreviation**  $eq\_less :: state \Rightarrow state \Rightarrow level \Rightarrow bool$

$((- = -'(< -^)) [51,51,0] 50)$  **where**  
 $s = s' (< l) == (\forall x. \text{sec } x < l \longrightarrow s \ x = s' \ x)$

**lemma** *aval\_eq\_if\_eq\_le*:

$\llbracket s_1 = s_2 (\leq l); \text{sec } a \leq l \rrbracket \Longrightarrow \text{aval } a \ s_1 = \text{aval } a \ s_2$   
**by** (*induct a*) *auto*

**lemma** *bval\_eq\_if\_eq\_le*:

$\llbracket s_1 = s_2 (\leq l); \text{sec } b \leq l \rrbracket \Longrightarrow \text{bval } b \ s_1 = \text{bval } b \ s_2$   
**by** (*induct b*) (*auto simp add: aval\_eq\_if\_eq\_le*)

**end**

## 8.2 Security Typing of Commands

**theory** *Sec\_Typing* **imports** *Sec\_Type\_Expr*  
**begin**

### 8.2.1 Syntax Directed Typing

**inductive** *sec\_type* :: *nat*  $\Rightarrow$  *com*  $\Rightarrow$  *bool* ( $(-/ \vdash -) [0,0] 50$ ) **where**

*Skip*:

$l \vdash \text{SKIP} \mid$

*Assign*:

$\llbracket \text{sec } x \geq \text{sec } a; \text{sec } x \geq l \rrbracket \Longrightarrow l \vdash x ::= a \mid$

*Seq*:

$\llbracket l \vdash c_1; l \vdash c_2 \rrbracket \Longrightarrow l \vdash c_1;;c_2 \mid$

*If*:

$\llbracket \max(\text{sec } b) \ l \vdash c_1; \max(\text{sec } b) \ l \vdash c_2 \rrbracket \Longrightarrow l \vdash \text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2 \mid$

*While*:

$\max(\text{sec } b) \ l \vdash c \Longrightarrow l \vdash \text{WHILE } b \ \text{DO } c$

**code\_pred** (*expected\_modes*:  $i \Rightarrow i \Rightarrow \text{bool}$ ) *sec\_type* .

**value**  $0 \vdash \text{IF } \text{Less } (V \ "x1") \ (V \ "x") \ \text{THEN } "x1" ::= N \ 0 \ \text{ELSE } \text{SKIP}$

**value**  $1 \vdash \text{IF } \text{Less } (V \ "x1") \ (V \ "x") \ \text{THEN } "x" ::= N \ 0 \ \text{ELSE } \text{SKIP}$

**value**  $2 \vdash \text{IF } \text{Less } (V \ "x1") \ (V \ "x") \ \text{THEN } "x1" ::= N \ 0 \ \text{ELSE } \text{SKIP}$

**inductive\_cases** [*elim!*]:

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

An important property: anti-monotonicity.

**lemma** *anti\_mono*:  $\llbracket l \vdash c; l' \leq l \rrbracket \Longrightarrow l' \vdash c$

**apply**(*induction 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:  $\llbracket (c,s) \Rightarrow t; l \vdash c \rrbracket \Longrightarrow s = t (< l)$ 
proof(induction rule: big_step_induct)
  case Skip thus ?case by simp
next
  case Assign thus ?case by auto
next
  case Seq thus ?case by auto
next
  case (IfTrue b s c1)
  hence max (sec b)  $l \vdash c1$  by auto
  hence  $l \vdash c1$  by (metis max.cobounded2 anti_mono)
  thus ?case using IfTrue.IH by metis
next
  case (IfFalse b s c2)
  hence max (sec b)  $l \vdash c2$  by auto
  hence  $l \vdash c2$  by (metis max.cobounded2 anti_mono)
  thus ?case using IfFalse.IH by metis
next
  case WhileFalse thus ?case by auto
next
  case (WhileTrue b s1 c)
  hence max (sec b)  $l \vdash c$  by auto
  hence  $l \vdash c$  by (metis max.cobounded2 anti_mono)
  thus ?case using WhileTrue by metis
qed

```

```

theorem noninterference:
   $\llbracket (c,s) \Rightarrow s'; (c,t) \Rightarrow t'; 0 \vdash c; s = t (\leq l) \rrbracket$ 
   $\Longrightarrow s' = t' (\leq l)$ 
proof(induction arbitrary: t t' rule: big_step_induct)
  case Skip thus ?case by auto
next
  case (Assign x a s)
  have [simp]:  $t' = t(x := \text{aval } a \ t)$  using Assign by auto
  have sec x  $\geq$  sec a using  $\langle 0 \vdash x ::= a \rangle$  by auto
  show ?case

```

```

proof auto
  assume  $\text{sec } x \leq l$ 
  with  $\langle \text{sec } x \geq \text{sec } a \rangle$  have  $\text{sec } a \leq l$  by arith
  thus  $\text{aval } a \text{ } s = \text{aval } a \text{ } t$ 
    by (rule aval_eq_if_eq_le[OF  $\langle s = t (\leq l) \rangle$ ])
next
  fix  $y$  assume  $y \neq x$   $\text{sec } y \leq l$ 
  thus  $s \ y = t \ y$  using  $\langle s = t (\leq l) \rangle$  by simp
qed
next
  case Seq thus ?case by blast
next
  case (IfTrue  $b \ s \ c1 \ s' \ c2$ )
  have  $\text{sec } b \vdash c1 \ \text{sec } b \vdash c2$  using  $\langle 0 \vdash \text{IF } b \ \text{THEN } c1 \ \text{ELSE } c2 \rangle$  by auto
  show ?case
  proof cases
    assume  $\text{sec } b \leq l$ 
    hence  $s = t (\leq \text{sec } b)$  using  $\langle s = t (\leq l) \rangle$  by auto
    hence  $\text{bval } b \ t$  using  $\langle \text{bval } b \ s \rangle$  by (simp add: bval_eq_if_eq_le)
    with IfTrue.IH IfTrue.prems(1,3)  $\langle \text{sec } b \vdash c1 \rangle$  anti_mono
    show ?thesis by auto
  next
    assume  $\neg \text{sec } b \leq l$ 
    have 1:  $\text{sec } b \vdash \text{IF } b \ \text{THEN } c1 \ \text{ELSE } c2$ 
      by (rule sec.type.intros)(simp_all add:  $\langle \text{sec } b \vdash c1 \rangle \langle \text{sec } b \vdash c2 \rangle$ )
    from confinement[OF  $\langle (c1, s) \Rightarrow s' \rangle \langle \text{sec } b \vdash c1 \rangle$ ]  $\langle \neg \text{sec } b \leq l \rangle$ 
    have  $s = s' (\leq l)$  by auto
    moreover
    from confinement[OF  $\langle (\text{IF } b \ \text{THEN } c1 \ \text{ELSE } c2, t) \Rightarrow t' \ 1 \rangle \langle \neg \text{sec } b \leq l \rangle$ ]
    have  $t = t' (\leq l)$  by auto
    ultimately show  $s' = t' (\leq l)$  using  $\langle s = t (\leq l) \rangle$  by auto
  qed
next
  case (IfFalse  $b \ s \ c2 \ s' \ c1$ )
  have  $\text{sec } b \vdash c1 \ \text{sec } b \vdash c2$  using  $\langle 0 \vdash \text{IF } b \ \text{THEN } c1 \ \text{ELSE } c2 \rangle$  by auto
  show ?case
  proof cases
    assume  $\text{sec } b \leq l$ 
    hence  $s = t (\leq \text{sec } b)$  using  $\langle s = t (\leq l) \rangle$  by auto
    hence  $\neg \text{bval } b \ t$  using  $\langle \neg \text{bval } b \ s \rangle$  by (simp add: bval_eq_if_eq_le)
    with IfFalse.IH IfFalse.prems(1,3)  $\langle \text{sec } b \vdash c2 \rangle$  anti_mono
    show ?thesis by auto
  next

```



```

assume  $\neg \text{sec } b \leq l$ 
have  $1: \text{sec } b \vdash \text{IF } b \text{ THEN } c1 \text{ ELSE } c2$ 
  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]  $\langle \neg \text{sec } b \leq l$ 
have  $s = s' (\leq l)$  by auto
moreover
from confinement[OF (IF b THEN c1 ELSE c2, t) ⇒ t' 1]  $\langle \neg \text{sec } b$ 
 $\leq l$ 
have  $t = t' (\leq l)$  by auto
ultimately show  $s' = t' (\leq l)$  using  $\langle s = t (\leq l) \rangle$  by auto
qed
next
case (WhileFalse b s c)
have  $\text{sec } b \vdash c$  using WhileFalse.prem(2) by auto
show ?case
proof cases
  assume  $\text{sec } b \leq l$ 
  hence  $s = t (\leq \text{sec } b)$  using  $\langle s = t (\leq l) \rangle$  by auto
  hence  $\neg \text{bval } b \text{ t}$  using  $\langle \neg \text{bval } b \text{ s} \rangle$  by(simp add: bval_eq_if_eq_le)
  with WhileFalse.prem(1,3) show ?thesis by auto
next
  assume  $\neg \text{sec } b \leq l$ 
  have  $1: \text{sec } b \vdash \text{WHILE } b \text{ DO } c$ 
    by(rule sec.type.intros)(simp_all add: (sec b ⊢ c))
  from confinement[OF (WHILE b DO c, t) ⇒ t' 1]  $\langle \neg \text{sec } b \leq l$ 
  have  $t = t' (\leq l)$  by auto
  thus  $s = t' (\leq l)$  using  $\langle s = t (\leq l) \rangle$  by auto
qed
next
case (WhileTrue b s1 c s2 s3 t1 t3)
let  $?w = \text{WHILE } b \text{ DO } c$ 
have  $\text{sec } b \vdash c$  using  $\langle 0 \vdash \text{WHILE } b \text{ DO } c \rangle$  by auto
show ?case
proof cases
  assume  $\text{sec } b \leq l$ 
  hence  $s1 = t1 (\leq \text{sec } b)$  using  $\langle s1 = t1 (\leq l) \rangle$  by auto
  hence  $\text{bval } b \text{ t1}$ 
    using  $\langle \text{bval } b \text{ s1} \rangle$  by(simp add: bval_eq_if_eq_le)
  then obtain  $t2$  where  $(c, t1) \Rightarrow t2$   $(?w, t2) \Rightarrow t3$ 
    using  $\langle (?w, t1) \Rightarrow t3 \rangle$  by auto
  from WhileTrue.IH(2)[OF (c, t1) ⇒ t2 (?w, t2) ⇒ t3] (0 ⊢ ?w)
    WhileTrue.IH(1)[OF (c, t1) ⇒ t2] anti_mono[OF (sec b ⊢ c)
     $\langle s1 = t1 (\leq l) \rangle]$ 
  show ?thesis by simp

```

```

next
  assume  $\neg \text{sec } b \leq l$ 
  have  $1: \text{sec } b \vdash ?w$  by (rule sec_type.intros)(simp_all add: (sec  $b \vdash c$ ))
  from confinement[OF big_step.WhileTrue[OF WhileTrue.hyps] 1]  $\langle \neg \text{sec } b \leq l \rangle$ 
  have  $s1 = s3 (\leq l)$  by auto
  moreover
  from confinement[OF  $\langle (\text{WHILE } b \text{ DO } c, t1) \Rightarrow t3 \rangle 1$ ]  $\langle \neg \text{sec } b \leq l \rangle$ 
  have  $t1 = t3 (\leq l)$  by auto
  ultimately show  $s3 = t3 (\leq l)$  using  $\langle s1 = t1 (\leq l) \rangle$  by auto
qed
qed

```

## 8.2.2 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*  $\Rightarrow$  *com*  $\Rightarrow$  *bool* (( $\_ / \vdash'' \_$ ) [0,0] 50) **where**

*Skip'*:

$l \vdash' \text{SKIP} \mid$

*Assign'*:

$\llbracket \text{sec } x \geq \text{sec } a; \text{sec } x \geq l \rrbracket \Longrightarrow l \vdash' x ::= a \mid$

*Seq'*:

$\llbracket l \vdash' c_1; l \vdash' c_2 \rrbracket \Longrightarrow l \vdash' c_1;;c_2 \mid$

*If'*:

$\llbracket \text{sec } b \leq l; l \vdash' c_1; l \vdash' c_2 \rrbracket \Longrightarrow l \vdash' \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 \mid$

*While'*:

$\llbracket \text{sec } b \leq l; l \vdash' c \rrbracket \Longrightarrow l \vdash' \text{WHILE } b \text{ DO } c \mid$

*anti\_mono'*:

$\llbracket l \vdash' c; l' \leq l \rrbracket \Longrightarrow l' \vdash' c$

**lemma** *sec\_type\_sec\_type'*:  $l \vdash c \Longrightarrow l \vdash' c$

**apply** (*induction rule: sec\_type.induct*)

**apply** (*metis Skip'*)

**apply** (*metis Assign'*)

**apply** (*metis Seq'*)

**apply** (*metis max.commute max.absorb\_iff2 nat.le\_linear If' anti\_mono'*)

**by** (*metis less\_or\_eq\_imp\_le max.absorb1 max.absorb2 nat.le\_linear While' anti\_mono'*)

**lemma** *sec\_type'\_sec\_type*:  $l \vdash' c \Longrightarrow l \vdash c$

```

apply(induction rule: sec_type'.induct)
apply (metis Skip)
apply (metis Assign)
apply (metis Seq)
apply (metis max.absorb2 If)
apply (metis max.absorb2 While)
by (metis anti_mono)

```

### 8.2.3 A Bottom-Up Typing System

**inductive** *sec\_type2* :: *com*  $\Rightarrow$  *level*  $\Rightarrow$  *bool* (( $\vdash$  \_ : \_) [0,0] 50) **where**

*Skip2*:

$\vdash$  *SKIP* : *l* |

*Assign2*:

$sec\ x \geq sec\ a \Longrightarrow \vdash x ::= a : sec\ x$  |

*Seq2*:

$\llbracket \vdash c_1 : l_1; \vdash c_2 : l_2 \rrbracket \Longrightarrow \vdash c_1;;c_2 : min\ l_1\ l_2$  |

*If2*:

$\llbracket sec\ b \leq min\ l_1\ l_2; \vdash c_1 : l_1; \vdash c_2 : l_2 \rrbracket$   
 $\Longrightarrow \vdash IF\ b\ THEN\ c_1\ ELSE\ c_2 : min\ l_1\ l_2$  |

*While2*:

$\llbracket sec\ b \leq l; \vdash c : l \rrbracket \Longrightarrow \vdash WHILE\ b\ DO\ c : l$

**lemma** *sec\_type2\_sec\_type'*:  $\vdash c : l \Longrightarrow l \vdash' c$

**apply**(*induction rule: sec\_type2.induct*)

**apply** (*metis Skip'*)

**apply** (*metis Assign' eq\_imp\_le*)

**apply** (*metis Seq' anti\_mono' min.cobounded1 min.cobounded2*)

**apply** (*metis If' anti\_mono' min.absorb2 min.absorb\_iff1 nat\_le\_linear*)

**by** (*metis While'*)

**lemma** *sec\_type'\_sec\_type2*:  $l \vdash' c \Longrightarrow \exists l' \geq l. \vdash c : l'$

**apply**(*induction rule: sec\_type'.induct*)

**apply** (*metis Skip2 le\_refl*)

**apply** (*metis Assign2*)

**apply** (*metis Seq2 min.boundedI*)

**apply** (*metis If2 inf\_greatest inf\_nat\_def le\_trans*)

**apply** (*metis While2 le\_trans*)

**by** (*metis le\_trans*)

**end**

### 8.3 Termination-Sensitive Systems

```
theory Sec_TypingT imports Sec_Type_Expr
begin
```

#### 8.3.1 A Syntax Directed System

```
inductive sec_type :: nat  $\Rightarrow$  com  $\Rightarrow$  bool ((-/  $\vdash$  -) [0,0] 50) where
```

```
  Skip:
```

```
    l  $\vdash$  SKIP |
```

```
  Assign:
```

```
     $\llbracket \text{sec } x \geq \text{sec } a; \text{sec } x \geq l \rrbracket \Longrightarrow l \vdash x ::= a |$ 
```

```
  Seq:
```

```
    l  $\vdash$  c1  $\Longrightarrow$  l  $\vdash$  c2  $\Longrightarrow$  l  $\vdash$  c1;;c2 |
```

```
  If:
```

```
     $\llbracket \text{max } (\text{sec } b) \text{ l } \vdash \text{c}_1; \text{max } (\text{sec } b) \text{ l } \vdash \text{c}_2 \rrbracket$   

 $\Longrightarrow l \vdash \text{IF } b \text{ THEN } \text{c}_1 \text{ ELSE } \text{c}_2 |$ 
```

```
  While:
```

```
    sec b = 0  $\Longrightarrow$  0  $\vdash$  c  $\Longrightarrow$  0  $\vdash$  WHILE b DO c
```

```
code_pred (expected_modes: i  $\Rightarrow$  i  $\Rightarrow$  bool) sec_type .
```

```
inductive_cases [elim!]:
```

```
  l  $\vdash$  x ::= a | l  $\vdash$  c1;;c2 | l  $\vdash$  IF b THEN c1 ELSE c2 | l  $\vdash$  WHILE b DO c
```

```
lemma anti_mono: l  $\vdash$  c  $\Longrightarrow$  l'  $\leq$  l  $\Longrightarrow$  l'  $\vdash$  c
```

```
apply(induction arbitrary: l' rule: sec_type.induct)
```

```
apply (metis sec_type.intros(1))
```

```
apply (metis le_trans sec_type.intros(2))
```

```
apply (metis sec_type.intros(3))
```

```
apply (metis If le_refl sup_mono sup_nat_def)
```

```
by (metis While le_0_eq)
```

```
lemma confinement: (c,s)  $\Rightarrow$  t  $\Longrightarrow$  l  $\vdash$  c  $\Longrightarrow$  s = t (< l)
```

```
proof(induction rule: big_step_induct)
```

```
  case Skip thus ?case by simp
```

```
next
```

```
  case Assign thus ?case by auto
```

```
next
```

```
  case Seq thus ?case by auto
```

```
next
```

```
  case (IfTrue b s c1)
```

```

hence  $\text{max} (\text{sec } b) l \vdash c1$  by auto
hence  $l \vdash c1$  by (metis max.cobounded2 anti_mono)
thus ?case using IfTrue.IH by metis
next
  case (IfFalse b s c2)
  hence  $\text{max} (\text{sec } b) l \vdash c2$  by auto
  hence  $l \vdash c2$  by (metis max.cobounded2 anti_mono)
  thus ?case using IfFalse.IH by metis
next
  case WhileFalse thus ?case by auto
next
  case (WhileTrue b s1 c)
  hence  $l \vdash c$  by auto
  thus ?case using WhileTrue by metis
qed

lemma termi_if_non0:  $l \vdash c \implies l \neq 0 \implies \exists t. (c,s) \Rightarrow t$ 
apply(induction arbitrary: s rule: sec_type.induct)
apply (metis big_step.Skip)
apply (metis big_step.Assign)
apply (metis big_step.Seq)
apply (metis IfFalse IfTrue le0 le_antisym max.cobounded2)
apply simp
done

theorem noninterference:  $(c,s) \Rightarrow s' \implies 0 \vdash c \implies s = t (\leq l)$ 
 $\implies \exists t'. (c,t) \Rightarrow t' \wedge s' = t' (\leq l)$ 
proof(induction arbitrary: t rule: big_step.induct)
  case Skip thus ?case by auto
next
  case (Assign x a s)
  have  $\text{sec } x \geq \text{sec } a$  using  $\langle 0 \vdash x ::= a \rangle$  by auto
  have  $(x ::= a, t) \Rightarrow t(x := \text{aval } a t)$  by auto
  moreover
  have  $s(x := \text{aval } a s) = t(x := \text{aval } a t) (\leq l)$ 
  proof auto
    assume  $\text{sec } x \leq l$ 
    with  $\langle \text{sec } x \geq \text{sec } a \rangle$  have  $\text{sec } a \leq l$  by arith
    thus  $\text{aval } a s = \text{aval } a t$ 
    by (rule aval_eq_if_eq_le[OF \langle s = t (\leq l) \rangle])
  next
  fix  $y$  assume  $y \neq x$   $\text{sec } y \leq l$ 
  thus  $s y = t y$  using  $\langle s = t (\leq l) \rangle$  by simp
qed

```

**ultimately show** *?case* **by** *blast*  
**next**  
**case** *Seq* **thus** *?case* **by** *blast*  
**next**  
**case** (*IfTrue* *b s c1 s' c2*)  
**have** *sec b*  $\vdash$  *c1* *sec b*  $\vdash$  *c2* **using**  $\langle 0 \vdash \text{IF } b \text{ THEN } c1 \text{ ELSE } c2 \rangle$  **by** *auto*  
**obtain** *t'* **where** *t'*:  $(c1, t) \Rightarrow t' s' = t' (\leq l)$   
**using** *IfTrue.IH*[*OF anti\_mono*[*OF*  $\langle \text{sec } b \vdash c1 \rangle$ ]  $\langle s = t (\leq l) \rangle$ ] **by** *blast*  
**show** *?case*  
**proof cases**  
**assume** *sec b*  $\leq l$   
**hence**  $s = t (\leq \text{sec } b)$  **using**  $\langle s = t (\leq l) \rangle$  **by** *auto*  
**hence** *bval b t* **using**  $\langle \text{bval } b s \rangle$  **by**(*simp add: bval\_eq\_if\_eq\_le*)  
**thus** *?thesis* **by** (*metis t' big\_step.IfTrue*)  
**next**  
**assume**  $\neg \text{sec } b \leq l$   
**hence** *0: sec b*  $\neq 0$  **by** *arith*  
**have** *1: sec b*  $\vdash$  *IF b THEN c1 ELSE c2*  
**by**(*rule sec.type.intros*)(*simp\_all add:*  $\langle \text{sec } b \vdash c1 \rangle \langle \text{sec } b \vdash c2 \rangle$ )  
**from** *confinement*[*OF big\_step.IfTrue*[*OF IfTrue*(*1,2*)] *1*]  $\langle \neg \text{sec } b \leq l \rangle$   
**have**  $s = s' (\leq l)$  **by** *auto*  
**moreover**  
**from** *termi\_if\_non0*[*OF 1 0, of t*] **obtain** *t'* **where**  
*t'*:  $(\text{IF } b \text{ THEN } c1 \text{ ELSE } c2, t) \Rightarrow t' ..$   
**moreover**  
**from** *confinement*[*OF t' 1*]  $\langle \neg \text{sec } b \leq l \rangle$   
**have**  $t = t' (\leq l)$  **by** *auto*  
**ultimately**  
**show** *?case* **using**  $\langle s = t (\leq l) \rangle$  **by** *auto*  
**qed**  
**next**  
**case** (*IfFalse* *b s c2 s' c1*)  
**have** *sec b*  $\vdash$  *c1* *sec b*  $\vdash$  *c2* **using**  $\langle 0 \vdash \text{IF } b \text{ THEN } c1 \text{ ELSE } c2 \rangle$  **by** *auto*  
**obtain** *t'* **where** *t'*:  $(c2, t) \Rightarrow t' s' = t' (\leq l)$   
**using** *IfFalse.IH*[*OF anti\_mono*[*OF*  $\langle \text{sec } b \vdash c2 \rangle$ ]  $\langle s = t (\leq l) \rangle$ ] **by** *blast*  
**show** *?case*  
**proof cases**  
**assume** *sec b*  $\leq l$   
**hence**  $s = t (\leq \text{sec } b)$  **using**  $\langle s = t (\leq l) \rangle$  **by** *auto*  
**hence**  $\neg \text{bval } b t$  **using**  $\langle \neg \text{bval } b s \rangle$  **by**(*simp add: bval\_eq\_if\_eq\_le*)  
**thus** *?thesis* **by** (*metis t' big\_step.IfFalse*)  
**next**  
**assume**  $\neg \text{sec } b \leq l$   
**hence** *0: sec b*  $\neq 0$  **by** *arith*

```

have 1:  $sec\ b \vdash IF\ b\ THEN\ c1\ ELSE\ c2$ 
  by (rule sec_type.intros)(simp_all add:  $\langle sec\ b \vdash c1 \rangle \langle sec\ b \vdash c2 \rangle$ )
from confinement[OF big_step.IfFalse[OF IfFalse(1,2)] 1]  $\langle \neg\ sec\ b \leq l \rangle$ 
have  $s = s' (\leq l)$  by auto
moreover
from termi.if_non0[OF 1 0, of t] obtain  $t'$  where
   $t': (IF\ b\ THEN\ c1\ ELSE\ c2, t) \Rightarrow t' ..$ 
moreover
from confinement[OF  $t' 1$ ]  $\langle \neg\ sec\ b \leq l \rangle$ 
have  $t = t' (\leq l)$  by auto
ultimately
show ?case using  $\langle s = t (\leq l) \rangle$  by auto
qed
next
case (WhileFalse  $b\ s\ c$ )
  hence [simp]:  $sec\ b = 0$  by auto
  have  $s = t (\leq sec\ b)$  using  $\langle s = t (\leq l) \rangle$  by auto
  hence  $\neg\ bval\ b\ t$  using  $\langle \neg\ bval\ b\ s \rangle$  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  $\langle 0 \vdash ?w \rangle$  have [simp]:  $sec\ b = 0$  by auto
  have  $0 \vdash c$  using  $\langle 0 \vdash WHILE\ b\ DO\ c \rangle$  by auto
  from WhileTrue.IH(1)[OF this  $\langle s = t (\leq l) \rangle$ ]
  obtain  $t''$  where  $(c, t) \Rightarrow t''$  and  $s'' = t'' (\leq l)$  by blast
  from WhileTrue.IH(2)[OF  $\langle 0 \vdash ?w \rangle$  this(2)]
  obtain  $t'$  where  $(?w, t'') \Rightarrow t'$  and  $s' = t' (\leq l)$  by blast
  from  $\langle bval\ b\ s \rangle$  have  $bval\ b\ t$ 
  using bval_eq_if_eq_le[OF  $\langle s = t (\leq l) \rangle$ ] by auto
  show ?case
  using big_step.WhileTrue[OF  $\langle bval\ b\ t \rangle \langle (c, t) \Rightarrow t'' \rangle \langle (?w, t'') \Rightarrow t' \rangle$ ]
  by (metis  $\langle s' = t' (\leq l) \rangle$ )
qed

```

### 8.3.2 The Standard 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'* ::  $nat \Rightarrow com \Rightarrow bool$  ( $(\_ / \vdash'' \_)$  [0,0] 50) **where**  
*Skip'*:  
 $l \vdash' SKIP \mid$

*Assign'*:  
 $\llbracket \text{sec } x \geq \text{sec } a; \text{sec } x \geq l \rrbracket \Longrightarrow l \vdash' x ::= a \mid$   
*Seq'*:  
 $l \vdash' c_1 \Longrightarrow l \vdash' c_2 \Longrightarrow l \vdash' c_1;;c_2 \mid$   
*If'*:  
 $\llbracket \text{sec } b \leq l; l \vdash' c_1; l \vdash' c_2 \rrbracket \Longrightarrow l \vdash' \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 \mid$   
*While'*:  
 $\llbracket \text{sec } b = 0; 0 \vdash' c \rrbracket \Longrightarrow 0 \vdash' \text{WHILE } b \text{ DO } c \mid$   
*anti\_mono'*:  
 $\llbracket l \vdash' c; l' \leq l \rrbracket \Longrightarrow l' \vdash' c$

**lemma** *sec\_type\_sec\_type'*:  
 $l \vdash c \Longrightarrow l \vdash' c$   
**apply**(*induction rule: sec\_type.induct*)  
**apply** (*metis Skip'*)  
**apply** (*metis Assign'*)  
**apply** (*metis Seq'*)  
**apply** (*metis max.commute max.absorb\_iff2 nat.le\_linear If' anti\_mono'*)  
**by** (*metis While'*)

**lemma** *sec\_type'\_sec\_type*:  
 $l \vdash' c \Longrightarrow l \vdash c$   
**apply**(*induction rule: sec\_type'.induct*)  
**apply** (*metis Skip*)  
**apply** (*metis Assign*)  
**apply** (*metis Seq*)  
**apply** (*metis max.absorb2 If*)  
**apply** (*metis While*)  
**by** (*metis anti\_mono*)

**corollary** *sec\_type\_eq*:  $l \vdash c \longleftrightarrow l \vdash' c$   
**by** (*metis sec\_type'\_sec\_type sec\_type\_sec\_type'*)

**end**

## 9 Definite Initialization Analysis

**theory** *Vars* **imports** *Com*  
**begin**



## 9.1 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 ⇒ vname 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_aexp :: aexp ⇒ vname set where  
vars (N n) = {} |  
vars (V x) = {x} |  
vars (Plus a1 a2) = vars a1 ∪ vars a2
```

```
instance ..
```

```
end
```

```
value vars (Plus (V "x") (V "y"))
```

```
instantiation bexp :: vars  
begin
```

```
fun vars_bexp :: bexp ⇒ vname set where  
vars (Bc v) = {} |  
vars (Not b) = vars b |  
vars (And b1 b2) = vars b1 ∪ vars b2 |  
vars (Less a1 a2) = vars a1 ∪ vars a2
```

```
instance ..
```

```
end
```

```
value vars (Less (Plus (V "z") (V "y")) (V "x"))
```

```
abbreviation
```

```
eq_on :: ('a ⇒ 'b) ⇒ ('a ⇒ 'b) ⇒ 'a set ⇒ bool  
((- =/ -/ on -) [50,0,50] 50) where
```

$f = g \text{ on } X \iff \forall x \in X. f x = g x$

**lemma** *aval\_eq\_if\_eq\_on\_vars*[simp]:

$s_1 = s_2 \text{ on vars } a \implies \text{aval } a \ s_1 = \text{aval } a \ s_2$

**apply** (*induction a*)

**apply** *simp\_all*

**done**

**lemma** *bval\_eq\_if\_eq\_on\_vars*:

$s_1 = s_2 \text{ on vars } b \implies \text{bval } b \ s_1 = \text{bval } b \ s_2$

**proof** (*induction b*)

**case** (*Less a1 a2*)

**hence**  $\text{aval } a1 \ s_1 = \text{aval } a1 \ s_2$  **and**  $\text{aval } a2 \ s_1 = \text{aval } a2 \ s_2$  **by** *simp\_all*

**thus** *?case* **by** *simp*

**qed** *simp\_all*

**fun** *lvars* :: *com*  $\Rightarrow$  *vname set* **where**

*lvars* *SKIP* = {} |

*lvars* ( $x ::= e$ ) = {*x*} |

*lvars* ( $c1 ;; c2$ ) = *lvars* *c1*  $\cup$  *lvars* *c2* |

*lvars* (*IF* *b* *THEN* *c1* *ELSE* *c2*) = *lvars* *c1*  $\cup$  *lvars* *c2* |

*lvars* (*WHILE* *b* *DO* *c*) = *lvars* *c*

**fun** *rvars* :: *com*  $\Rightarrow$  *vname set* **where**

*rvars* *SKIP* = {} |

*rvars* ( $x ::= e$ ) = *vars* *e* |

*rvars* ( $c1 ;; c2$ ) = *rvars* *c1*  $\cup$  *rvars* *c2* |

*rvars* (*IF* *b* *THEN* *c1* *ELSE* *c2*) = *vars* *b*  $\cup$  *rvars* *c1*  $\cup$  *rvars* *c2* |

*rvars* (*WHILE* *b* *DO* *c*) = *vars* *b*  $\cup$  *rvars* *c*

**instantiation** *com* :: *vars*

**begin**

**definition** *vars\_com* *c* = *lvars* *c*  $\cup$  *rvars* *c*

**instance** ..

**end**

**lemma** *vars\_com\_simps*[simp]:

*vars* *SKIP* = {}

*vars* ( $x ::= e$ ) = {*x*}  $\cup$  *vars* *e*

*vars* ( $c1 ;; c2$ ) = *vars* *c1*  $\cup$  *vars* *c2*

*vars* (*IF* *b* *THEN* *c1* *ELSE* *c2*) = *vars* *b*  $\cup$  *vars* *c1*  $\cup$  *vars* *c2*

$vars (WHILE\ b\ DO\ c) = vars\ b \cup vars\ c$   
**by**(*auto simp: vars\_com\_def*)

**lemma** *finite\_aval[simp]: finite(vars(a::aexp))*  
**by**(*induction a simp\_all*)

**lemma** *finite\_bvars[simp]: finite(vars(b::bexp))*  
**by**(*induction b simp\_all*)

**lemma** *finite\_lvars[simp]: finite(lvars(c))*  
**by**(*induction c simp\_all*)

**lemma** *finite\_rvars[simp]: finite(rvars(c))*  
**by**(*induction c simp\_all*)

**lemma** *finite\_cvars[simp]: finite(vars(c::com))*  
**by**(*simp add: vars\_com\_def*)

**end**

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

## 9.2 Initialization-Sensitive Expressions Evaluation

**type\_synonym** *state = vname  $\Rightarrow$  val option*

**fun** *aval :: aexp  $\Rightarrow$  state  $\Rightarrow$  val option where*  
*aval (N i) s = Some i |*  
*aval (V x) s = s x |*  
*aval (Plus a<sub>1</sub> a<sub>2</sub>) s =*  
*(case (aval a<sub>1</sub> s, aval a<sub>2</sub> s) of*  
*(Some i<sub>1</sub>, Some i<sub>2</sub>)  $\Rightarrow$  Some(i<sub>1</sub>+i<sub>2</sub>) | \_  $\Rightarrow$  None)*

**fun** *bval :: bexp  $\Rightarrow$  state  $\Rightarrow$  bool option where*  
*bval (Bc v) s = Some v |*  
*bval (Not b) s = (case bval b s of None  $\Rightarrow$  None | Some bv  $\Rightarrow$  Some( $\neg$  bv))*  
*|*  
*bval (And b<sub>1</sub> b<sub>2</sub>) s = (case (bval b<sub>1</sub> s, bval b<sub>2</sub> s) of*  
*(Some bv<sub>1</sub>, Some bv<sub>2</sub>)  $\Rightarrow$  Some(bv<sub>1</sub> & bv<sub>2</sub>) | \_  $\Rightarrow$  None) |*

*bval (Less a<sub>1</sub> a<sub>2</sub>) s = (case (aval a<sub>1</sub> s, aval a<sub>2</sub> s) of  
 (Some i<sub>1</sub>, Some i<sub>2</sub>) ⇒ Some(i<sub>1</sub> < i<sub>2</sub>) | \_ ⇒ 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\_Init*

**imports** *Vars Com*

**begin**

### 9.3 Definite Initialization Analysis

**inductive** *D :: vname set ⇒ com ⇒ vname set ⇒ bool* **where**

*Skip: D A SKIP A |*

*Assign: vars a ⊆ A ⇒ D A (x ::= a) (insert x A) |*

*Seq: [ D A<sub>1</sub> c<sub>1</sub> A<sub>2</sub>; D A<sub>2</sub> c<sub>2</sub> A<sub>3</sub> ] ⇒ D A<sub>1</sub> (c<sub>1</sub>;; c<sub>2</sub>) A<sub>3</sub> |*

*If: [ vars b ⊆ A; D A c<sub>1</sub> A<sub>1</sub>; D A c<sub>2</sub> A<sub>2</sub> ] ⇒*

*D A (IF b THEN c<sub>1</sub> ELSE c<sub>2</sub>) (A<sub>1</sub> Int A<sub>2</sub>) |*

*While: [ vars b ⊆ A; D A c A' ] ⇒ D A (WHILE b DO c) A*

**inductive\_cases** [*elim!*]:

*D A SKIP A'*

*D A (x ::= a) A'*

*D A (c<sub>1</sub>;;c<sub>2</sub>) A'*

*D A (IF b THEN c<sub>1</sub> ELSE c<sub>2</sub>) A'*

*D A (WHILE b DO c) A'*

**lemma** *D.incr:*

*D A c A' ⇒ A ⊆ A'*

**by** (*induct rule: D.induct*) *auto*

**end**

**theory** *Def\_Init\_Big*

**imports** *Def\_Init\_Exp Def\_Init*

**begin**

## 9.4 Initialization-Sensitive Big Step Semantics

**inductive**

$big\_step :: (com \times state\ option) \Rightarrow state\ option \Rightarrow bool$  (**infix**  $\Rightarrow$  55)

**where**

$None: (c, None) \Rightarrow None$  |

$Skip: (SKIP, s) \Rightarrow s$  |

$AssignNone: aval\ a\ s = None \Longrightarrow (x ::= a, Some\ s) \Rightarrow None$  |

$Assign: aval\ a\ s = Some\ i \Longrightarrow (x ::= a, Some\ s) \Rightarrow Some(s(x := Some\ i))$

|

$Seq: (c_1, s_1) \Rightarrow s_2 \Longrightarrow (c_2, s_2) \Rightarrow s_3 \Longrightarrow (c_1;;c_2, s_1) \Rightarrow s_3$  |

$IfNone: bval\ b\ s = None \Longrightarrow (IF\ b\ THEN\ c_1\ ELSE\ c_2, Some\ s) \Rightarrow None$  |

$IfTrue: \llbracket bval\ b\ s = Some\ True; (c_1, Some\ s) \Rightarrow s' \rrbracket \Longrightarrow$

$(IF\ b\ THEN\ c_1\ ELSE\ c_2, Some\ s) \Rightarrow s'$  |

$IfFalse: \llbracket bval\ b\ s = Some\ False; (c_2, Some\ s) \Rightarrow s' \rrbracket \Longrightarrow$

$(IF\ b\ THEN\ c_1\ ELSE\ c_2, Some\ s) \Rightarrow s'$  |

$WhileNone: bval\ b\ s = None \Longrightarrow (WHILE\ b\ DO\ c, Some\ s) \Rightarrow None$  |

$WhileFalse: bval\ b\ s = Some\ False \Longrightarrow (WHILE\ b\ DO\ c, Some\ s) \Rightarrow Some\ s$  |

$WhileTrue:$

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

$\Longrightarrow$

$(WHILE\ b\ DO\ c, Some\ s) \Rightarrow s''$

**lemmas**  $big\_step\_induct = big\_step.induct[split\_format(complete)]$

## 9.5 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** (*induction*  $c\ Some\ s\ s'$  *arbitrary*:  $s\ A\ A'$  *rule*:  $big\_step\_induct$ )

**case** *AssignNone* **thus** *?case*

**by** *auto* (*metis*  $aval\_Some\ option.simps(3)$  *subset\\_trans*)

**next**

**case** *Seq* **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  $\langle D A (WHILE\ b\ DO\ c) A' \rangle$  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

```

```

theory Def_Init_Small
imports Star Def_Init_Exp Def_Init
begin

```

## 9.6 Initialization-Sensitive Small Step Semantics

```

inductive
  small_step :: (com  $\times$  state)  $\Rightarrow$  (com  $\times$  state)  $\Rightarrow$  bool (infix  $\rightarrow$  55)
where
  Assign:  $aval\ a\ s = Some\ i \Longrightarrow (x ::= a, s) \rightarrow (SKIP, s(x := Some\ i))$  |
  Seq1:  $(SKIP;;c,s) \rightarrow (c,s)$  |
  Seq2:  $(c_1,s) \rightarrow (c_1',s') \Longrightarrow (c_1;;c_2,s) \rightarrow (c_1';;c_2,s')$  |
  IfTrue:  $bval\ b\ s = Some\ True \Longrightarrow (IF\ b\ THEN\ c_1\ ELSE\ c_2,s) \rightarrow (c_1,s)$  |
  IfFalse:  $bval\ b\ s = Some\ False \Longrightarrow (IF\ b\ THEN\ c_1\ ELSE\ c_2,s) \rightarrow (c_2,s)$  |
  While:  $(WHILE\ b\ DO\ c,s) \rightarrow (IF\ b\ THEN\ c;;\ WHILE\ b\ DO\ c\ ELSE\ SKIP,s)$ 

lemmas small_step_induct = small_step.induct[split_format(complete)]

abbreviation small_steps :: com * state  $\Rightarrow$  com * state  $\Rightarrow$  bool (infix  $\rightarrow$ *
55)

```

where  $x \rightarrow^* y == \text{star\_small\_step } x \ y$

## 9.7 Soundness wrt Small Steps

**theorem** *progress*:

$D (\text{dom } s) \ c \ A' \Longrightarrow c \neq \text{SKIP} \Longrightarrow \exists cs'. (c,s) \rightarrow cs'$

**proof** (*induction c arbitrary: s A'*)

**case** *Assign* **thus** *?case by auto (metis aval\_Some small\_step.Assign)*

**next**

**case** (*If b c1 c2*)

**then obtain** *bv* **where**  $\text{bval } b \ s = \text{Some } bv$  **by** (*auto dest!:bval\_Some*)

**then show** *?case*

**by**(*cases bv*)(*auto intro: small\_step.IfTrue small\_step.IfFalse*)

**qed** (*fastforce intro: small\_step.intros*)+

**lemma** *D\_mono*:  $D \ A \ c \ M \Longrightarrow A \subseteq A' \Longrightarrow \exists M'. D \ A' \ c \ M' \ \& \ M \leq M'$

**proof** (*induction c arbitrary: A A' M*)

**case** *Seq* **thus** *?case by auto (metis D.intros(3))*

**next**

**case** (*If b c1 c2*)

**then obtain** *M1 M2* **where**  $\text{vars } b \subseteq A \ D \ A \ c1 \ M1 \ D \ A \ c2 \ M2 \ M = M1 \cap M2$

**by** *auto*

**with** *If.IH*  $\langle A \subseteq A' \rangle$  **obtain** *M1' M2'*

**where**  $D \ A' \ c1 \ M1' \ D \ A' \ c2 \ M2'$  **and**  $M1 \subseteq M1' \ M2 \subseteq M2'$  **by** *metis*  
**hence**  $D \ A' \ (\text{IF } b \ \text{THEN } c1 \ \text{ELSE } c2) \ (M1' \cap M2')$  **and**  $M \subseteq M1' \cap M2'$

**using**  $\langle \text{vars } b \subseteq A \rangle \langle A \subseteq A' \rangle \langle M = M1 \cap M2 \rangle$  **by**(*fastforce intro: D.intros*)+

**thus** *?case by metis*

**next**

**case** *While* **thus** *?case by auto (metis D.intros(5) subset\_trans)*

**qed** (*auto intro: D.intros*)

**theorem** *D\_preservation*:

$(c,s) \rightarrow (c',s') \Longrightarrow D (\text{dom } s) \ c \ A \Longrightarrow \exists A'. D (\text{dom } s') \ c' \ A' \ \& \ A \leq A'$

**proof** (*induction arbitrary: A rule: small\_step\_induct*)

**case** (*While b c s*)

**then obtain** *A'* **where**  $A': \text{vars } b \subseteq \text{dom } s \ A = \text{dom } s \ D (\text{dom } s) \ c \ A'$   
**by** *blast*

**then obtain** *A''* **where**  $D \ A' \ c \ A''$  **by** (*metis D\_incr D\_mono*)

**with** *A'* **have**  $D (\text{dom } s) \ (\text{IF } b \ \text{THEN } c;; \ \text{WHILE } b \ \text{DO } c \ \text{ELSE } \text{SKIP}) \ (\text{dom } s)$

**by** (*metis D.If[OF*  $\langle \text{vars } b \subseteq \text{dom } s \rangle$  *D.Seq[OF*  $\langle D (\text{dom } s) \ c \ A' \rangle$

```

D.While[OF _ <D A' c A'>] D.Skip] D.incr Int.absorb1 subset_trans)
  thus ?case by (metis D.incr <A = dom s>)
next
  case Seq2 thus ?case by auto (metis D_mono D.intros(3))
qed (auto intro: D.intros)

theorem D_sound:
  (c,s) →* (c',s') ⇒ D (dom s) c A'
  ⇒ (∃ cs''. (c',s') → cs'') ∨ c' = SKIP
apply(induction arbitrary: A' rule:star_induct)
apply (metis progress)
by (metis D_preservation)

end

```

## 10 Constant Folding

```

theory Sem_Equiv
imports Big_Step
begin

```

### 10.1 Semantic Equivalence up to a Condition

```

type_synonym assn = state ⇒ bool

```

#### definition

```

equiv_up_to :: assn ⇒ com ⇒ com ⇒ bool (- ⊨ - ~ - [50,0,10] 50)

```

#### where

```

(P ⊨ c ~ c') = (∀ s s'. P s ⟶ (c,s) ⇒ s' ⟷ (c',s) ⇒ s')

```

#### definition

```

bequiv_up_to :: assn ⇒ bexp ⇒ bexp ⇒ bool (- ⊨ - <~> - [50,0,10] 50)

```

#### where

```

(P ⊨ b <~> b') = (∀ s. P s ⟶ bval b s = bval b' s)

```

#### lemma equiv\_up\_to\_True:

```

((λ_. True) ⊨ c ~ c') = (c ~ c')

```

```

by (simp add: equiv_def equiv_up_to_def)

```

#### lemma equiv\_up\_to\_weaken:

```

P ⊨ c ~ c' ⟹ (∧ s. P' s ⟹ P s) ⟹ P' ⊨ c ~ c'

```

```

by (simp add: equiv_up_to_def)

```

#### lemma equiv\_up\_toI:



$(\bigwedge s s'. P s \implies (c, s) \Rightarrow s' = (c', s) \Rightarrow s') \implies P \models c \sim c'$   
**by** (*unfold equiv\_up\_to\_def*) *blast*

**lemma** *equiv\_up\_toD1*:

$P \models c \sim c' \implies (c, s) \Rightarrow s' \implies P s \implies (c', s) \Rightarrow s'$   
**by** (*unfold equiv\_up\_to\_def*) *blast*

**lemma** *equiv\_up\_toD2*:

$P \models c \sim c' \implies (c', s) \Rightarrow s' \implies P s \implies (c, s) \Rightarrow s'$   
**by** (*unfold equiv\_up\_to\_def*) *blast*

**lemma** *equiv\_up\_to\_refl* [*simp, intro!*]:

$P \models c \sim c$   
**by** (*auto simp: equiv\_up\_to\_def*)

**lemma** *equiv\_up\_to\_sym*:

$(P \models c \sim c') = (P \models c' \sim c)$   
**by** (*auto simp: equiv\_up\_to\_def*)

**lemma** *equiv\_up\_to\_trans*:

$P \models c \sim c' \implies P \models c' \sim c'' \implies P \models c \sim c''$   
**by** (*auto simp: equiv\_up\_to\_def*)

**lemma** *bequiv\_up\_to\_refl* [*simp, intro!*]:

$P \models b <\sim> b$   
**by** (*auto simp: bequiv\_up\_to\_def*)

**lemma** *bequiv\_up\_to\_sym*:

$(P \models b <\sim> b') = (P \models b' <\sim> b)$   
**by** (*auto simp: bequiv\_up\_to\_def*)

**lemma** *bequiv\_up\_to\_trans*:

$P \models b <\sim> b' \implies P \models b' <\sim> b'' \implies P \models b <\sim> b''$   
**by** (*auto simp: bequiv\_up\_to\_def*)

**lemma** *bequiv\_up\_to\_subst*:

$P \models b <\sim> b' \implies P s \implies \text{bval } b s = \text{bval } b' s$   
**by** (*simp add: bequiv\_up\_to\_def*)

**lemma** *equiv\_up\_to\_seq*:

$P \models c \sim c' \implies Q \models d \sim d' \implies$

$(\bigwedge s s'. (c, s) \Rightarrow s' \Longrightarrow P s \Longrightarrow Q s') \Longrightarrow$   
 $P \models (c;; d) \sim (c';; d')$   
**by** (*clarsimp simp: equiv\_up\_to\_def*) *blast*

**lemma** *equiv\_up\_to\_while\_lemma\_weak*:

**shows**  $(d, s) \Rightarrow s' \Longrightarrow$   
 $P \models b <\sim> b' \Longrightarrow$   
 $P \models c \sim c' \Longrightarrow$   
 $(\bigwedge s s'. (c, s) \Rightarrow s' \Longrightarrow P s \Longrightarrow \text{bval } b s \Longrightarrow P s') \Longrightarrow$   
 $P s \Longrightarrow$   
 $d = \text{WHILE } b \text{ DO } c \Longrightarrow$   
 $(\text{WHILE } b' \text{ DO } c', s) \Rightarrow s'$

**proof** (*induction rule: big\_step\_induct*)

**case** (*WhileTrue*  $b s1 c s2 s3$ )

**hence** *IH*:  $P s2 \Longrightarrow (\text{WHILE } b' \text{ DO } c', s2) \Rightarrow s3$  **by** *auto*

**from** *WhileTrue.prem*s

**have**  $P \models b <\sim> b'$  **by** *simp*

**with**  $\langle \text{bval } b s1 \rangle \langle P s1 \rangle$

**have**  $\text{bval } b' s1$  **by** (*simp add: bequiv\_up\_to\_def*)

**moreover**

**from** *WhileTrue.prem*s

**have**  $P \models c \sim c'$  **by** *simp*

**with**  $\langle \text{bval } b s1 \rangle \langle P s1 \rangle \langle (c, s1) \Rightarrow s2 \rangle$

**have**  $(c', s1) \Rightarrow s2$  **by** (*simp add: equiv\_up\_to\_def*)

**moreover**

**from** *WhileTrue.prem*s

**have**  $\bigwedge s s'. (c, s) \Rightarrow s' \Longrightarrow P s \Longrightarrow \text{bval } b s \Longrightarrow P s'$  **by** *simp*

**with**  $\langle P s1 \rangle \langle \text{bval } b s1 \rangle \langle (c, s1) \Rightarrow s2 \rangle$

**have**  $P s2$  **by** *simp*

**hence**  $(\text{WHILE } b' \text{ DO } c', s2) \Rightarrow s3$  **by** (*rule IH*)

**ultimately**

**show** *?case* **by** *blast*

**next**

**case** *WhileFalse*

**thus** *?case* **by** (*auto simp: bequiv\_up\_to\_def*)

**qed** (*fastforce simp: equiv\_up\_to\_def bequiv\_up\_to\_def*)**+**

**lemma** *equiv\_up\_to\_while\_weak*:

**assumes**  $b: P \models b <\sim> b'$

**assumes**  $c: P \models c \sim c'$

**assumes**  $I: \bigwedge s s'. (c, s) \Rightarrow s' \Longrightarrow P s \Longrightarrow \text{bval } b s \Longrightarrow P s'$

**shows**  $P \models \text{WHILE } b \text{ DO } c \sim \text{WHILE } b' \text{ DO } c'$

**proof** –

**from**  $b$  **have**  $b': P \models b' <\sim> b$  **by** (*simp add: bequiv\_up\_to\_sym*)

**from**  $c\ b$  **have**  $c'$ :  $P \models c' \sim c$  **by** (*simp add: equiv\_up\_to\_sym*)  
**from**  $I$   
**have**  $I'$ :  $\bigwedge s\ s'. (c', s) \Rightarrow s' \Longrightarrow P\ s \Longrightarrow \text{bval } b'\ s \Longrightarrow P\ s'$   
**by** (*auto dest!: equiv\_up\_toD1 [OF c'] simp: bequiv\_up\_to\_subst [OF b']*)  
**note** *equiv\_up\_to\_while\_lemma\_weak [OF - b c]*  
*equiv\_up\_to\_while\_lemma\_weak [OF - b' c']*  
**thus** *?thesis using I I'* **by** (*auto intro!: equiv\_up\_toI*)  
**qed**

**lemma** *equiv\_up\_to\_if\_weak*:  
 $P \models b <\sim> b' \Longrightarrow P \models c \sim c' \Longrightarrow P \models d \sim d' \Longrightarrow$   
 $P \models \text{IF } b\ \text{THEN } c\ \text{ELSE } d \sim \text{IF } b'\ \text{THEN } c'\ \text{ELSE } d'$   
**by** (*auto simp: bequiv\_up\_to\_def equiv\_up\_to\_def*)

**lemma** *equiv\_up\_to\_if\_True [intro!]*:  
 $(\bigwedge s. P\ s \Longrightarrow \text{bval } b\ s) \Longrightarrow P \models \text{IF } b\ \text{THEN } c1\ \text{ELSE } c2 \sim c1$   
**by** (*auto simp: equiv\_up\_to\_def*)

**lemma** *equiv\_up\_to\_if\_False [intro!]*:  
 $(\bigwedge s. P\ s \Longrightarrow \neg \text{bval } b\ s) \Longrightarrow P \models \text{IF } b\ \text{THEN } c1\ \text{ELSE } c2 \sim c2$   
**by** (*auto simp: equiv\_up\_to\_def*)

**lemma** *equiv\_up\_to\_while\_False [intro!]*:  
 $(\bigwedge s. P\ s \Longrightarrow \neg \text{bval } b\ s) \Longrightarrow P \models \text{WHILE } b\ \text{DO } c \sim \text{SKIP}$   
**by** (*auto simp: equiv\_up\_to\_def*)

**lemma** *while\_never*:  $(c, s) \Rightarrow u \Longrightarrow c \neq \text{WHILE } (Bc\ \text{True})\ \text{DO } c'$   
**by** (*induct rule: big\_step\_induct*) *auto*

**lemma** *equiv\_up\_to\_while\_True [intro!,simp]*:  
 $P \models \text{WHILE } Bc\ \text{True}\ \text{DO } c \sim \text{WHILE } Bc\ \text{True}\ \text{DO } \text{SKIP}$   
**unfolding** *equiv\_up\_to\_def*  
**by** (*blast dest: while\_never*)

**end**  
**theory** *Fold imports Sem\_Equiv Vars begin*

## 10.2 Simple folding of arithmetic expressions

**type\_synonym**

$tab = vname \Rightarrow val\ option$

**fun** *afold* ::  $aexp \Rightarrow tab \Rightarrow aexp$  **where**  
*afold* ( $N\ n$ ) \_ =  $N\ n$  |  
*afold* ( $V\ x$ )  $t$  = (*case*  $t\ x$  of  $None \Rightarrow V\ x$  |  $Some\ k \Rightarrow N\ k$ ) |  
*afold* (*Plus*  $e1\ e2$ )  $t$  = (*case* (*afold*  $e1\ t$ , *afold*  $e2\ t$ ) of  
( $N\ n1$ ,  $N\ n2$ )  $\Rightarrow N(n1+n2)$  | ( $e1',e2'$ )  $\Rightarrow Plus\ e1'\ e2'$ )

**definition** *approx*  $t\ s \longleftrightarrow (\forall x\ k. t\ x = Some\ k \longrightarrow s\ x = k)$

**theorem** *aval\_afold[simp]*:

**assumes** *approx*  $t\ s$

**shows** *aval* (*afold*  $a\ t$ )  $s = aval\ a\ s$

**using** *assms*

**by** (*induct*  $a$ ) (*auto simp: approx\_def split: aexp.split option.split*)

**theorem** *aval\_afold\_N*:

**assumes** *approx*  $t\ s$

**shows** *afold*  $a\ t = N\ n \Longrightarrow aval\ a\ s = n$

**by** (*metis assms aval.simps(1) aval\_afold*)

**definition**

*merge*  $t1\ t2 = (\lambda m. \text{if } t1\ m = t2\ m \text{ then } t1\ m \text{ else } None)$

**primrec** *defs* ::  $com \Rightarrow tab \Rightarrow tab$  **where**

*defs* *SKIP*  $t = t$  |

*defs* ( $x ::= a$ )  $t =$

(*case* *afold*  $a\ t$  of  $N\ k \Rightarrow t(x \mapsto k)$  | \_  $\Rightarrow t(x := None)$ ) |

*defs* ( $c1;;c2$ )  $t = (defs\ c2\ o\ defs\ c1)\ t$  |

*defs* (*IF*  $b$  *THEN*  $c1$  *ELSE*  $c2$ )  $t = merge\ (defs\ c1\ t)\ (defs\ c2\ t)$  |

*defs* (*WHILE*  $b$  *DO*  $c$ )  $t = t$  |' (*-lvars*  $c$ )

**primrec** *fold* **where**

*fold* *SKIP* \_ = *SKIP* |

*fold* ( $x ::= a$ )  $t = (x ::= (afold\ a\ t))$  |

*fold* ( $c1;;c2$ )  $t = (fold\ c1\ t;; fold\ c2\ (defs\ c1\ t))$  |

*fold* (*IF*  $b$  *THEN*  $c1$  *ELSE*  $c2$ )  $t = IF\ b\ THEN\ fold\ c1\ t\ ELSE\ fold\ c2\ t$  |

*fold* (*WHILE*  $b$  *DO*  $c$ )  $t = WHILE\ b\ DO\ fold\ c\ t$  |' (*-lvars*  $c$ )

**lemma** *approx\_merge*:

*approx*  $t1\ s \vee approx\ t2\ s \Longrightarrow approx\ (merge\ t1\ t2)\ s$

**by** (*fastforce simp: merge\_def approx\_def*)

**lemma** *approx\_map\_le*:

$approx\ t2\ s \implies t1 \subseteq_m t2 \implies approx\ t1\ s$   
 by (clarsimp simp: approx\_def map\_le\_def dom\_def)

**lemma** *restrict\_map\_le* [intro!, simp]:  $t \mid' S \subseteq_m t$   
 by (clarsimp simp: restrict\_map\_def map\_le\_def)

**lemma** *merge\_restrict*:  
 assumes  $t1 \mid' S = t \mid' S$   
 assumes  $t2 \mid' S = t \mid' S$   
 shows  $merge\ t1\ t2 \mid' S = t \mid' S$

**proof** –  
 from *assms*  
 have  $\forall x. (t1 \mid' S)\ x = (t \mid' S)\ x$   
 and  $\forall x. (t2 \mid' S)\ x = (t \mid' S)\ x$  by *auto*  
 thus ?thesis  
 by (auto simp: merge\_def restrict\_map\_def  
 split: if\_splits)

qed

**lemma** *defs\_restrict*:

$defs\ c\ t \mid' (-\ lvars\ c) = t \mid' (-\ lvars\ c)$

**proof** (*induction c arbitrary: t*)

**case** (*Seq c1 c2*)

**hence**  $defs\ c1\ t \mid' (-\ lvars\ c1) = t \mid' (-\ lvars\ c1)$

by *simp*

**hence**  $defs\ c1\ t \mid' (-\ lvars\ c1) \mid' (-\ lvars\ c2) =$

$t \mid' (-\ lvars\ c1) \mid' (-\ lvars\ c2)$  by *simp*

**moreover**

**from** *Seq*

**have**  $defs\ c2\ (defs\ c1\ t) \mid' (-\ lvars\ c2) =$

$defs\ c1\ t \mid' (-\ lvars\ c2)$

by *simp*

**hence**  $defs\ c2\ (defs\ c1\ t) \mid' (-\ lvars\ c2) \mid' (-\ lvars\ c1) =$

$defs\ c1\ t \mid' (-\ lvars\ c2) \mid' (-\ lvars\ c1)$

by *simp*

**ultimately**

**show** ?case by (clarsimp simp: *Int\_commute*)

**next**

**case** (*If b c1 c2*)

**hence**  $defs\ c1\ t \mid' (-\ lvars\ c1) = t \mid' (-\ lvars\ c1)$  by *simp*

**hence**  $defs\ c1\ t \mid' (-\ lvars\ c1) \mid' (-\ lvars\ c2) =$

$t \mid' (-\ lvars\ c1) \mid' (-\ lvars\ c2)$  by *simp*

**moreover**

```

from If
have defs c2 t |' (- lvars c2) = t |' (- lvars c2) by simp
hence defs c2 t |' (- lvars c2) |' (-lvars c1) =
      t |' (- lvars c2) |' (-lvars c1) by simp
ultimately
show ?case by (auto simp: Int_commute intro: merge_restrict)
qed (auto split: aexp.split)

```

```

lemma big_step_pres_approx:
   $(c,s) \Rightarrow s' \Longrightarrow \text{approx } t \ s \Longrightarrow \text{approx } (\text{defs } c \ t) \ s'$ 
proof (induction arbitrary: t rule: big_step_induct)
  case Skip thus ?case by simp
next
  case Assign
  thus ?case
    by (clarsimp simp: aval_afold_N approx_def split: aexp.split)
next
  case (Seq c1 s1 s2 c2 s3)
  have approx (defs c1 t) s2 by (rule Seq.IH(1)[OF Seq.premis])
  hence approx (defs c2 (defs c1 t)) s3 by (rule Seq.IH(2))
  thus ?case by simp
next
  case (IfTrue b s c1 s')
  hence approx (defs c1 t) s' by simp
  thus ?case by (simp add: approx_merge)
next
  case (IfFalse b s c2 s')
  hence approx (defs c2 t) s' by simp
  thus ?case by (simp add: approx_merge)
next
  case WhileFalse
  thus ?case by (simp add: approx_def restrict_map_def)
next
  case (WhileTrue b s1 c s2 s3)
  hence approx (defs c t) s2 by simp
  with WhileTrue
  have approx (defs c t |' (-lvars c)) s3 by simp
  thus ?case by (simp add: defs_restrict)
qed

```

```

lemma big_step_pres_approx_restrict:
   $(c,s) \Rightarrow s' \Longrightarrow \text{approx } (t \ |' \ (-lvars \ c)) \ s \Longrightarrow \text{approx } (t \ |' \ (-lvars \ c)) \ s'$ 

```

```

proof (induction arbitrary: t rule: big_step_induct)
  case Assign
  thus ?case by (clarsimp simp: approx_def)
next
  case (Seq c1 s1 s2 c2 s3)
  hence approx (t |' (-lvars c2) |' (-lvars c1)) s1
    by (simp add: Int_commute)
  hence approx (t |' (-lvars c2) |' (-lvars c1)) s2
    by (rule Seq)
  hence approx (t |' (-lvars c1) |' (-lvars c2)) s2
    by (simp add: Int_commute)
  hence approx (t |' (-lvars c1) |' (-lvars c2)) s3
    by (rule Seq)
  thus ?case by simp
next
  case (IfTrue b s c1 s' c2)
  hence approx (t |' (-lvars c2) |' (-lvars c1)) s
    by (simp add: Int_commute)
  hence approx (t |' (-lvars c2) |' (-lvars c1)) s'
    by (rule IfTrue)
  thus ?case by (simp add: Int_commute)
next
  case (IfFalse b s c2 s' c1)
  hence approx (t |' (-lvars c1) |' (-lvars c2)) s
    by simp
  hence approx (t |' (-lvars c1) |' (-lvars c2)) s'
    by (rule IfFalse)
  thus ?case by simp
qed auto

```

```

declare assign_simp [simp]

```

```

lemma approx_eq:

```

```

  approx t  $\models$  c  $\sim$  fold c t

```

```

proof (induction c arbitrary: t)

```

```

  case SKIP show ?case by simp

```

```

next

```

```

  case Assign

```

```

  show ?case by (simp add: equiv_up_to_def)

```

```

next

```

```

  case Seq

```

```

  thus ?case by (auto intro!: equiv_up_to_seq big_step_pres_approx)

```

```

next

```

```

case If
thus ?case by (auto intro!: equiv_up_to_if_weak)
next
case (While b c)
hence approx (t |' (- lvars c)) |=
      WHILE b DO c ~ WHILE b DO fold c (t |' (- lvars c))
by (auto intro: equiv_up_to_while_weak big_step_pres_approx_restrict)
thus ?case
by (auto intro: equiv_up_to_weaken approx_map_le)
qed

```

```

lemma approx_empty [simp]:
  approx Map.empty = ( $\lambda$ _. True)
by (auto simp: approx_def)

```

```

theorem constant_folding_equiv:
  fold c Map.empty ~ c
using approx_eq [of Map.empty c]
by (simp add: equiv_up_to_True sim_sym)

```

**end**

## 11 Live Variable Analysis

```

theory Live imports Vars Big_Step
begin

```

### 11.1 Liveness Analysis

```

fun L :: com  $\Rightarrow$  vname set  $\Rightarrow$  vname set where
  L SKIP X = X |
  L (x ::= a) X = vars a  $\cup$  (X - {x}) |
  L (c1;; c2) X = L c1 (L c2 X) |
  L (IF b THEN c1 ELSE c2) X = vars b  $\cup$  L c1 X  $\cup$  L c2 X |
  L (WHILE b DO c) X = vars b  $\cup$  X  $\cup$  L c X

```

```

value show (L ("y" ::= V "z";; "x" ::= Plus (V "y") (V "z")) {"x"})

```

```

value show (L (WHILE Less (V "x") (V "x") DO "y" ::= V "z") {"x"})

```

```

fun kill :: com  $\Rightarrow$  vname 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**  $gen :: com \Rightarrow vname\ set$  **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**  $L\_gen\_kill: L\ c\ X = gen\ c \cup (X - kill\ c)$   
**by**( $induct\ c\ arbitrary:X$ ) *auto*

**lemma**  $L\_While\_pfp: L\ c\ (L\ (WHILE\ b\ DO\ c)\ X) \subseteq L\ (WHILE\ b\ DO\ c)\ X$   
**by**( $auto\ simp\ add:L\_gen\_kill$ )

**lemma**  $L\_While\_lpfp:$   
 $vars\ b \cup X \cup L\ c\ P \subseteq P \implies L\ (WHILE\ b\ DO\ c)\ X \subseteq P$   
**by**( $simp\ add: L\_gen\_kill$ )

**lemma**  $L\_While\_vars: vars\ b \subseteq L\ (WHILE\ b\ DO\ c)\ X$   
**by** *auto*

**lemma**  $L\_While\_X: X \subseteq L\ (WHILE\ b\ DO\ c)\ X$   
**by** *auto*

Disable L WHILE equation and reason only with L WHILE constraints

**declare**  $L.simps(5)[simp\ del]$

## 11.2 Correctness

**theorem**  $L\_correct:$   
 $(c,s) \Rightarrow s' \implies s = t\ on\ L\ c\ X \implies$   
 $\exists\ t'. (c,t) \Rightarrow t' \ \&\ s' = t'\ on\ X$   
**proof** ( $induction\ arbitrary: X\ t\ rule: big\_step\_induct$ )  
**case** *Skip* **then show** *?case* **by** *auto*  
**next**  
**case** *Assign* **then show** *?case*  
**by** ( $auto\ simp: ball\_Un$ )  
**next**

```

case (Seq c1 s1 s2 c2 s3 X t1)
from Seq.IH(1) Seq.premis obtain t2 where
  t12: (c1, t1)  $\Rightarrow$  t2 and s2t2: s2 = t2 on L c2 X
  by simp blast
from Seq.IH(2)[OF s2t2] obtain t3 where
  t23: (c2, t2)  $\Rightarrow$  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.IH[OF ⟨s = t on L c1 X⟩] obtain t' where
  (c1, t)  $\Rightarrow$  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  $\sim$  bval b t by simp
from IfFalse.IH[OF ⟨s = t on L c2 X⟩] obtain t' where
  (c2, t)  $\Rightarrow$  t' s' = t' on X by auto
thus ?case using ⟨ $\sim$  bval b t⟩ by auto
next
case (WhileFalse b s c)
hence  $\sim$  bval b t
  by (metis L_While_vars bval_eq_if_eq_on_vars set_mp)
thus ?case by(metis WhileFalse.premis L_While_X big_step.WhileFalse
set_mp)
next
case (WhileTrue b s1 c s2 s3 X t1)
let ?w = WHILE b DO c
from ⟨bval b s1⟩ WhileTrue.premis have bval b t1
  by (metis L_While_vars bval_eq_if_eq_on_vars set_mp)
have s1 = t1 on L c (L ?w X) using L_While_pfp WhileTrue.premis
  by (blast)
from WhileTrue.IH(1)[OF this] obtain t2 where
  (c, t1)  $\Rightarrow$  t2 s2 = t2 on L ?w X by auto
from WhileTrue.IH(2)[OF this(2)] obtain t3 where (?w, t2)  $\Rightarrow$  t3 s3
= t3 on X
  by auto
with ⟨bval b t1⟩ ⟨(c, t1)  $\Rightarrow$  t2⟩ show ?case by auto
qed

```

### 11.3 Program Optimization

Burying assignments to dead variables:

```

fun bury :: com  $\Rightarrow$  vname set  $\Rightarrow$  com where
bury SKIP X = SKIP |
bury (x ::= a) X = (if x  $\in$  X then x ::= a else SKIP) |
bury (c1;; c2) X = (bury c1 (L c2 X));; bury c2 X) |
bury (IF b THEN c1 ELSE c2) X = IF b THEN bury c1 X ELSE bury c2 X |
bury (WHILE b DO c) X = WHILE b DO bury c (L (WHILE b DO c) X)

```

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

**theorem** *bury\_correct*:

```

(c,s)  $\Rightarrow$  s'  $\Longrightarrow$  s = t on L c X  $\Longrightarrow$ 
 $\exists$  t'. (bury c X,t)  $\Rightarrow$  t' & s' = t' on X

```

**proof** (*induction 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** (*Seq c1 s1 s2 c2 s3 X t1*)

**from** *Seq.IH(1) Seq.prem*s **obtain** *t2* **where**

*t12*: (bury c1 (L c2 X), t1)  $\Rightarrow$  t2 **and** *s2t2*: s2 = t2 on L c2 X

**by** *simp blast*

**from** *Seq.IH(2)[OF s2t2]* **obtain** *t3* **where**

*t23*: (bury c2 X, t2)  $\Rightarrow$  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.IH[OF <s = t on L c1 X>]* **obtain** *t'* **where**

(bury c1 X, t)  $\Rightarrow$  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**  $\sim$ *bval b t* **by** *simp*

**from** *IfFalse.IH[OF <s = t on L c2 X>]* **obtain** *t'* **where**

(bury c2 X, t)  $\Rightarrow$  t' s' = t' on X **by** *auto*

**thus** ?case **using** *< $\sim$ bval b t>* **by** *auto*

**next**  
 case (*WhileFalse* *b s c*)  
 hence  $\sim$  *bval* *b t* **by** (*metis* *L\_While\_vars bval\_eq\_if\_eq\_on\_vars set\_mp*)  
 thus *?case*  
 by *simp* (*metis* *L\_While\_X WhileFalse.premis big\_step.WhileFalse set\_mp*)  
**next**  
 case (*WhileTrue* *b s1 c s2 s3 X t1*)  
 let *?w* = *WHILE* *b DO c*  
 from  $\langle$ *bval* *b s1* $\rangle$  *WhileTrue.premis* **have** *bval* *b t1*  
 by (*metis* *L\_While\_vars bval\_eq\_if\_eq\_on\_vars set\_mp*)  
 have *s1 = t1* on *L c* (*L ?w X*)  
 using *L\_While\_pfp WhileTrue.premis* **by** *blast*  
 from *WhileTrue.IH(1)[OF this]* **obtain** *t2* **where**  
 (*bury* *c* (*L ?w X*), *t1*)  $\Rightarrow$  *t2 s2 = t2* on *L ?w X* **by** *auto*  
 from *WhileTrue.IH(2)[OF this(2)]* **obtain** *t3*  
 where (*bury* *?w X, t2*)  $\Rightarrow$  *t3 s3 = t3* on *X*  
 by *auto*  
 with  $\langle$ *bval* *b t1* $\rangle$   $\langle$ (*bury* *c* (*L ?w X*), *t1*)  $\Rightarrow$  *t2* $\rangle$  **show** *?case* **by** *auto*  
**qed**

**corollary** *final\_bury\_correct*: (*c, s*)  $\Rightarrow$  *s'*  $\Longrightarrow$  (*bury* *c UNIV, s*)  $\Rightarrow$  *s'*  
**using** *bury\_correct[of c s s' UNIV]*  
**by** (*auto simp: fun\_eq\_iff[symmetric]*)

Now the opposite direction.

**lemma** *SKIP\_bury[simp]*:  
*SKIP = bury* *c X*  $\longleftrightarrow$  *c = SKIP* | ( $\exists$  *x a. c = x::=a* & *x*  $\notin$  *X*)  
**by** (*cases* *c*) *auto*

**lemma** *Assign\_bury[simp]*: *x::=a = bury* *c X*  $\longleftrightarrow$  *c = x::=a*  $\wedge$  *x*  $\in$  *X*  
**by** (*cases* *c*) *auto*

**lemma** *Seq\_bury[simp]*: *bc1;;bc2 = bury* *c X*  $\longleftrightarrow$   
 ( $\exists$  *c1 c2. c = c1;;c2* & *bc2 = bury* *c2 X* & *bc1 = bury* *c1* (*L c2 X*))  
**by** (*cases* *c*) *auto*

**lemma** *If\_bury[simp]*: *IF b THEN bc1 ELSE bc2 = bury* *c X*  $\longleftrightarrow$   
 ( $\exists$  *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*  $\longleftrightarrow$   
 ( $\exists$  *c'. c = WHILE b DO c'* & *bc' = bury* *c'* (*L* (*WHILE b DO c'*) *X*))  
**by** (*cases* *c*) *auto*

**theorem** *bury\_correct2*:

$(\text{bury } c \ X, s) \Rightarrow s' \implies s = t \text{ on } L \ c \ X \implies$   
 $\exists t'. (c, t) \Rightarrow t' \ \& \ s' = t' \text{ on } X$

**proof** (*induction 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** (*Seq bc1 s1 s2 bc2 s3 c X t1*)

**then obtain** *c1 c2* **where**  $c: c = c1;;c2$

**and** *bc2*:  $bc2 = \text{bury } c2 \ X$  **and** *bc1*:  $bc1 = \text{bury } c1 \ (L \ c2 \ X)$  **by** *auto*

**note**  $IH = \text{Seq.hyps}(2,4)$

**from**  $IH(1)[OF \ bc1, \ of \ t1]$  *Seq.prem*s *c* **obtain** *t2* **where**  
 $t12: (c1, t1) \Rightarrow t2$  **and**  $s2t2: s2 = t2 \text{ on } L \ c2 \ X$  **by** *auto*

**from**  $IH(2)[OF \ bc2 \ s2t2]$  **obtain** *t3* **where**  
 $t23: (c2, t2) \Rightarrow t3$  **and**  $s3t3: s3 = t3 \text{ on } X$

**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*:  $bc1 = \text{bury } c1 \ X$  **and** *bc2*:  $bc2 = \text{bury } c2 \ X$  **by** *auto*

**have**  $s = t \text{ on vars } b \ s = t \text{ on } L \ c1 \ X$  **using** *IfTrue.prem*s *c* **by** *auto*

**from** *bval\_eq\_if\_eq\_on\_vars*[*OF this(1)*] *IfTrue(1)* **have**  $\text{bval } b \ t$  **by** *simp*

**note**  $IH = \text{IfTrue.hyps}(3)$

**from**  $IH[OF \ bc1 \ \langle s = t \text{ on } L \ c1 \ X \rangle]$  **obtain** *t'* **where**  
 $(c1, t) \Rightarrow t' \ s' = t' \text{ on } X$  **by** *auto*

**thus** *?case* **using** *c*  $\langle \text{bval } b \ t \rangle$  **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*:  $bc1 = \text{bury } c1 \ X$  **and** *bc2*:  $bc2 = \text{bury } c2 \ X$  **by** *auto*

**have**  $s = t \text{ on vars } b \ s = t \text{ on } L \ c2 \ X$  **using** *IfFalse.prem*s *c* **by** *auto*

**from** *bval\_eq\_if\_eq\_on\_vars*[*OF this(1)*] *IfFalse(1)* **have**  $\sim \text{bval } b \ t$  **by** *simp*

**note**  $IH = \text{IfFalse.hyps}(3)$

**from**  $IH[OF \ bc2 \ \langle s = t \text{ on } L \ c2 \ X \rangle]$  **obtain** *t'* **where**  
 $(c2, t) \Rightarrow t' \ s' = t' \text{ on } X$  **by** *auto*

**thus** *?case* **using** *c*  $\langle \sim \text{bval } b \ t \rangle$  **by** *auto*

**next**

**case** (*WhileFalse b s c*)

**hence**  $\sim \text{bval } b \ t$

**by** *auto* (*metis L\_While\_vars bval\_eq\_if\_eq\_on\_vars set\_rev\_mp*)

```

thus ?case using WhileFalse
  by auto (metis L_While_X big_step.WhileFalse set_mp)
next
  case (WhileTrue b s1 bc' s2 s3 w X t1)
  then obtain c' where w: w = WHILE b DO c'
    and bc': bc' = bury c' (L (WHILE b DO c') X) by auto
  from ⟨bval b s1⟩ WhileTrue.premis w have bval b t1
    by auto (metis L_While_vars bval_eq_if_eq_on_vars set_mp)
  note IH = WhileTrue.hyps(3,5)
  have s1 = t1 on L c' (L w X)
    using L_While_pfp WhileTrue.premis w by blast
  with IH(1)[OF bc', of t1] w obtain t2 where
    (c', t1) ⇒ t2 s2 = t2 on L w X by auto
  from IH(2)[OF WhileTrue.hyps(6), of t2] w this(2) obtain t3
    where (w,t2) ⇒ t3 s3 = t3 on X
    by auto
  with ⟨bval b t1⟩ ⟨(c', t1) ⇒ t2⟩ w show ?case by auto
qed

```

```

corollary final_bury_correct2: (bury c UNIV, s) ⇒ s' ⇒ (c, s) ⇒ s'
using bury_correct2[of c UNIV]
by (auto simp: fun_eq_iff[symmetric])

```

```

corollary bury_sim: bury c UNIV ∼ c
by(metis final_bury_correct final_bury_correct2)

```

**end**

## 11.4 True Liveness Analysis

```

theory Live_True
imports HOL-Library.While_Combinator Vars Big_Step
begin

```

### 11.4.1 Analysis

```

fun L :: com ⇒ vname set ⇒ vname set where
  L SKIP X = X |
  L (x ::= a) X = (if x ∈ X then vars a ∪ (X - {x}) else X) |
  L (c1;; c2) X = L c1 (L c2 X) |
  L (IF b THEN c1 ELSE c2) X = vars b ∪ L c1 X ∪ L c2 X |
  L (WHILE b DO c) X = lfp(λY. vars b ∪ X ∪ L c Y)

```

```

lemma L_mono: mono (L c)

```

```

proof–
  have  $X \subseteq Y \implies L\ c\ X \subseteq L\ c\ Y$  for  $X\ Y$ 
  proof(induction c arbitrary: X Y)
    case (While b c)
    show ?case
    proof(simp, rule lfp_mono)
      fix  $Z$  show  $\text{vars } b \cup X \cup L\ c\ Z \subseteq \text{vars } b \cup Y \cup L\ c\ Z$ 
      using While by auto
    qed
  next
    case If thus ?case by(auto simp: subset_iff)
  qed auto
  thus ?thesis by(rule monoI)
qed

```

```

lemma mono_union_L:
  mono ( $\lambda Y. X \cup L\ c\ Y$ )
by (metis (no_types) L_mono mono_def order_eq_iff set_eq_subset sup_mono)

```

```

lemma L_While_unfold:
   $L\ (WHILE\ b\ DO\ c)\ X = \text{vars } b \cup X \cup L\ c\ (L\ (WHILE\ b\ DO\ c)\ X)$ 
by(metis lfp_unfold[OF mono_union_L] L.simps(5))

```

```

lemma L_While_pfp:  $L\ c\ (L\ (WHILE\ b\ DO\ c)\ X) \subseteq L\ (WHILE\ b\ DO\ c)\ X$ 
using L_While_unfold by blast

```

```

lemma L_While_vars:  $\text{vars } b \subseteq L\ (WHILE\ b\ DO\ c)\ X$ 
using L_While_unfold by blast

```

```

lemma L_While_X:  $X \subseteq L\ (WHILE\ b\ DO\ c)\ X$ 
using L_While_unfold by blast

```

Disable  $L\ WHILE$  equation and reason only with  $L\ WHILE$  constraints:

```

declare L.simps(5)[simp del]

```

#### 11.4.2 Correctness

```

theorem L_correct:
   $(c,s) \Rightarrow s' \implies s = t\ \text{on } L\ c\ X \implies$ 
   $\exists t'. (c,t) \Rightarrow t' \ \& \ s' = t'\ \text{on } X$ 
proof (induction 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 (Seq c1 s1 s2 c2 s3 X t1)
from Seq.IH(1) Seq.prems obtain t2 where
  t12: (c1, t1) ⇒ t2 and s2t2: s2 = t2 on L c2 X
  by simp blast
from Seq.IH(2)[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 and 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.IH[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  $\sim$ bval b t by simp
from IfFalse.IH[OF ⟨s = t on L c2 X⟩] obtain t' where
  (c2, t) ⇒ t' s' = t' on X by auto
thus ?case using ⟨ $\sim$ bval b t⟩ by auto
next
case (WhileFalse b s c)
hence  $\sim$  bval b t
  by (metis L_While_vars bval_eq_if_eq_on_vars set_mp)
thus ?case using WhileFalse.prems L_While_X[of X b c] by auto
next
case (WhileTrue b s1 c s2 s3 X t1)
let ?w = WHILE b DO c
from ⟨bval b s1⟩ WhileTrue.prems have bval b t1
  by (metis L_While_vars bval_eq_if_eq_on_vars set_mp)
have s1 = t1 on L c (L ?w X) using L_While_pfp WhileTrue.prems
  by (blast)
from WhileTrue.IH(1)[OF this] obtain t2 where
  (c, t1) ⇒ t2 s2 = t2 on L ?w X by auto
from WhileTrue.IH(2)[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

```



### 11.4.3 Executability

**lemma** *L\_subset\_vars*:  $L\ c\ X \subseteq rvars\ c \cup X$

**proof**(*induction c arbitrary: X*)

**case** (*While b c*)

**have**  $lfp(\lambda Y. vars\ b \cup X \cup L\ c\ Y) \subseteq vars\ b \cup rvars\ c \cup X$

**using** *While.IH*[*of vars b ∪ rvars c ∪ X*]

**by** (*auto intro!: lfp\_lowerbound*)

**thus** *?case* **by** (*simp add: L.simps(5)*)

**qed** *auto*

Make *L* executable by replacing *lfp* with the *while* combinator from theory *HOL-Library.While\_Combinator*. The *while* combinator obeys the recursion equation

$while\ b\ c\ s = (if\ b\ s\ then\ while\ b\ c\ (c\ s)\ else\ s)$

and is thus executable.

**lemma** *L\_While*: **fixes** *b c X*

**assumes** *finite X* **defines**  $f == \lambda Y. vars\ b \cup X \cup L\ c\ Y$

**shows**  $L\ (WHILE\ b\ DO\ c)\ X = while\ (\lambda Y. f\ Y \neq Y)\ f\ \{\}\ (is\_ =\ ?r)$

**proof** –

**let**  $?V = vars\ b \cup rvars\ c \cup X$

**have**  $lfp\ f = ?r$

**proof**(*rule lfp\_while*[**where**  $C = ?V$ ])

**show** *mono f* **by**(*simp add: f\_def mono\_union\_L*)

**next**

**fix** *Y* **show**  $Y \subseteq ?V \implies f\ Y \subseteq ?V$

**unfolding** *f\_def* **using** *L\_subset\_vars*[*of c*] **by** *blast*

**next**

**show** *finite ?V* **using**  $\langle finite\ X \rangle$  **by** *simp*

**qed**

**thus** *?thesis* **by** (*simp add: f\_def L.simps(5)*)

**qed**

**lemma** *L\_While\_let*:  $finite\ X \implies L\ (WHILE\ b\ DO\ c)\ X =$

(*let*  $f = (\lambda Y. vars\ b \cup X \cup L\ c\ Y)$

*in*  $while\ (\lambda Y. f\ Y \neq Y)\ f\ \{\}$ )

**by**(*simp add: L\_While*)

**lemma** *L\_While\_set*:  $L\ (WHILE\ b\ DO\ c)\ (set\ xs) =$

(*let*  $f = (\lambda Y. vars\ b \cup set\ xs \cup L\ c\ Y)$

*in*  $while\ (\lambda Y. f\ Y \neq Y)\ f\ \{\}$ )

**by**(*rule L\_While\_let, simp*)

Replace the equation for  $L\ (WHILE\ \dots)$  by the executable *L\_While\_set*:

**lemmas** [code] = L.simps(1-4) L.While\_set

Sorry, this syntax is odd.

A test:

**lemma** (let b = Less (N 0) (V "y"); c = "y" ::= V "x"; "x" ::= V "z"  
in L (WHILE b DO c) {"y"}) = {"x", "y", "z"}  
**by** eval

#### 11.4.4 Limiting the number of iterations

The final parameter is the default value:

**fun** iter :: ('a ⇒ 'a) ⇒ nat ⇒ 'a ⇒ 'a ⇒ 'a **where**  
iter f 0 p d = d |  
iter f (Suc n) p d = (if f p = p then p else iter f n (f p) d)

A version of L with a bounded number of iterations (here: 2) in the WHILE case:

**fun** Lb :: com ⇒ vname set ⇒ vname set **where**  
Lb SKIP X = X |  
Lb (x ::= a) X = (if x ∈ X then X - {x} ∪ vars a else X) |  
Lb (c<sub>1</sub>; c<sub>2</sub>) X = (Lb c<sub>1</sub> ∘ Lb c<sub>2</sub>) X |  
Lb (IF b THEN c<sub>1</sub> ELSE c<sub>2</sub>) X = vars b ∪ Lb c<sub>1</sub> X ∪ Lb c<sub>2</sub> X |  
Lb (WHILE b DO c) X = iter (λA. vars b ∪ X ∪ Lb c A) 2 {} (vars b ∪  
rvars c ∪ X)

Lb (and iter) is not monotone!

**lemma** let w = WHILE Bc False DO ("x" ::= V "y"; "z" ::= V "x")  
in ¬ (Lb w {"z"} ⊆ Lb w {"y", "z"})  
**by** eval

**lemma** lfp\_subset\_iter:

[ mono f; !!X. f X ⊆ f' X; lfp f ⊆ D ] ⇒ lfp f ⊆ iter f' n A D

**proof**(induction n arbitrary: A)

case 0 **thus** ?case **by** simp

**next**

case Suc **thus** ?case **by** simp (metis lfp\_lowerbound)

**qed**

**lemma** L c X ⊆ Lb c X

**proof**(induction c arbitrary: X)

case (While b c)

let ?f = λA. vars b ∪ X ∪ L c A

let ?fb = λA. vars b ∪ X ∪ Lb c A

**show** ?case

```

proof (simp add: L.simps(5), rule lfp_subset_iter[OF mono_union_L])
  show !!X. ?f X ⊆ ?fb X using While.IH by blast
  show lfp ?f ⊆ vars b ∪ rvars c ∪ X
    by (metis (full_types) L.simps(5) L_subset_vars rvars.simps(5))
qed
next
  case Seq thus ?case by simp (metis (full_types) L_mono monoD subset_trans)
qed auto

end

```

## 12 Denotational Semantics of Commands

```

theory Denotational imports Big_Step begin

```

```

type_synonym com_den = (state × state) set

```

```

definition W :: (state ⇒ bool) ⇒ com_den ⇒ (com_den ⇒ com_den)

```

```

where

```

```

W db dc = (λdw. {(s,t). if db s then (s,t) ∈ dc O dw else s=t})

```

```

fun D :: com ⇒ com_den where

```

```

D SKIP = Id |

```

```

D (x ::= a) = {(s,t). t = s(x := aval a s)} |

```

```

D (c1;;c2) = D(c1) O D(c2) |

```

```

D (IF b THEN c1 ELSE c2)

```

```

= {(s,t). if bval b s then (s,t) ∈ D c1 else (s,t) ∈ D c2} |

```

```

D (WHILE b DO c) = lfp (W (bval b) (D c))

```

```

lemma W_mono: mono (W b r)

```

```

by (unfold W_def mono_def) auto

```

```

lemma D_While_If:

```

```

D (WHILE b DO c) = D(IF b THEN c;;WHILE b DO c ELSE SKIP)

```

```

proof–

```

```

let ?w = WHILE b DO c let ?f = W (bval b) (D c)

```

```

have D ?w = lfp ?f by simp

```

```

also have ... = ?f (lfp ?f) by(rule lfp_unfold [OF W_mono])

```

```

also have ... = D(IF b THEN c;;?w ELSE SKIP) by (simp add: W_def)

```

```

finally show ?thesis .

```

```

qed

```

Equivalence of denotational and big-step semantics:

**lemma** *D-if-big-step*:  $(c,s) \Rightarrow t \implies (s,t) \in D(c)$   
**proof** (*induction rule: big\_step\_induct*)  
  **case** *WhileFalse*  
  **with** *D-While-If* **show** *?case* **by** *auto*  
**next**  
  **case** *WhileTrue*  
  **show** *?case* **unfolding** *D-While-If* **using** *WhileTrue* **by** *auto*  
**qed** *auto*

**abbreviation** *Big\_step* ::  $com \Rightarrow com\_den$  **where**  
*Big\_step* *c*  $\equiv \{(s,t). (c,s) \Rightarrow t\}$

**lemma** *Big\_step-if-D*:  $(s,t) \in D(c) \implies (s,t) \in Big\_step\ c$   
**proof** (*induction c arbitrary: s t*)  
  **case** *Seq* **thus** *?case* **by** *fastforce*  
**next**  
  **case** (*While b c*)  
  **let** *?B* = *Big\_step* (*WHILE b DO c*) **let** *?f* = *W* (*bval b*) (*D c*)  
  **have** *?f ?B*  $\subseteq$  *?B* **using** *While.IH* **by** (*auto simp: W\_def*)  
  **from** *lfp\_lowerbound*[**where** *?f* = *?f*, *OF this*] *While.prem*s  
  **show** *?case* **by** *auto*  
**qed** (*auto split: if\_splits*)

**theorem** *denotational-is-big-step*:  
 $(s,t) \in D(c) = ((c,s) \Rightarrow t)$   
**by** (*metis D-if-big\_step Big\_step-if-D[simplified]*)

**corollary** *equiv\_c\_iff\_equal\_D*:  $(c1 \sim c2) \longleftrightarrow D\ c1 = D\ c2$   
**by**(*simp add: denotational-is-big\_step[symmetric] set\_eq\_iff*)

## 12.1 Continuity

**definition** *chain* ::  $(nat \Rightarrow 'a\ set) \Rightarrow bool$  **where**  
*chain* *S* =  $(\forall i. S\ i \subseteq S(Suc\ i))$

**lemma** *chain\_total*:  $chain\ S \implies S\ i \leq S\ j \vee S\ j \leq S\ i$   
**by** (*metis chain\_def le\_cases lift\_Suc\_mono\_le*)

**definition** *cont* ::  $('a\ set \Rightarrow 'b\ set) \Rightarrow bool$  **where**  
*cont* *f* =  $(\forall S. chain\ S \longrightarrow f(UN\ n. S\ n) = (UN\ n. f(S\ n)))$

**lemma** *mono-if-cont*: **fixes** *f* ::  $'a\ set \Rightarrow 'b\ set$   
  **assumes** *cont f* **shows** *mono f*  
**proof**

```

fix a b :: 'a set assume a  $\subseteq$  b
let ?S =  $\lambda n::nat.$  if n=0 then a else b
have chain ?S using  $\langle a \subseteq b \rangle$  by (auto simp: chain_def)
hence f (UN n. ?S n) = (UN n. f (?S n))
  using assms by (simp add: cont_def)
moreover have (UN n. ?S n) = b using  $\langle a \subseteq b \rangle$  by (auto split: if_splits)
moreover have (UN n. f (?S n)) = f a  $\cup$  f b by (auto split: if_splits)
ultimately show f a  $\subseteq$  f b by (metis Un_upper1)
qed

```

```

lemma chain_iterates: fixes f :: 'a set  $\Rightarrow$  'a set
  assumes mono f shows chain( $\lambda n.$  (f^n) {})

```

```

proof -

```

```

  have (f^n) {}  $\subseteq$  (f^Suc n) {} for n

```

```

  proof (induction n)

```

```

    case 0 show ?case by simp

```

```

  next

```

```

    case (Suc n) thus ?case using assms by (auto simp: mono_def)

```

```

  qed

```

```

  thus ?thesis by (auto simp: chain_def assms)

```

```

qed

```

```

theorem lfp_if_cont:

```

```

  assumes cont f shows lfp f = (UN n. (f^n) {}) (is _ = ?U)

```

```

proof

```

```

  from assms mono_if_cont

```

```

  have mono: (f^n) {}  $\subseteq$  (f^Suc n) {} for n

```

```

    using funpow_decreasing [of n Suc n] by auto

```

```

  show lfp f  $\subseteq$  ?U

```

```

  proof (rule lfp_lowerbound)

```

```

    have f ?U = (UN n. (f^Suc n) {})

```

```

      using chain_iterates[OF mono_if_cont[OF assms]] assms

```

```

      by (simp add: cont_def)

```

```

    also have ... = (f^0) {}  $\cup$  ... by simp

```

```

    also have ... = ?U

```

```

      using mono by auto (metis funpow_simps_right(2) funpow_swap1
o_apply)

```

```

    finally show f ?U  $\subseteq$  ?U by simp

```

```

  qed

```

```

next

```

```

  have (f^n) {}  $\subseteq$  p if f p  $\subseteq$  p for n p

```

```

  proof -

```

```

    show ?thesis

```

```

    proof (induction n)

```

```

    case 0 show ?case by simp
  next
    case Suc
    from monoD[OF mono_if_cont[OF assms] Suc] ⟨f p ⊆ p⟩
    show ?case by simp
  qed
qed
thus ?U ⊆ lfp f by(auto simp: lfp_def)
qed

```

```

lemma cont_W: cont(W b r)
by(auto simp: cont_def W_def)

```

## 12.2 The denotational semantics is deterministic

```

lemma single_valued_UN_chain:
  assumes chain S (∧n. single_valued (S n))
  shows single_valued(UN n. S n)
proof(auto simp: single_valued_def)
  fix m n x y z assume (x, y) ∈ S m (x, z) ∈ S n
  with chain_total[OF assms(1), of m n] assms(2)
  show y = z by (auto simp: single_valued_def)
qed

```

```

lemma single_valued_lfp: fixes f :: com_den ⇒ com_den
  assumes cont f ∧ r. single_valued r ⇒ single_valued (f r)
  shows single_valued(lfp f)
  unfolding lfp_if_cont[OF assms(1)]
proof(rule single_valued_UN_chain[OF chain_iterates[OF mono_if_cont[OF
  assms(1)]]])
  fix n show single_valued ((f ^^ n) {})
  by(induction n)(auto simp: assms(2))
qed

```

```

lemma single_valued_D: single_valued (D c)
proof(induction c)
  case Seq thus ?case by(simp add: single_valued_relcomp)
next
  case (While b c)
  let ?f = W (bval b) (D c)
  have single_valued (lfp ?f)
  proof(rule single_valued_lfp[OF cont_W])
    show ∧r. single_valued r ⇒ single_valued (?f r)
    using While.IH by(force simp: single_valued_def W_def)
  qed

```

```

    qed
  thus ?case by simp
qed (auto simp add: single_valued_def)

end

```

## 13 Hoare Logic

### 13.1 Hoare Logic for Partial Correctness

```

theory Hoare imports Big_Step begin

```

```

type_synonym assn = state  $\Rightarrow$  bool

```

**definition**

```

hoare_valid :: assn  $\Rightarrow$  com  $\Rightarrow$  assn  $\Rightarrow$  bool ( $\models \{(1\_)\} / (-) / \{(1\_)\}$  50) where
 $\models \{P\} c \{Q\} = (\forall s t. P s \wedge (c, s) \Rightarrow t \longrightarrow Q t)$ 

```

**abbreviation** *state\_subst* :: state  $\Rightarrow$  aexp  $\Rightarrow$  vname  $\Rightarrow$  state

```

(-['/']) [1000,0,0] 999)

```

**where**  $s[a/x] == s(x := \text{aval } a \ s)$

**inductive**

```

hoare :: assn  $\Rightarrow$  com  $\Rightarrow$  assn  $\Rightarrow$  bool ( $\vdash \{(1\_)\} / (-) / \{(1\_)\}$  50)

```

**where**

```

Skip:  $\vdash \{P\} \text{SKIP} \{P\} \mid$ 

```

```

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

```

```

Seq:  $\llbracket \vdash \{P\} c_1 \{Q\}; \vdash \{Q\} c_2 \{R\} \rrbracket$ 
 $\implies \vdash \{P\} c_1;;c_2 \{R\} \mid$ 

```

```

If:  $\llbracket \vdash \{\lambda s. P s \wedge \text{bval } b \ s\} c_1 \{Q\}; \vdash \{\lambda s. P s \wedge \neg \text{bval } b \ s\} c_2 \{Q\} \rrbracket$ 
 $\implies \vdash \{P\} \text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2 \{Q\} \mid$ 

```

```

While:  $\vdash \{\lambda s. P s \wedge \text{bval } b \ s\} c \{P\} \implies$ 
 $\vdash \{P\} \ \text{WHILE } b \ \text{DO } c \{\lambda s. P s \wedge \neg \text{bval } b \ s\} \mid$ 

```

```

conseq:  $\llbracket \forall s. P' s \longrightarrow P s; \vdash \{P\} c \{Q\}; \forall s. Q s \longrightarrow Q' s \rrbracket$ 
 $\implies \vdash \{P'\} c \{Q'\}$ 

```

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

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

**lemma** *strengthen\_pre*:  
 $\llbracket \forall s. P' s \longrightarrow P s; \vdash \{P\} c \{Q\} \rrbracket \Longrightarrow \vdash \{P'\} c \{Q\}$   
**by** (*blast intro: conseq*)

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

**end**

## 13.2 Examples

**theory** *Hoare\_Examples* **imports** *Hoare* **begin**

**hide\_const** (**open**) *sum*

Summing up the first  $x$  natural numbers in variable  $y$ .

**fun** *sum* :: *int*  $\Rightarrow$  *int* **where**  
*sum*  $i = (\text{if } i \leq 0 \text{ then } 0 \text{ else } \text{sum } (i - 1) + i)$

**lemma** *sum\_simps*[*simp*]:  
 $0 < i \Longrightarrow \text{sum } i = \text{sum } (i - 1) + i$   
 $i \leq 0 \Longrightarrow \text{sum } i = 0$   
**by**(*simp-all*)

**declare** *sum\_simps*[*simp del*]

**abbreviation** *wsum* ==  
 $\text{WHILE } \text{Less } (N\ 0) (V\ 'x')$   
 $\text{DO } ('y' ::= \text{Plus } (V\ 'y') (V\ 'x'));$   
 $'x' ::= \text{Plus } (V\ 'x') (N\ (-1))$ )



### 13.2.1 Proof by Operational Semantics

The behaviour of the loop is proved by induction:

```

lemma while_sum:
  (wsum, s)  $\Rightarrow$  t  $\Longrightarrow$  t "y" = s "y" + sum(s "x")
apply(induction wsum s t rule: big_step_induct)
apply(auto)
done

```

We were lucky that the proof was 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 ("y" ::= N 0;; wsum, s)  $\Rightarrow$  t
  shows t "y" = sum (s "x")
proof -
  from assms have (wsum,s("y"::=0))  $\Rightarrow$  t by auto
  from while_sum[OF this] show ?thesis by simp
qed

```

### 13.2.2 Proof by Hoare Logic

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

```

lemma  $\vdash$  { $\lambda s. s$  "x" = n} "y" ::= N 0;; wsum { $\lambda s. s$  "y" = sum n}
apply(rule Seq)
prefer 2
apply(rule While' [where P =  $\lambda s. (s$  "y" = sum n - sum(s "x"))])
apply(rule Seq)
prefer 2
apply(rule Assign)
apply(rule Assign')
apply simp
apply simp
apply(rule Assign')
apply simp
done

```

The proof is intentionally an apply script 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**

### 13.3 Soundness and Completeness

**theory** *Hoare\_Sound\_Complete*

**imports** *Hoare*

**begin**

#### 13.3.1 Soundness

**lemma** *hoare\_sound*:  $\vdash \{P\}c\{Q\} \implies \models \{P\}c\{Q\}$

**proof**(*induction rule: hoare.induct*)

**case** (*While P b c*)

**have** (*WHILE b DO c,s*)  $\Rightarrow t \implies P s \implies P t \wedge \neg \text{bval } b t$  **for** *s t*

**proof**(*induction WHILE b DO c s t rule: big\_step\_induct*)

**case** *WhileFalse* **thus** *?case* **by** *blast*

**next**

**case** *WhileTrue* **thus** *?case*

**using** *While.IH* **unfolding** *hoare\_valid\_def* **by** *blast*

**qed**

**thus** *?case* **unfolding** *hoare\_valid\_def* **by** *blast*

**qed** (*auto simp: hoare\_valid\_def*)

#### 13.3.2 Weakest Precondition

**definition** *wp* :: *com*  $\Rightarrow$  *assn*  $\Rightarrow$  *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\_Seq[simp]*: *wp (c<sub>1</sub>;;c<sub>2</sub>) Q* = *wp c<sub>1</sub> (wp c<sub>2</sub> Q)*

**by** (*rule ext*) (*auto simp: wp\_def*)

**lemma** *wp\_If[simp]*:

*wp (IF b THEN c<sub>1</sub> ELSE c<sub>2</sub>) Q* =

( $\lambda s. \text{if } \text{bval } b s \text{ then } wp c_1 Q s \text{ else } wp c_2 Q s$ )

**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]*:  $bval\ b\ s \implies$   
 $wp (WHILE\ b\ DO\ c)\ Q\ s = wp (c;;\ WHILE\ b\ DO\ c)\ Q\ s$   
**by**(*simp add: wp\_While\_If*)

**lemma** *wp\_While\_False[simp]*:  $\neg\ bval\ b\ s \implies wp (WHILE\ b\ DO\ c)\ Q\ s =$   
 $Q\ s$   
**by**(*simp add: wp\_While\_If*)

### 13.3.3 Completeness

**lemma** *wp\_is\_pre*:  $\vdash \{wp\ c\ Q\} c \{Q\}$   
**proof**(*induction c arbitrary: Q*)  
**case If thus ?case** **by**(*auto intro: conseq*)  
**next**  
**case** (*While b c*)  
**let**  $?w = WHILE\ b\ DO\ c$   
**show**  $\vdash \{wp\ ?w\ Q\} ?w \{Q\}$   
**proof**(*rule While'*)  
**show**  $\vdash \{\lambda s. wp\ ?w\ Q\ s \wedge\ bval\ b\ s\} c \{wp\ ?w\ Q\}$   
**proof**(*rule strengthen\_pre[OF - While.IH]*)  
**show**  $\forall s. wp\ ?w\ Q\ s \wedge\ bval\ b\ s \longrightarrow wp\ c\ (wp\ ?w\ Q)\ s$  **by** *auto*  
**qed**  
**show**  $\forall s. wp\ ?w\ Q\ s \wedge\ \neg\ bval\ b\ s \longrightarrow Q\ s$  **by** *auto*  
**qed**  
**qed** *auto*

**lemma** *hoare\_complete*: **assumes**  $\models \{P\}c\{Q\}$  **shows**  $\vdash \{P\}c\{Q\}$   
**proof**(*rule strengthen\_pre*)  
**show**  $\forall s. P\ s \longrightarrow wp\ c\ Q\ s$  **using** *assms*  
**by** (*auto simp: hoare\_valid\_def wp\_def*)  
**show**  $\vdash \{wp\ c\ Q\} c \{Q\}$  **by**(*rule wp\_is\_pre*)  
**qed**

**corollary** *hoare\_sound\_complete*:  $\vdash \{P\}c\{Q\} \longleftrightarrow \models \{P\}c\{Q\}$   
**by** (*metis hoare\_complete hoare\_sound*)

**end**

## 13.4 Verification Condition Generation

theory *VCG* imports *Hoare* begin

### 13.4.1 Annotated Commands

Commands where loops are annotated with invariants.

```
datatype acom =
  Askip                (SKIP) |
  Aassign vname aexp  ((_ ::= _) [1000, 61] 61) |
  Aseq  acom acom    (_;;/ _ [60, 61] 60) |
  Aif bexp acom acom ((IF _/ THEN _/ ELSE _) [0, 0, 61] 61) |
  Awhile assn bexp acom (({_}/ WHILE _/ DO _) [0, 0, 61] 61)
```

**notation** *com.SKIP* (*SKIP*)

Strip annotations:

```
fun strip :: acom ⇒ com where
strip SKIP = SKIP |
strip (x ::= a) = (x ::= a) |
strip (C1;; C2) = (strip C1;; strip C2) |
strip (IF b THEN C1 ELSE C2) = (IF b THEN strip C1 ELSE strip C2) |
strip ({_} WHILE b DO C) = (WHILE b DO strip C)
```

### 13.4.2 Weakest Precondition and Verification Condition

Weakest precondition:

```
fun pre :: acom ⇒ assn ⇒ assn where
pre SKIP Q = Q |
pre (x ::= a) Q = (λs. Q(s(x := aval a s))) |
pre (C1;; C2) Q = pre C1 (pre C2 Q) |
pre (IF b THEN C1 ELSE C2) Q =
  (λs. if bval b s then pre C1 Q s else pre C2 Q s) |
pre ({I} WHILE b DO C) Q = I
```

Verification condition:

```
fun vc :: acom ⇒ assn ⇒ bool where
vc SKIP Q = True |
vc (x ::= a) Q = True |
vc (C1;; C2) Q = (vc C1 (pre C2 Q) ∧ vc C2 Q) |
vc (IF b THEN C1 ELSE C2) Q = (vc C1 Q ∧ vc C2 Q) |
vc ({I} WHILE b DO C) Q =
  ((∀ s. (I s ∧ bval b s → pre C I s) ∧
    (I s ∧ ¬ bval b s → Q s)) ∧
   vc C I)
```

### 13.4.3 Soundness

**lemma** *vc\_sound*:  $vc\ C\ Q \implies \vdash \{pre\ C\ Q\}\ strip\ C\ \{Q\}$   
**proof** (*induction C arbitrary: Q*)  
  **case** (*Awhile I b C*)  
  **show** *?case*  
  **proof** (*simp, rule While'*)  
  **from**  $\langle vc\ (Awhile\ I\ b\ C)\ Q \rangle$   
  **have** *vc*:  $vc\ C\ I$  **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**  $\vdash \{pre\ C\ I\}\ strip\ C\ \{I\}$  **by** (*rule Awhile.IH[OF vc]*)  
  **with** *pre* **show**  $\vdash \{\lambda s. I\ s \wedge\ bval\ b\ s\}\ strip\ C\ \{I\}$   
  **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'*:

$\llbracket vc\ C\ Q; \forall s. P\ s \longrightarrow pre\ C\ Q\ s \rrbracket \implies \vdash \{P\}\ strip\ C\ \{Q\}$   
**by** (*metis strengthen\_pre vc\_sound*)

### 13.4.4 Completeness

**lemma** *pre\_mono*:

$\forall s. P\ s \longrightarrow P'\ s \implies pre\ C\ P\ s \implies pre\ C\ P'\ s$   
**proof** (*induction C arbitrary: P P' s*)  
  **case** *Aseq* **thus** *?case* **by** *simp metis*  
**qed** *simp\_all*

**lemma** *vc\_mono*:

$\forall s. P\ s \longrightarrow P'\ s \implies vc\ C\ P \implies vc\ C\ P'$   
**proof** (*induction C arbitrary: P P'*)  
  **case** *Aseq* **thus** *?case* **by** *simp (metis pre\_mono)*  
**qed** *simp\_all*

**lemma** *vc\_complete*:

$\vdash \{P\}c\{Q\} \implies \exists C. strip\ C = c \wedge vc\ C\ Q \wedge (\forall s. P\ s \longrightarrow pre\ C\ Q\ s)$   
  (*is \_  $\implies \exists C. ?G\ P\ c\ Q\ C$* )  
**proof** (*induction rule: hoare.induct*)  
  **case** *Skip*  
  **show** *?case* (*is  $\exists C. ?C\ C$* )  
  **proof show** *?C Askip* **by** *simp qed*  
**next**  
  **case** (*Assign P a x*)

```

  show ?case (is  $\exists C. ?C C$ )
  proof show ?C(Aassign x a) by simp qed
next
  case (Seq P c1 Q c2 R)
  from Seq.IH obtain C1 where ih1: ?G P c1 Q C1 by blast
  from Seq.IH obtain C2 where ih2: ?G Q c2 R C2 by blast
  show ?case (is  $\exists C. ?C C$ )
  proof
    show ?C(Aseq C1 C2)
    using ih1 ih2 by (fastforce elim!: pre_mono vc_mono)
  qed
next
  case (If P b c1 Q c2)
  from If.IH obtain C1 where ih1: ?G ( $\lambda s. P s \wedge bval b s$ ) c1 Q C1
  by blast
  from If.IH obtain C2 where ih2: ?G ( $\lambda s. P s \wedge \neg bval b s$ ) c2 Q C2
  by blast
  show ?case (is  $\exists C. ?C C$ )
  proof
    show ?C(Aif b C1 C2) using ih1 ih2 by simp
  qed
next
  case (While P b c)
  from While.IH obtain C where ih: ?G ( $\lambda s. P s \wedge bval b s$ ) c P C by
blast
  show ?case (is  $\exists C. ?C C$ )
  proof show ?C(Awhile P b C) using ih by simp qed
next
  case conseq thus ?case by(fast elim!: pre_mono vc_mono)
qed

end

```

## 13.5 Hoare Logic for Total Correctness

### 13.5.1 Separate Termination Relation

```

theory Hoare_Total
imports Hoare_Examples
begin

```

Note that this definition of total validity  $\models_t$  only works if execution is deterministic (which it is in our case).

```

definition hoare_tvalid :: assn  $\Rightarrow$  com  $\Rightarrow$  assn  $\Rightarrow$  bool
( $\models_t \{(-)\} / (-) / \{(-)\}$  50) where

```

$$\models_t \{P\}c\{Q\} \longleftrightarrow (\forall s. P s \longrightarrow (\exists t. (c, s) \Rightarrow t \wedge Q t))$$

Provability 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**

$$hoaret :: assn \Rightarrow com \Rightarrow assn \Rightarrow bool \ (\vdash_t \ (\{(1\_)\} / \_ / \{(1\_)\}) \ 50)$$

**where**

$$Skip: \vdash_t \{P\} SKIP \{P\} \quad |$$

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

$$Seq: \llbracket \vdash_t \{P_1\} c_1 \{P_2\}; \vdash_t \{P_2\} c_2 \{P_3\} \rrbracket \Longrightarrow \vdash_t \{P_1\} c_1;;c_2 \{P_3\} \quad |$$

$$If: \llbracket \vdash_t \{\lambda s. P s \wedge bval b s\} c_1 \{Q\}; \vdash_t \{\lambda s. P s \wedge \neg bval b s\} c_2 \{Q\} \rrbracket \\ \Longrightarrow \vdash_t \{P\} IF b THEN c_1 ELSE c_2 \{Q\} \quad |$$

*While*:

$$(\wedge n::nat.$$

$$\vdash_t \{\lambda s. P s \wedge bval b s \wedge T s n\} c \{\lambda s. P s \wedge (\exists n' < n. T s n')\}$$

$$\Longrightarrow \vdash_t \{\lambda s. P s \wedge (\exists n. T s n)\} WHILE b DO c \{\lambda s. P s \wedge \neg bval b s\} \quad |$$

$$conseq: \llbracket \forall s. P' s \longrightarrow P s; \vdash_t \{P\}c\{Q\}; \forall s. Q s \longrightarrow Q' s \rrbracket \Longrightarrow \\ \vdash_t \{P'\}c\{Q'\}$$

The *While*-rule is like the one for partial correctness but it requires additionally that with every execution of the loop body some measure relation  $T :: state \Rightarrow nat \Rightarrow bool$  decreases. The following functional version is more intuitive:

**lemma** *While\_fun*:

$$\llbracket \wedge n::nat. \vdash_t \{\lambda s. P s \wedge bval b s \wedge n = f s\} c \{\lambda s. P s \wedge f s < n\} \rrbracket$$

$$\Longrightarrow \vdash_t \{P\} WHILE b DO c \{\lambda s. P s \wedge \neg bval b s\}$$

**by** (*rule While* [**where**  $T = \lambda s n. n = f s$ , *simplified*])

Building in the consequence rule:

**lemma** *strengthen\_pre*:

$$\llbracket \forall s. P' s \longrightarrow P s; \vdash_t \{P\} c \{Q\} \rrbracket \Longrightarrow \vdash_t \{P'\} c \{Q\}$$

**by** (*metis conseq*)

**lemma** *weaken\_post*:

$$\llbracket \vdash_t \{P\} c \{Q\}; \forall s. Q s \longrightarrow Q' s \rrbracket \Longrightarrow \vdash_t \{P\} c \{Q'\}$$

**by** (*metis conseq*)

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

**lemma** *While\_fun'*:

**assumes**  $\bigwedge n::nat. \vdash_t \{\lambda s. P\ s \wedge bval\ b\ s \wedge n = f\ s\}\ c\ \{\lambda s. P\ s \wedge f\ s < n\}$   
**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\_fun assms(2)]*)

Our standard example:

**lemma**  $\vdash_t \{\lambda s. s\ 'x'' = i\}\ 'y'' ::= N\ 0;;\ wsum\ \{\lambda s. s\ 'y'' = sum\ i\}$   
**apply**(*rule Seq*)  
**prefer** 2  
**apply**(*rule While\_fun'* [**where**  $P = \lambda s. (s\ 'y'' = sum\ i - sum(s\ 'x''))$   
**and**  $f = \lambda s. nat(s\ 'x'')$ ])  
**apply**(*rule Seq*)  
**prefer** 2  
**apply**(*rule Assign*)  
**apply**(*rule Assign'*)  
**apply** *simp*  
**apply**(*simp*)  
**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, induction rule: hoaret.induct*)  
**case** (*While P b T c*)  
**have**  $\llbracket P\ s; T\ s\ n \rrbracket \implies \exists t. (WHILE\ b\ DO\ c, s) \Rightarrow t \wedge P\ t \wedge \neg bval\ b\ t$   
**for**  $s\ n$   
**proof**(*induction n arbitrary: s rule: less\_induct*)  
**case** (*less n*) **thus** ?*case* **by** (*metis While.IH WhileFalse WhileTrue*)  
**qed**  
**thus** ?*case* **by** *auto*  
**next**  
**case** *If* **thus** ?*case* **by** *auto blast*  
**qed** *fastforce+*

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**  $wpt :: com \Rightarrow assn \Rightarrow assn\ (wp_t)$  **where**



$wp_t c Q = (\lambda s. \exists t. (c, s) \Rightarrow t \wedge Q t)$

**lemma** [simp]:  $wp_t \text{ SKIP } Q = Q$   
**by**(auto intro!: ext simp: wpt\_def)

**lemma** [simp]:  $wp_t (x ::= e) Q = (\lambda s. Q(s(x := \text{aval } e \ s)))$   
**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]:  
 $wp_t (\text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2) Q = (\lambda s. wp_t (\text{if } \text{bval } b \ s \ \text{then } c_1 \ \text{else } c_2) Q \ s)$   
**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* ::  $\text{bexp} \Rightarrow \text{com} \Rightarrow \text{state} \Rightarrow \text{nat} \Rightarrow \text{bool}$  **where**  
*Its\_0*:  $\neg \text{bval } b \ s \Longrightarrow \text{Its } b \ c \ s \ 0 \mid$   
*Its\_Suc*:  $\llbracket \text{bval } b \ s; (c, s) \Rightarrow s'; \text{Its } b \ c \ s' \ n \rrbracket \Longrightarrow \text{Its } b \ c \ s \ (\text{Suc } n)$

The relation is in fact a function:

**lemma** *Its\_fun*:  $\text{Its } b \ c \ s \ n \Longrightarrow \text{Its } b \ c \ s \ n' \Longrightarrow n=n'$   
**proof**(induction arbitrary:  $n'$  rule: *Its.induct*)  
**case** *Its\_0* **thus** ?case **by**(metis *Its.cases*)  
**next**  
**case** *Its\_Suc* **thus** ?case **by**(metis *Its.cases* big\_step\_determ)  
**qed**

For all terminating loops, *Its* yields a result:

**lemma** *WHILE\_Its*:  $(\text{WHILE } b \ \text{DO } c, s) \Rightarrow t \Longrightarrow \exists n. \text{Its } b \ c \ s \ n$   
**proof**(induction *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**

```

lemma wpt_is_pre:  $\vdash_t \{wp_t c Q\} c \{Q\}$ 
proof (induction c arbitrary: Q)
  case SKIP show ?case by (auto intro:hoaret.Skip)
next
  case Assign show ?case by (auto intro:hoaret.Assign)
next
  case Seq thus ?case by (auto intro:hoaret.Seq)
next
  case If thus ?case by (auto intro:hoaret.If hoaret.conseq)
next
  case (While b c)
  let ?w = WHILE b DO c
  let ?T = Its b c
  have 1:  $\forall s. wp_t ?w Q s \longrightarrow wp_t ?w Q s \wedge (\exists n. Its b c s n)$ 
    unfolding wpt_def by (metis WHILE_Its)
  let ?R =  $\lambda n s'. wp_t ?w Q s' \wedge (\exists n' < n. ?T s' n')$ 
  have  $\forall s. wp_t ?w Q s \wedge bval b s \wedge ?T s n \longrightarrow wp_t c (?R n) s$  for n
  proof -
    have  $wp_t c (?R n) s$  if  $bval b s$  and  $?T s n$  and  $(?w, s) \Rightarrow t$  and  $Q t$ 
    for s t
    proof -
      from  $\langle bval b s \rangle$  and  $\langle (?w, s) \Rightarrow t \rangle$  obtain s' where
         $\langle c, s \rangle \Rightarrow s' \langle ?w, s' \rangle \Rightarrow t$  by auto
      from  $\langle (?w, s') \Rightarrow t \rangle$  obtain n' where  $?T s' n'$ 
        by (blast dest: WHILE_Its)
      with  $\langle bval b s \rangle$  and  $\langle c, s \rangle \Rightarrow s'$  have  $?T s (Suc n')$  by (rule Its_Suc)
      with  $\langle ?T s n \rangle$  have  $n = Suc n'$  by (rule Its_fun)
      with  $\langle c, s \rangle \Rightarrow s'$  and  $\langle (?w, s') \Rightarrow t \rangle$  and  $\langle Q t \rangle$  and  $\langle ?T s' n' \rangle$ 
      show ?thesis by (auto simp: wpt_def)
    qed
  thus ?thesis
    unfolding wpt_def by auto

qed

note 2 = hoaret.While[OF strengthen_pre[OF this While.IH]]
have  $\forall s. wp_t ?w Q s \wedge \neg bval b s \longrightarrow Q s$ 
  by (auto simp add:wpt_def)
with 1 2 show ?case by (rule conseq)
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*:  $\models_t \{P\}c\{Q\} \implies \vdash_t \{P\}c\{Q\}$   
**apply**(*rule strengthen\_pre*[*OF \_ wpt.is\_pre*])  
**apply**(*auto simp: hoare\_tvalid\_def wpt\_def*)  
**done**

**corollary** *hoaret\_sound\_complete*:  $\vdash_t \{P\}c\{Q\} \longleftrightarrow \models_t \{P\}c\{Q\}$   
**by** (*metis hoaret\_sound hoaret\_complete*)

**end**

## 14 Abstract Interpretation

### 14.1 Complete Lattice

**theory** *Complete\_Lattice*  
**imports** *Main*  
**begin**

**locale** *Complete\_Lattice* =  
**fixes** *L* :: 'a::order set **and** *Glb* :: 'a set  $\Rightarrow$  'a  
**assumes** *Glb\_lower*:  $A \subseteq L \implies a \in A \implies \text{Glb } A \leq a$   
**and** *Glb\_greatest*:  $b \in L \implies \forall a \in A. b \leq a \implies b \leq \text{Glb } A$   
**and** *Glb\_in\_L*:  $A \subseteq L \implies \text{Glb } A \in L$   
**begin**

**definition** *lfp* :: ('a  $\Rightarrow$  'a)  $\Rightarrow$  'a **where**  
*lfp* *f* =  $\text{Glb } \{a : L. f a \leq a\}$

**lemma** *index\_lfp*: *lfp* *f*  $\in$  *L*  
**by**(*auto simp: lfp\_def intro: Glb\_in\_L*)

**lemma** *lfp\_lowerbound*:  
 $\llbracket a \in L; f a \leq a \rrbracket \implies \text{lfp } f \leq a$   
**by** (*auto simp add: lfp\_def intro: Glb\_lower*)

**lemma** *lfp\_greatest*:  
 $\llbracket a \in L; \bigwedge u. \llbracket u \in L; f u \leq u \rrbracket \implies a \leq u \rrbracket \implies a \leq \text{lfp } f$   
**by** (*auto simp add: lfp\_def intro: Glb\_greatest*)

**lemma** *lfp\_unfold*: **assumes**  $\bigwedge x. f x \in L \longleftrightarrow x \in L$   
**and** *mono*: *mono* *f* **shows** *lfp* *f* = *f* (*lfp* *f*)  
**proof**–

**note** *assms*(1)[*simp*] *index\_lfp*[*simp*]  
**have** 1:  $f (\text{lfp } f) \leq \text{lfp } f$

```

    apply(rule lfp_greatest)
    apply simp
    by (blast intro: lfp_lowerbound monoD[OF mono] order_trans)
  have lfp f ≤ f (lfp f)
    by (fastforce intro: 1 monoD[OF mono] lfp_lowerbound)
  with 1 show ?thesis by (blast intro: order_antisym)
qed

end

end

```

## 14.2 Annotated Commands

```

theory ACom
imports Com
begin

```

```

datatype 'a acom =
  SKIP 'a (SKIP {-} 61) |
  Assign vname aexp 'a ((- ::= -/ {-}) [1000, 61, 0] 61) |
  Seq ('a acom) ('a acom) (-;;/- [60, 61] 60) |
  If bexp 'a ('a acom) 'a ('a acom) 'a
  ((IF -/ THEN ({-}/ -)/ ELSE ({-}/ -)//{-}) [0, 0, 0, 61, 0, 0] 61) |
  While 'a bexp 'a ('a acom) 'a
  (({-}//WHILE -//DO ({-}//-)//{-}) [0, 0, 0, 61, 0] 61)

```

```

notation com.SKIP (SKIP)

```

```

fun strip :: 'a acom ⇒ com where
strip (SKIP {P}) = SKIP |
strip (x ::= e {P}) = x ::= e |
strip (C1;;C2) = strip C1;; strip C2 |
strip (IF b THEN {P1} C1 ELSE {P2} C2 {P}) =
  IF b THEN strip C1 ELSE strip C2 |
strip ({I} WHILE b DO {P} C {Q}) = WHILE b DO strip C

```

```

fun asize :: com ⇒ nat where

```

```

asize SKIP = 1 |
asize (x ::= e) = 1 |
asize (C1;;C2) = asize C1 + asize C2 |
asize (IF b THEN C1 ELSE C2) = asize C1 + asize C2 + 3 |
asize (WHILE b DO C) = asize C + 3

```

```

definition shift :: (nat ⇒ 'a) ⇒ nat ⇒ nat ⇒ 'a where

```

$shift\ f\ n = (\lambda p. f(p+n))$

**fun** *annotate* :: (*nat*  $\Rightarrow$  'a)  $\Rightarrow$  *com*  $\Rightarrow$  'a *acom* **where**  
*annotate* *f* *SKIP* = *SKIP* {*f* 0} |  
*annotate* *f* (*x* ::= *e*) = *x* ::= *e* {*f* 0} |  
*annotate* *f* (*c*<sub>1</sub>;;*c*<sub>2</sub>) = *annotate* *f* *c*<sub>1</sub>;; *annotate* (*shift* *f* (*asize* *c*<sub>1</sub>)) *c*<sub>2</sub> |  
*annotate* *f* (*IF* *b* *THEN* *c*<sub>1</sub> *ELSE* *c*<sub>2</sub>) =  
  *IF* *b* *THEN* {*f* 0} *annotate* (*shift* *f* 1) *c*<sub>1</sub>  
  *ELSE* {*f*(*asize* *c*<sub>1</sub> + 1)} *annotate* (*shift* *f* (*asize* *c*<sub>1</sub> + 2)) *c*<sub>2</sub>  
  {*f*(*asize* *c*<sub>1</sub> + *asize* *c*<sub>2</sub> + 2)} |  
*annotate* *f* (*WHILE* *b* *DO* *c*) =  
  {*f* 0} *WHILE* *b* *DO* {*f* 1} *annotate* (*shift* *f* 2) *c* {*f*(*asize* *c* + 2)}

**fun** *annos* :: 'a *acom*  $\Rightarrow$  'a *list* **where**  
*annos* (*SKIP* {*P*}) = [*P*] |  
*annos* (*x* ::= *e* {*P*}) = [*P*] |  
*annos* (*C*<sub>1</sub>;;*C*<sub>2</sub>) = *annos* *C*<sub>1</sub> @ *annos* *C*<sub>2</sub> |  
*annos* (*IF* *b* *THEN* {*P*<sub>1</sub>} *C*<sub>1</sub> *ELSE* {*P*<sub>2</sub>} *C*<sub>2</sub> {*Q*}) =  
  *P*<sub>1</sub> # *annos* *C*<sub>1</sub> @ *P*<sub>2</sub> # *annos* *C*<sub>2</sub> @ [*Q*] |  
*annos* ({*I*} *WHILE* *b* *DO* {*P*} *C* {*Q*}) = *I* # *P* # *annos* *C* @ [*Q*]

**definition** *anno* :: 'a *acom*  $\Rightarrow$  *nat*  $\Rightarrow$  'a **where**  
*anno* *C* *p* = *annos* *C* ! *p*

**definition** *post* :: 'a *acom*  $\Rightarrow$  'a **where**  
*post* *C* = *last*(*annos* *C*)  
**fun** *map\_acom* :: ('a  $\Rightarrow$  'b)  $\Rightarrow$  'a *acom*  $\Rightarrow$  'b *acom* **where**  
*map\_acom* *f* (*SKIP* {*P*}) = *SKIP* {*f* *P*} |  
*map\_acom* *f* (*x* ::= *e* {*P*}) = *x* ::= *e* {*f* *P*} |  
*map\_acom* *f* (*C*<sub>1</sub>;;*C*<sub>2</sub>) = *map\_acom* *f* *C*<sub>1</sub>;; *map\_acom* *f* *C*<sub>2</sub> |  
*map\_acom* *f* (*IF* *b* *THEN* {*P*<sub>1</sub>} *C*<sub>1</sub> *ELSE* {*P*<sub>2</sub>} *C*<sub>2</sub> {*Q*}) =  
  *IF* *b* *THEN* {*f* *P*<sub>1</sub>} *map\_acom* *f* *C*<sub>1</sub> *ELSE* {*f* *P*<sub>2</sub>} *map\_acom* *f* *C*<sub>2</sub>  
  {*f* *Q*} |  
*map\_acom* *f* ({*I*} *WHILE* *b* *DO* {*P*} *C* {*Q*}) =  
  {*f* *I*} *WHILE* *b* *DO* {*f* *P*} *map\_acom* *f* *C* {*f* *Q*}

**lemma** *annos\_ne*: *annos* *C*  $\neq$  []  
**by**(*induction* *C*) *auto*

**lemma** *strip\_annotate[simp]*: *strip*(*annotate* *f* *c*) = *c*  
**by**(*induction* *c* *arbitrary*: *f*) *auto*

**lemma** *length\_annos\_annotate[simp]*: *length* (*annos* (*annotate* *f* *c*)) = *asize* *c*  
**by**(*induction* *c* *arbitrary*: *f*) *auto*

**lemma** *size\_annos*:  $size(annos\ C) = asize(strip\ C)$   
**by** (*induction C*) (*auto*)

**lemma** *size\_annos\_same*:  $strip\ C1 = strip\ C2 \implies size(annos\ C1) = size(annos\ C2)$   
**apply** (*induct C2 arbitrary: C1*)  
**apply** (*case\_tac C1, simp\_all*)  
**done**

**lemmas** *size\_annos\_same2* = *eqTrueI[OF size\_annos\_same]*

**lemma** *anno\_annotate[simp]*:  $p < asize\ c \implies anno\ (annotate\ f\ c)\ p = f\ p$   
**apply** (*induction c arbitrary: f p*)  
**apply** (*auto simp: anno\_def nth\_append nth\_Cons numeral\_eq\_Suc shift\_def split: nat.split*)  
**apply** (*metis add\_Suc\_right add\_diff\_inverse add commute*)  
**apply** (*rule\_tac f=f in arg\_cong*)  
**apply** *arith*  
**apply** (*metis less\_Suc\_eq*)  
**done**

**lemma** *eq\_acom\_iff\_strip\_annos*:  
 $C1 = C2 \iff strip\ C1 = strip\ C2 \wedge annos\ C1 = annos\ C2$   
**apply** (*induction C1 arbitrary: C2*)  
**apply** (*case\_tac C2, auto simp: size\_annos\_same2*)  
**done**

**lemma** *eq\_acom\_iff\_strip\_anno*:  
 $C1 = C2 \iff strip\ C1 = strip\ C2 \wedge (\forall p < size(annos\ C1). anno\ C1\ p = anno\ C2\ p)$   
**by** (*auto simp add: eq\_acom\_iff\_strip\_annos anno\_def list\_eq\_iff\_nth\_eq size\_annos\_same2*)

**lemma** *post\_map\_acom[simp]*:  $post(map\_acom\ f\ C) = f(post\ C)$   
**by** (*induction C*) (*auto simp: post\_def last\_append annos\_ne*)

**lemma** *strip\_map\_acom[simp]*:  $strip\ (map\_acom\ f\ C) = strip\ C$   
**by** (*induction C*) *auto*

**lemma** *anno\_map\_acom*:  $p < size(annos\ C) \implies anno\ (map\_acom\ f\ C)\ p = f(anno\ C\ p)$   
**apply** (*induction C arbitrary: p*)  
**apply** (*auto simp: anno\_def nth\_append nth\_Cons' size\_annos*)

**done**

**lemma** *strip\_eq\_SKIP*:

$strip\ C = SKIP \longleftrightarrow (\exists P. C = SKIP\ \{P\})$

**by** (*cases C simp\_all*)

**lemma** *strip\_eq\_Assign*:

$strip\ C = x ::= e \longleftrightarrow (\exists P. C = x ::= e\ \{P\})$

**by** (*cases C simp\_all*)

**lemma** *strip\_eq\_Seq*:

$strip\ C = c1 ;; c2 \longleftrightarrow (\exists C1\ C2. C = C1 ;; C2 \ \&\ strip\ C1 = c1 \ \&\ strip\ C2 = c2)$

**by** (*cases C simp\_all*)

**lemma** *strip\_eq\_If*:

$strip\ C = IF\ b\ THEN\ c1\ ELSE\ c2 \longleftrightarrow$

$(\exists P1\ P2\ C1\ C2\ Q. C = IF\ b\ THEN\ \{P1\}\ C1\ ELSE\ \{P2\}\ C2\ \{Q\} \ \&\ strip\ C1 = c1 \ \&\ strip\ C2 = c2)$

**by** (*cases C simp\_all*)

**lemma** *strip\_eq\_While*:

$strip\ C = WHILE\ b\ DO\ c1 \longleftrightarrow$

$(\exists I\ P\ C1\ Q. C = \{I\}\ WHILE\ b\ DO\ \{P\}\ C1\ \{Q\} \ \&\ strip\ C1 = c1)$

**by** (*cases C simp\_all*)

**lemma** [*simp*]:  $shift\ (\lambda p. a)\ n = (\lambda p. a)$

**by**(*simp add: shift\_def*)

**lemma** *set\_annos\_anno*[*simp*]:  $set\ (annos\ (annotate\ (\lambda p. a)\ c)) = \{a\}$

**by**(*induction c simp\_all*)

**lemma** *post\_in\_annos*:  $post\ C \in set\ (annos\ C)$

**by**(*auto simp: post\_def annos\_ne*)

**lemma** *post\_anno\_asize*:  $post\ C = anno\ C\ (size\ (annos\ C) - 1)$

**by**(*simp add: post\_def last\_conv\_nth[OF annos\_ne] anno\_def*)

**end**

### 14.3 Collecting Semantics of Commands

**theory** *Collecting*

**imports** *Complete\_Lattice Big\_Step ACom*

**begin**

### 14.3.1 The generic Step function

**notation**

*sup* (**infixl**  $\sqcup$  65) **and**

*inf* (**infixl**  $\sqcap$  70) **and**

*bot* ( $\perp$ ) **and**

*top* ( $\top$ )

**context**

**fixes**  $f :: vname \Rightarrow aexp \Rightarrow 'a \Rightarrow 'a::sup$

**fixes**  $g :: bexp \Rightarrow 'a \Rightarrow 'a$

**begin**

**fun** *Step* ::  $'a \Rightarrow 'a\ acom \Rightarrow 'a\ acom$  **where**

*Step* *S* (*SKIP* {*Q*}) = (*SKIP* {*S*}) |

*Step* *S* ( $x ::= e$  {*Q*}) =

$x ::= e$  { $f\ x\ e\ S$ } |

*Step* *S* (*C1*;; *C2*) = *Step* *S* *C1*;; *Step* (*post* *C1*) *C2* |

*Step* *S* (*IF* *b* *THEN* {*P1*} *C1* *ELSE* {*P2*} *C2* {*Q*}) =

*IF* *b* *THEN* { $g\ b\ S$ } *Step* *P1* *C1* *ELSE* { $g\ (Not\ b)\ S$ } *Step* *P2* *C2*

{*post* *C1*  $\sqcup$  *post* *C2*} |

*Step* *S* ({*I*} *WHILE* *b* *DO* {*P*} *C* {*Q*}) =

{*S*  $\sqcup$  *post* *C*} *WHILE* *b* *DO* { $g\ b\ I$ } *Step* *P* *C* { $g\ (Not\ b)\ I$ }

**end**

**lemma** *strip\_Step[simp]*:  $strip(Step\ f\ g\ S\ C) = strip\ C$

**by**(*induct* *C* *arbitrary*: *S*) *auto*

### 14.3.2 Annotated commands as a complete lattice

**instantiation** *acom* :: (*order*) *order*

**begin**

**definition** *less\_eq\_acom* ::  $('a::order)acom \Rightarrow 'a\ acom \Rightarrow bool$  **where**

$C1 \leq C2 \iff strip\ C1 = strip\ C2 \wedge (\forall p < size(annos\ C1).\ anno\ C1\ p \leq anno\ C2\ p)$

**definition** *less\_acom* ::  $'a\ acom \Rightarrow 'a\ acom \Rightarrow bool$  **where**

$less\_acom\ x\ y = (x \leq y \wedge \neg y \leq x)$

**instance**

**proof** (*standard*, *goal\_cases*)

**case** 1 **show** ?*case* **by**(*simp* *add*: *less\_acom\_def*)



```

next
  case 2 thus ?case by(auto simp: less_eq_acom_def)
next
  case 3 thus ?case by(fastforce simp: less_eq_acom_def size_annos)
next
  case 4 thus ?case
    by(fastforce simp: le_antisym less_eq_acom_def size_annos
      eq_acom_iff_strip_anno)
qed

end

lemma less_eq_acom_annos:
   $C1 \leq C2 \iff \text{strip } C1 = \text{strip } C2 \wedge \text{list\_all2 } (\leq) (\text{annos } C1) (\text{annos } C2)$ 
by(auto simp add: less_eq_acom_def anno_def list_all2_conv_all_nth size_annos_same2)

lemma SKIP_le[simp]:  $SKIP \{S\} \leq c \iff (\exists S'. c = SKIP \{S'\} \wedge S \leq S')$ 
by (cases c) (auto simp: less_eq_acom_def anno_def)

lemma Assign_le[simp]:  $x ::= e \{S\} \leq c \iff (\exists S'. c = x ::= e \{S'\} \wedge S \leq S')$ 
by (cases c) (auto simp: less_eq_acom_def anno_def)

lemma Seq_le[simp]:  $C1 ;; C2 \leq C \iff (\exists C1' C2'. C = C1' ;; C2' \wedge C1 \leq C1' \wedge C2 \leq C2')$ 
apply (cases C)
apply(auto simp: less_eq_acom_annos list_all2_append size_annos_same2)
done

lemma If_le[simp]:  $IF \ b \ THEN \ \{p1\} \ C1 \ ELSE \ \{p2\} \ C2 \ \{S\} \leq C \iff$ 
 $(\exists p1' p2' C1' C2' S'. C = IF \ b \ THEN \ \{p1'\} \ C1' \ ELSE \ \{p2'\} \ C2' \ \{S'\} \wedge$ 
 $p1 \leq p1' \wedge p2 \leq p2' \wedge C1 \leq C1' \wedge C2 \leq C2' \wedge S \leq S')$ 
apply (cases C)
apply(auto simp: less_eq_acom_annos list_all2_append size_annos_same2)
done

lemma While_le[simp]:  $\{I\} \ WHILE \ b \ DO \ \{p\} \ C \ \{P\} \leq W \iff$ 
 $(\exists I' p' C' P'. W = \{I'\} \ WHILE \ b \ DO \ \{p'\} \ C' \ \{P'\} \wedge C \leq C' \wedge p \leq p' \wedge I \leq I' \wedge P \leq P')$ 
apply (cases W)
apply(auto simp: less_eq_acom_annos list_all2_append size_annos_same2)

```

**done**

**lemma** *mono\_post*:  $C \leq C' \implies \text{post } C \leq \text{post } C'$   
**using** *annos\_ne*[of *C'*]  
**by**(*auto simp: post\_def less\_eq\_acom\_def last\_conv\_nth*[OF *annos\_ne*] *anno\_def*  
    *dest: size\_annos\_same*)

**definition** *Inf\_acom* :: *com*  $\Rightarrow$  'a::*complete\_lattice* *acom set*  $\Rightarrow$  'a *acom*  
**where**  
*Inf\_acom* *c M* = *annotate* ( $\lambda p. \text{INF } C:M. \text{anno } C p$ ) *c*

**global interpretation**

*Complete\_Lattice* {*C. strip C = c*} *Inf\_acom c* **for** *c*  
**proof** (*standard, goal\_cases*)  
  **case** 1 **thus** ?*case*  
    **by**(*auto simp: Inf\_acom\_def less\_eq\_acom\_def size\_annos intro:INF\_lower*)  
**next**  
  **case** 2 **thus** ?*case*  
    **by**(*auto simp: Inf\_acom\_def less\_eq\_acom\_def size\_annos intro:INF\_greatest*)  
**next**  
  **case** 3 **thus** ?*case* **by**(*auto simp: Inf\_acom\_def*)  
**qed**

### 14.3.3 Collecting semantics

**definition** *step* = *Step* ( $\lambda x e S. \{s(x := \text{aval } e s) \mid s. s \in S\}$ ) ( $\lambda b S. \{s:S. \text{bval } b s\}$ )

**definition** *CS* :: *com*  $\Rightarrow$  *state set acom* **where**  
*CS* *c* = *lfp* *c* (*step UNIV*)

**lemma** *mono2\_Step*: **fixes** *C1 C2* :: 'a::*semilattice\_sup* *acom*  
  **assumes**  $\forall x e S1 S2. S1 \leq S2 \implies f x e S1 \leq f x e S2$   
     $\forall b S1 S2. S1 \leq S2 \implies g b S1 \leq g b S2$   
  **shows**  $C1 \leq C2 \implies S1 \leq S2 \implies \text{Step } f g S1 C1 \leq \text{Step } f g S2 C2$   
**proof**(*induction S1 C1 arbitrary: C2 S2 rule: Step.induct*)  
  **case** 1 **thus** ?*case* **by**(*auto*)  
**next**  
  **case** 2 **thus** ?*case* **by** (*auto simp: assms(1)*)  
**next**  
  **case** 3 **thus** ?*case* **by**(*auto simp: mono\_post*)  
**next**  
  **case** 4 **thus** ?*case*  
    **by**(*auto simp: subset\_iff assms(2)*)

(metis mono\_post le\_supI1 le\_supI2)+  
**next**  
 case 5 thus ?case  
 by(auto simp: subset\_iff assms(2))  
 (metis mono\_post le\_supI1 le\_supI2)+  
**qed**

**lemma mono2\_step:**  $C1 \leq C2 \implies S1 \subseteq S2 \implies \text{step } S1 \ C1 \leq \text{step } S2 \ C2$   
**unfolding** step\_def **by**(rule mono2\_Step) auto

**lemma mono\_step:** mono (step S)  
**by**(blast intro: monoI mono2\_step)

**lemma strip\_step:** strip(step S C) = strip C  
**by** (induction C arbitrary: S) (auto simp: step\_def)

**lemma lfp\_cs\_unfold:** lfp c (step S) = step S (lfp c (step S))  
**apply**(rule lfp\_unfold[OF \_ mono\_step])  
**apply**(simp add: strip\_step)  
**done**

**lemma CS\_unfold:** CS c = step UNIV (CS c)  
**by** (metis CS\_def lfp\_cs\_unfold)

**lemma strip\_CS[simp]:** strip(CS c) = c  
**by**(simp add: CS\_def index\_lfp[simplified])

#### 14.3.4 Relation to big-step semantics

**lemma asize\_nz:** asize(c::com)  $\neq 0$   
**by** (metis length\_0\_conv length\_annos\_annotate annos\_ne)

**lemma post\_Inf\_acom:**  
 $\forall C \in M. \text{strip } C = c \implies \text{post } (\text{Inf\_acom } c \ M) = \bigcap (\text{post } ' M)$   
**apply**(subgoal\_tac  $\forall C \in M. \text{size}(\text{annos } C) = \text{asize } c$ )  
**apply**(simp add: post\_anno\_asize Inf\_acom\_def asize\_nz neq0\_conv[symmetric])  
**apply**(simp add: size\_annos)  
**done**

**lemma post\_lfp:** post(lfp c f) =  $(\bigcap \{\text{post } C \mid C. \text{strip } C = c \wedge f \ C \leq C\})$   
**by**(auto simp add: lfp\_def post\_Inf\_acom)

**lemma big\_step\_post\_step:**  
 $\llbracket (c, s) \Rightarrow t; \text{strip } C = c; s \in S; \text{step } S \ C \leq C \rrbracket \implies t \in \text{post } C$

```

proof(induction arbitrary: C S rule: big_step_induct)
  case Skip thus ?case by(auto simp: strip_eq_SKIP step_def post_def)
next
  case Assign thus ?case
    by(fastforce simp: strip_eq_Assign step_def post_def)
next
  case Seq thus ?case
    by(fastforce simp: strip_eq_Seq step_def post_def last_append annos_ne)
next
  case IfTrue thus ?case apply(auto simp: strip_eq>If step_def post_def)
    by (metis (lifting,full_types) mem_Collect_eq set_mp)
next
  case IfFalse thus ?case apply(auto simp: strip_eq>If step_def post_def)
    by (metis (lifting,full_types) mem_Collect_eq set_mp)
next
  case (WhileTrue b s1 c' s2 s3)
    from WhileTrue.prems(1) obtain I P C' Q where  $C = \{I\}$  WHILE b
    DO  $\{P\}$   $C' \{Q\}$  strip  $C' = c'$ 
    by(auto simp: strip_eq_While)
    from WhileTrue.prems(3)  $\langle C = \_ \rangle$ 
    have step  $P C' \leq C' \{s \in I. \text{bval } b \ s\} \leq P \ S \leq I \ \text{step} \ (\text{post } C') \ C \leq$ 
     $C$ 
    by (auto simp: step_def post_def)
    have step  $\{s \in I. \text{bval } b \ s\} C' \leq C'$ 
    by (rule order_trans[OF mono2_step[OF order_refl \{s \in I. bval b s\} \leq
     $P]$   $\langle \text{step } P C' \leq C' \rangle$ ))
    have  $s1 \in \{s \in I. \text{bval } b \ s\}$  using  $\langle s1 \in S \rangle \langle S \subseteq I \rangle \langle \text{bval } b \ s1 \rangle$  by auto
    note  $s2.\text{in\_post\_}C' = \text{WhileTrue.IH}(1)[\text{OF } \langle \text{strip } C' = c' \rangle \text{ this } \langle \text{step } \{s$ 
     $\in I. \text{bval } b \ s\} C' \leq C' \rangle]$ 
    from WhileTrue.IH(2)[OF WhileTrue.prems(1)  $s2.\text{in\_post\_}C' \langle \text{step} \ (\text{post}$ 
     $C') \ C \leq C \rangle]$ 
    show ?case .
next
  case (WhileFalse b s1 c') thus ?case
    by (force simp: strip_eq_While step_def post_def)
qed

lemma big_step_lfp:  $\llbracket (c,s) \Rightarrow t; \ s \in S \rrbracket \Longrightarrow t \in \text{post}(\text{lfp } c \ (\text{step } S))$ 
by(auto simp add: post_lfp intro: big_step_post_step)

lemma big_step_CS:  $(c,s) \Rightarrow t \Longrightarrow t \in \text{post}(CS \ c)$ 
by(simp add: CS_def big_step_lfp)

end

```

## 14.4 Collecting Semantics Examples

```
theory Collecting_Examples
imports Collecting_Vars
begin
```

### 14.4.1 Pretty printing state sets

Tweak code generation to work with sets of non-equality types:

```
declare insert_code[code del] union_coset_filter[code del]
lemma insert_code [code]: insert x (set xs) = set (x#xs)
by simp
```

Compensate for the fact that sets may now have duplicates:

```
definition compact :: 'a set  $\Rightarrow$  'a set where
compact X = X
```

```
lemma [code]: compact(set xs) = set(remdups xs)
by(simp add: compact_def)
```

```
definition vars_acom = compact o vars o strip
```

In order to display commands annotated with state sets, states must be translated into a printable format as sets of variable-state pairs, for the variables in the command:

```
definition show_acom :: state set acom  $\Rightarrow$  (vname*val)set set acom where
show_acom C =
  annotate ( $\lambda p.$  ( $\lambda s.$  ( $\lambda x.$  (x, s x)) ' (vars_acom C)) ' anno C p) (strip C)
```

### 14.4.2 Examples

```
definition c0 = WHILE Less (V "x") (N 3)
  DO "x" ::= Plus (V "x") (N 2)
```

```
definition C0 :: state set acom where C0 = annotate ( $\lambda p.$  {p}) c0
```

Collecting semantics:

```
value show_acom (((step {<>}) ^^ 0) C0)
value show_acom (((step {<>}) ^^ 1) C0)
value show_acom (((step {<>}) ^^ 2) C0)
value show_acom (((step {<>}) ^^ 3) C0)
value show_acom (((step {<>}) ^^ 4) C0)
value show_acom (((step {<>}) ^^ 5) C0)
value show_acom (((step {<>}) ^^ 6) C0)
value show_acom (((step {<>}) ^^ 7) C0)
```

```
value show_acom (((step {<>}) ^^ 8) C0)
```

Small-step semantics:

```
value show_acom (((step {}) ^^ 0) (step {<>} C0))
value show_acom (((step {}) ^^ 1) (step {<>} C0))
value show_acom (((step {}) ^^ 2) (step {<>} C0))
value show_acom (((step {}) ^^ 3) (step {<>} C0))
value show_acom (((step {}) ^^ 4) (step {<>} C0))
value show_acom (((step {}) ^^ 5) (step {<>} C0))
value show_acom (((step {}) ^^ 6) (step {<>} C0))
value show_acom (((step {}) ^^ 7) (step {<>} C0))
value show_acom (((step {}) ^^ 8) (step {<>} C0))
```

end

## 14.5 Abstract Interpretation Test Programs

```
theory Abs_Int_Tests
```

```
imports Com
```

```
begin
```

For constant propagation:

Straight line code:

```
definition test1_const =
  "y" ::= N 7;;
  "z" ::= Plus (V "y") (N 2);;
  "y" ::= Plus (V "x") (N 0)
```

Conditional:

```
definition test2_const =
  IF Less (N 41) (V "x") THEN "x" ::= N 5 ELSE "x" ::= N 5
```

Conditional, test is relevant:

```
definition test3_const =
  "x" ::= N 42;;
  IF Less (N 41) (V "x") THEN "x" ::= N 5 ELSE "x" ::= N 6
```

While:

```
definition test4_const =
  "x" ::= N 0;; WHILE Bc True DO "x" ::= N 0
```

While, test is relevant:

```
definition test5_const =
  "x" ::= N 0;; WHILE Less (V "x") (N 1) DO "x" ::= N 1
```

Iteration is needed:

```
definition test6_const =  
  "x'' ::= N 0;; "y'' ::= N 0;; "z'' ::= N 2;;  
  WHILE Less (V "x'') (N 1) DO ("x'' ::= V "y'';; "y'' ::= V "z'')
```

For intervals:

```
definition test1_ivl =  
  "y'' ::= N 7;;  
  IF Less (V "x'') (V "y'')  
  THEN "y'' ::= Plus (V "y'') (V "x'')  
  ELSE "x'' ::= Plus (V "x'') (V "y'')
```

```
definition test2_ivl =  
  WHILE Less (V "x'') (N 100)  
  DO "x'' ::= Plus (V "x'') (N 1)
```

```
definition test3_ivl =  
  "x'' ::= N 0;;  
  WHILE Less (V "x'') (N 100)  
  DO "x'' ::= Plus (V "x'') (N 1)
```

```
definition test4_ivl =  
  "x'' ::= N 0;; "y'' ::= N 0;;  
  WHILE Less (V "x'') (N 11)  
  DO ("x'' ::= Plus (V "x'') (N 1);; "y'' ::= Plus (V "y'') (N 1))
```

```
definition test5_ivl =  
  "x'' ::= N 0;; "y'' ::= N 0;;  
  WHILE Less (V "x'') (N 100)  
  DO ("y'' ::= V "x'';; "x'' ::= Plus (V "x'') (N 1))
```

```
definition test6_ivl =  
  "x'' ::= N 0;;  
  WHILE Less (N (- 1)) (V "x'') DO "x'' ::= Plus (V "x'') (N 1)
```

**end**

**theory** Abs\_Int\_init

**imports** HOL-Library.While\_Combinator

HOL-Library.Extended

Vars Collecting Abs\_Int\_Tests

**begin**

**hide\_const** (**open**) top bot dom — to avoid qualified names

end

## 14.6 Abstract Interpretation

```
theory Abs_Int0
imports Abs_Int_init
begin
```

### 14.6.1 Orderings

The basic type classes *order*, *semilattice\_sup* and *order\_top* are defined in *Main*, more precisely in theories *HOL.Orderings* and *HOL.Lattices*. If you view this theory with *jedit*, just click on the names to get there.

```
class semilattice_sup_top = semilattice_sup + order_top
```

```
instance fun :: (type, semilattice_sup_top) semilattice_sup_top ..
```

```
instantiation option :: (order)order
begin
```

```
fun less_eq_option where
  Some x ≤ Some y = (x ≤ y) |
  None ≤ y = True |
  Some _ ≤ None = False
```

```
definition less_option where x < (y::'a option) = (x ≤ y ∧ ¬ y ≤ x)
```

```
lemma le_None[simp]: (x ≤ None) = (x = None)
by (cases x) simp_all
```

```
lemma Some_le[simp]: (Some x ≤ u) = (∃ y. u = Some y ∧ x ≤ y)
by (cases u) auto
```

```
instance
proof (standard, goal_cases)
  case 1 show ?case by(rule less_option_def)
next
  case (2 x) show ?case by(cases x, simp_all)
next
  case (3 x y z) thus ?case by(cases z, simp, cases y, simp, cases x, auto)
next
  case (4 x y) thus ?case by(cases y, simp, cases x, auto)
qed
```



**end**

**instantiation** *option* :: (*sup*)*sup*  
**begin**

**fun** *sup\_option* **where**  
*Some*  $x \sqcup$  *Some*  $y =$  *Some*( $x \sqcup y$ ) |  
*None*  $\sqcup y = y$  |  
 $x \sqcup$  *None* =  $x$

**lemma** *sup\_None2*[*simp*]:  $x \sqcup$  *None* =  $x$   
**by** (*cases*  $x$ ) *simp\_all*

**instance** ..

**end**

**instantiation** *option* :: (*semilattice\_sup\_top*)*semilattice\_sup\_top*  
**begin**

**definition** *top\_option* **where**  $\top =$  *Some*  $\top$

**instance**

**proof** (*standard*, *goal\_cases*)

**case** (*4*  $a$ ) **show** ?*case* **by**(*cases*  $a$ , *simp\_all* *add*: *top\_option\_def*)

**next**

**case** (*1*  $x$   $y$ ) **thus** ?*case* **by**(*cases*  $x$ , *simp*, *cases*  $y$ , *simp\_all*)

**next**

**case** (*2*  $x$   $y$ ) **thus** ?*case* **by**(*cases*  $y$ , *simp*, *cases*  $x$ , *simp\_all*)

**next**

**case** (*3*  $x$   $y$   $z$ ) **thus** ?*case* **by**(*cases*  $z$ , *simp*, *cases*  $y$ , *simp*, *cases*  $x$ ,  
*simp\_all*)

**qed**

**end**

**lemma** [*simp*]: (*Some*  $x <$  *Some*  $y$ ) = ( $x <$   $y$ )  
**by**(*auto* *simp*: *less\_le*)

**instantiation** *option* :: (*order*)*order\_bot*  
**begin**

**definition** *bot\_option* :: '*a* *option* **where**

$\perp = \text{None}$

**instance**

**proof** (*standard, goal\_cases*)

**case 1 thus ?case by**(*auto simp: bot\_option\_def*)

**qed**

**end**

**definition** *bot* :: *com*  $\Rightarrow$  'a *option* *acom* **where**

*bot* *c* = *annotate* ( $\lambda p. \text{None}$ ) *c*

**lemma** *bot\_least*: *strip* *C* = *c*  $\implies$  *bot* *c*  $\leq$  *C*

**by**(*auto simp: bot\_def less\_eq\_acom\_def*)

**lemma** *strip\_bot*[*simp*]: *strip*(*bot* *c*) = *c*

**by**(*simp add: bot\_def*)

#### 14.6.2 Pre-fixpoint iteration

**definition** *pf* :: ('a::order)  $\Rightarrow$  'a  $\Rightarrow$  'a *option* **where**

*pf* *f* = *while\_option* ( $\lambda x. \neg f\ x \leq x$ ) *f*

**lemma** *pf\_pfp*: **assumes** *pf* *f* *x0* = *Some* *x* **shows** *f* *x*  $\leq$  *x*

**using** *while\_option\_stop*[*OF* *assms*[*simplified pf\_def*]] **by** *simp*

**lemma** *while\_least*:

**fixes** *q* :: 'a::order

**assumes**  $\forall x \in L. \forall y \in L. x \leq y \longrightarrow f\ x \leq f\ y$  **and**  $\forall x. x \in L \longrightarrow f\ x \in L$

**and**  $\forall x \in L. b \leq x$  **and**  $b \in L$  **and**  $f\ q \leq q$  **and**  $q \in L$

**and** *while\_option* *P* *f* *b* = *Some* *p*

**shows**  $p \leq q$

**using** *while\_option\_rule*[*OF* - *assms*( $\gamma$ )[*unfolded pf\_def*],

**where**  $P = \%x. x \in L \wedge x \leq q$ ]

**by** (*metis* *assms*(1-6) *order\_trans*)

**lemma** *pf\_bot\_least*:

**assumes**  $\forall x \in \{C. \text{strip } C = c\}. \forall y \in \{C. \text{strip } C = c\}. x \leq y \longrightarrow f\ x \leq f\ y$

**and**  $\forall C. C \in \{C. \text{strip } C = c\} \longrightarrow f\ C \in \{C. \text{strip } C = c\}$

**and**  $f\ C' \leq C' \text{strip } C' = c$  *pf* *f* (*bot* *c*) = *Some* *C*

**shows**  $C \leq C'$

**by**(*rule* *while\_least*[*OF* *assms*(1,2) - *assms*(3) - *assms*(5)[*unfolded pf\_def*]])

  (*simp\_all* *add: assms*(4) *bot\_least*)

**lemma** *pf<sub>p</sub>-inv*:

*pf<sub>p</sub> f x = Some y*  $\implies (\bigwedge x. P x \implies P(f x)) \implies P x \implies P y$   
**unfolding** *pf<sub>p</sub>-def* **by** (*blast intro: while\_option\_rule*)

**lemma** *strip-pfp*:

**assumes**  $\bigwedge x. g(f x) = g x$  **and** *pf<sub>p</sub> f x0 = Some x* **shows**  $g x = g x0$   
**using** *pf<sub>p</sub>-inv[OF assms(2)*, **where**  $P = \%x. g x = g x0$  *assms(1)* **by**  
*simp*

### 14.6.3 Abstract Interpretation

**definition**  $\gamma\_fun :: ('a \Rightarrow 'b\ set) \Rightarrow ('c \Rightarrow 'a) \Rightarrow ('c \Rightarrow 'b)\ set$  **where**  
 $\gamma\_fun\ \gamma\ F = \{f. \forall x. f\ x \in \gamma(F\ x)\}$

**fun**  $\gamma\_option :: ('a \Rightarrow 'b\ set) \Rightarrow 'a\ option \Rightarrow 'b\ set$  **where**  
 $\gamma\_option\ \gamma\ None = \{\}$  |  
 $\gamma\_option\ \gamma\ (Some\ a) = \gamma\ a$

The interface for abstract values:

**locale** *Val\_semilattice* =  
**fixes**  $\gamma :: 'av::semilattice\_sup\_top \Rightarrow val\ set$   
**assumes** *mono\_gamma*:  $a \leq b \implies \gamma\ a \leq \gamma\ b$   
**and** *gamma\_Top[simp]*:  $\gamma\ \top = UNIV$   
**fixes** *num'* ::  $val \Rightarrow 'av$   
**and** *plus'* ::  $'av \Rightarrow 'av \Rightarrow 'av$   
**assumes** *gamma\_num'*:  $i \in \gamma(num'\ i)$   
**and** *gamma\_plus'*:  $i1 \in \gamma\ a1 \implies i2 \in \gamma\ a2 \implies i1+i2 \in \gamma(plus'\ a1\ a2)$   
**type\_synonym**  $'av\ st = (vname \Rightarrow 'av)$

The for-clause (here and elsewhere) only serves the purpose of fixing the name of the type parameter *'av* which would otherwise be renamed to *'a*.

**locale** *Abs\_Int\_fun* = *Val\_semilattice* **where**  $\gamma=\gamma$   
**for**  $\gamma :: 'av::semilattice\_sup\_top \Rightarrow val\ set$   
**begin**

**fun** *aval'* ::  $aexp \Rightarrow 'av\ st \Rightarrow 'av$  **where**  
*aval'* (*N i*)  $S = num'\ i$  |  
*aval'* (*V x*)  $S = S\ x$  |  
*aval'* (*Plus a1 a2*)  $S = plus'\ (aval'\ a1\ S)\ (aval'\ a2\ S)$

**definition** *asem*  $x\ e\ S = (case\ S\ of\ None \Rightarrow None\ |\ Some\ S \Rightarrow Some(S(x\ :=\ aval'\ e\ S)))$

**definition**  $step' = Step\ asem\ (\lambda b\ S.\ S)$

**lemma**  $strip\_step'[simp]: strip(step'\ S\ C) = strip\ C$   
**by** ( $simp\ add: step\_def$ )

**definition**  $AI :: com \Rightarrow 'av\ st\ option\ acom\ option$  **where**  
 $AI\ c = pfp\ (step'\ \top)\ (bot\ c)$

**abbreviation**  $\gamma_s :: 'av\ st \Rightarrow state\ set$   
**where**  $\gamma_s == \gamma\_fun\ \gamma$

**abbreviation**  $\gamma_o :: 'av\ st\ option \Rightarrow state\ set$   
**where**  $\gamma_o == \gamma\_option\ \gamma_s$

**abbreviation**  $\gamma_c :: 'av\ st\ option\ acom \Rightarrow state\ set\ acom$   
**where**  $\gamma_c == map\_acom\ \gamma_o$

**lemma**  $gamma\_s\_Top[simp]: \gamma_s\ \top = UNIV$   
**by** ( $simp\ add: top\_fun\_def\ \gamma\_fun\_def$ )

**lemma**  $gamma\_o\_Top[simp]: \gamma_o\ \top = UNIV$   
**by** ( $simp\ add: top\_option\_def$ )

**lemma**  $mono\_gamma\_s: f1 \leq f2 \implies \gamma_s\ f1 \subseteq \gamma_s\ f2$   
**by** ( $auto\ simp: le\_fun\_def\ \gamma\_fun\_def\ dest: mono\_gamma$ )

**lemma**  $mono\_gamma\_o:$   
 $S1 \leq S2 \implies \gamma_o\ S1 \subseteq \gamma_o\ S2$   
**by** ( $induction\ S1\ S2\ rule: less\_eq\_option.induct$ ) ( $simp\_all\ add: mono\_gamma\_s$ )

**lemma**  $mono\_gamma\_c: C1 \leq C2 \implies \gamma_c\ C1 \leq \gamma_c\ C2$   
**by** ( $simp\ add: less\_eq\_acom.def\ mono\_gamma\_o\ size\_annos\ anno\_map\_acom$   
 $size\_annos\_same[of\ C1\ C2]$ )

Correctness:

**lemma**  $aval\_correct: s \in \gamma_s\ S \implies aval\ a\ s \in \gamma(aval'\ a\ S)$   
**by** ( $induct\ a$ ) ( $auto\ simp: gamma\_num'\ gamma\_plus'\ \gamma\_fun\_def$ )

**lemma**  $in\_gamma\_update: [ s \in \gamma_s\ S; i \in \gamma\ a ] \implies s(x := i) \in \gamma_s(S(x := a))$   
**by** ( $simp\ add: \gamma\_fun\_def$ )

**lemma** *gamma\_Step\_subcomm*:  
**assumes**  $\bigwedge x \in S. f1\ x \in (\gamma_o\ S) \subseteq \gamma_o\ (f2\ x \in S) \ \bigwedge b \in S. g1\ b \in (\gamma_o\ S) \subseteq \gamma_o\ (g2\ b \in S)$   
**shows**  $Step\ f1\ g1\ (\gamma_o\ S)\ (\gamma_c\ C) \leq \gamma_c\ (Step\ f2\ g2\ S\ C)$   
**by** (*induction C arbitrary: S*) (*auto simp: mono\_gamma\_o assms*)

**lemma** *step\_step'*:  $step\ (\gamma_o\ S)\ (\gamma_c\ C) \leq \gamma_c\ (step'\ S\ C)$   
**unfolding** *step\_def step'\_def*  
**by**(*rule gamma\_Step\_subcomm*)  
(*auto simp: aval'\_correct in\_gamma\_update asem\_def split: option.splits*)

**lemma** *AI\_correct*:  $AI\ c = Some\ C \implies CS\ c \leq \gamma_c\ C$

**proof**(*simp add: CS\_def AI\_def*)  
**assume** *1*:  $fpf\ (step'\ \top)\ (bot\ c) = Some\ C$   
**have** *pfp'*:  $step'\ \top\ C \leq C$  **by**(*rule pfp\_pfp[OF 1]*)  
**have** *2*:  $step\ (\gamma_o\ \top)\ (\gamma_c\ C) \leq \gamma_c\ C$  — transfer the pfp'  
**proof**(*rule order\_trans*)  
**show**  $step\ (\gamma_o\ \top)\ (\gamma_c\ C) \leq \gamma_c\ (step'\ \top\ C)$  **by**(*rule step\_step'*)  
**show**  $\dots \leq \gamma_c\ C$  **by** (*metis mono\_gamma\_c[OF pfp']*)  
**qed**  
**have** *3*:  $strip\ (\gamma_c\ C) = c$  **by**(*simp add: strip\_pfp[OF - 1] step'\_def*)  
**have** *lfp*  $c\ (step\ (\gamma_o\ \top)) \leq \gamma_c\ C$   
**by**(*rule lfp\_lowerbound[simplified,where f=step (\gamma\_o \top), OF 3 2]*)  
**thus**  $lfp\ c\ (step\ UNIV) \leq \gamma_c\ C$  **by** *simp*  
**qed**

**end**

#### 14.6.4 Monotonicity

**locale** *Abs\_Int\_fun\_mono* = *Abs\_Int\_fun* +  
**assumes** *mono\_plus'*:  $a1 \leq b1 \implies a2 \leq b2 \implies plus'\ a1\ a2 \leq plus'\ b1\ b2$   
**begin**

**lemma** *mono\_aval'*:  $S \leq S' \implies aval'\ e\ S \leq aval'\ e\ S'$   
**by**(*induction e*)(*auto simp: le\_fun\_def mono\_plus'*)

**lemma** *mono\_update*:  $a \leq a' \implies S \leq S' \implies S(x := a) \leq S'(x := a')$   
**by**(*simp add: le\_fun\_def*)

**lemma** *mono\_step'*:  $S1 \leq S2 \implies C1 \leq C2 \implies step'\ S1\ C1 \leq step'\ S2\ C2$   
**unfolding** *step'\_def*  
**by**(*rule mono2\_Step*)

(*auto simp: mono\_update mono\_aval' asem\_def split: option.split*)

**lemma** *mono\_step'\_top*:  $C \leq C' \implies \text{step}' \top C \leq \text{step}' \top C'$   
**by** (*metis mono\_step' order\_refl*)

**lemma** *AI\_least\_pfp*: **assumes**  $AI\ c = \text{Some } C\ \text{step}' \top C' \leq C'\ \text{strip } C' = c$   
**shows**  $C \leq C'$   
**by**(*rule pfp\_bot\_least[OF \_ - assms(2,3) assms(1)[unfolded AI\_def]]*)  
(*simp\_all add: mono\_step'\_top*)

**end**

**instantiation** *acom* :: (*type*) *vars*  
**begin**

**definition** *vars\_acom* = *vars o strip*

**instance** ..

**end**

**lemma** *finite\_Cvars*: *finite(vars(C::'a acom))*  
**by**(*simp add: vars\_acom\_def*)

### 14.6.5 Termination

**lemma** *pfp\_termination*:  
**fixes**  $x0 :: 'a::order$  **and**  $m :: 'a \Rightarrow nat$   
**assumes** *mono*:  $\bigwedge x\ y. I\ x \implies I\ y \implies x \leq y \implies f\ x \leq f\ y$   
**and** *m*:  $\bigwedge x\ y. I\ x \implies I\ y \implies x < y \implies m\ x > m\ y$   
**and** *I*:  $\bigwedge x\ y. I\ x \implies I(f\ x)$  **and**  $I\ x0$  **and**  $x0 \leq f\ x0$   
**shows**  $\exists x. \text{pfp } f\ x0 = \text{Some } x$   
**proof**(*simp add: pfp\_def, rule wf\_while\_option\_Some*[**where**  $P = \%x. I\ x \ \& \ x \leq f\ x$ ])  
  **show**  $wf\ \{(y,x). ((I\ x \wedge x \leq f\ x) \wedge \neg f\ x \leq x) \wedge y = f\ x\}$   
  **by**(*rule wf\_subset[OF wf\_measure[of m]]*) (*auto simp: m I*)  
**next**  
  **show**  $I\ x0 \wedge x0 \leq f\ x0$  **using**  $\langle I\ x0 \rangle \langle x0 \leq f\ x0 \rangle$  **by** *blast*  
**next**  
  **fix**  $x$  **assume**  $I\ x \wedge x \leq f\ x$  **thus**  $I(f\ x) \wedge f\ x \leq f(f\ x)$   
  **by** (*blast intro: I mono*)  
**qed**

**lemma** *le\_iff\_le\_annos*:  $C1 \leq C2 \iff$   
 $strip\ C1 = strip\ C2 \wedge (\forall\ i < size(annos\ C1). annos\ C1\ !\ i \leq annos\ C2\ !$   
 $i)$   
**by**(*simp add: less\_eq\_acom\_def anno\_def*)

**locale** *Measure1\_fun* =  
**fixes**  $m :: 'av::top \Rightarrow nat$   
**fixes**  $h :: nat$   
**assumes**  $h: m\ x \leq h$   
**begin**

**definition**  $m_s :: 'av\ st \Rightarrow vname\ set \Rightarrow nat\ (m_s)$  **where**  
 $m_s\ S\ X = (\sum\ x \in X. m(S\ x))$

**lemma** *m\_s\_h*:  $finite\ X \implies m_s\ S\ X \leq h * card\ X$   
**by**(*simp add: m\_s\_def*) (*metis mult.commute of\_nat\_id sum\_bounded\_above[OF*  
 $h]$ )

**fun**  $m_o :: 'av\ st\ option \Rightarrow vname\ set \Rightarrow nat\ (m_o)$  **where**  
 $m_o\ (Some\ S)\ X = m_s\ S\ X\ |$   
 $m_o\ None\ X = h * card\ X + 1$

**lemma** *m\_o\_h*:  $finite\ X \implies m_o\ opt\ X \leq (h * card\ X + 1)$   
**by**(*cases opt*)(*auto simp add: m\_s\_h le\_SucI dest: m\_s\_h*)

**definition**  $m_c :: 'av\ st\ option\ acom \Rightarrow nat\ (m_c)$  **where**  
 $m_c\ C = sum\_list\ (map\ (\lambda a. m_o\ a\ (vars\ C))\ (annos\ C))$

Upper complexity bound:

**lemma** *m\_c\_h*:  $m_c\ C \leq size(annos\ C) * (h * card(vars\ C) + 1)$

**proof**–

**let**  $?X = vars\ C$  **let**  $?n = card\ ?X$  **let**  $?a = size(annos\ C)$   
**have**  $m_c\ C = (\sum\ i < ?a. m_o\ (annos\ C\ !\ i)\ ?X)$   
**by**(*simp add: m\_c\_def sum\_list\_sum\_nth atLeast0LessThan*)  
**also have**  $\dots \leq (\sum\ i < ?a. h * ?n + 1)$   
**apply**(*rule sum\_mono*) **using** *m\_o\_h[OF finite\_Cvars]* **by** *simp*  
**also have**  $\dots = ?a * (h * ?n + 1)$  **by** *simp*  
**finally show** *?thesis* .

**qed**

**end**

**locale** *Measure\_fun* = *Measure1\_fun* **where**  $m=m$   
**for**  $m :: 'av::\text{semilattice\_sup\_top} \Rightarrow \text{nat} +$   
**assumes**  $m2: x < y \Longrightarrow m\ x > m\ y$   
**begin**

The predicates *top\_on\_ty*  $a\ X$  that follow describe that any abstract state in  $a$  maps all variables in  $X$  to  $\top$ . This is an important invariant for the termination proof where we argue that only the finitely many variables in the program change. That the others do not change follows because they remain  $\top$ .

**fun** *top\_on\_st* ::  $'av\ st \Rightarrow \text{vname}\ \text{set} \Rightarrow \text{bool}\ (top\_on\_s)$  **where**  
*top\_on\_st*  $S\ X = (\forall x \in X. S\ x = \top)$

**fun** *top\_on\_opt* ::  $'av\ st\ \text{option} \Rightarrow \text{vname}\ \text{set} \Rightarrow \text{bool}\ (top\_on\_o)$  **where**  
*top\_on\_opt*  $(Some\ S)\ X = top\_on\_st\ S\ X \mid$   
*top\_on\_opt*  $None\ X = True$

**definition** *top\_on\_acom* ::  $'av\ st\ \text{option}\ \text{acom} \Rightarrow \text{vname}\ \text{set} \Rightarrow \text{bool}\ (top\_on\_c)$   
**where**  
*top\_on\_acom*  $C\ X = (\forall a \in \text{set}(\text{annos}\ C). top\_on\_opt\ a\ X)$

**lemma** *top\_on\_top*:  $top\_on\_opt\ \top\ X$   
**by**(*auto simp: top\_option\_def*)

**lemma** *top\_on\_bot*:  $top\_on\_acom\ (\text{bot}\ c)\ X$   
**by**(*auto simp add: top\_on\_acom\_def bot\_def*)

**lemma** *top\_on\_post*:  $top\_on\_acom\ C\ X \Longrightarrow top\_on\_opt\ (\text{post}\ C)\ X$   
**by**(*simp add: top\_on\_acom\_def post\_in\_annos*)

**lemma** *top\_on\_acom\_simps*:  
 $top\_on\_acom\ (\text{SKIP}\ \{Q\})\ X = top\_on\_opt\ Q\ X$   
 $top\_on\_acom\ (x ::= e\ \{Q\})\ X = top\_on\_opt\ Q\ X$   
 $top\_on\_acom\ (C1;;C2)\ X = (top\_on\_acom\ C1\ X \wedge top\_on\_acom\ C2\ X)$   
 $top\_on\_acom\ (\text{IF}\ b\ \text{THEN}\ \{P1\}\ C1\ \text{ELSE}\ \{P2\}\ C2\ \{Q\})\ X =$   
 $(top\_on\_opt\ P1\ X \wedge top\_on\_acom\ C1\ X \wedge top\_on\_opt\ P2\ X \wedge top\_on\_acom\ C2\ X \wedge top\_on\_opt\ Q\ X)$   
 $top\_on\_acom\ (\{I\}\ \text{WHILE}\ b\ \text{DO}\ \{P\}\ C\ \{Q\})\ X =$   
 $(top\_on\_opt\ I\ X \wedge top\_on\_acom\ C\ X \wedge top\_on\_opt\ P\ X \wedge top\_on\_opt\ Q\ X)$   
**by**(*auto simp add: top\_on\_acom\_def*)

**lemma** *top\_on\_sup*:  
 $top\_on\_opt\ o1\ X \Longrightarrow top\_on\_opt\ o2\ X \Longrightarrow top\_on\_opt\ (o1\ \sqcup\ o2)\ X$



**apply**(*induction o1 o2 rule: sup\_option.induct*)  
**apply**(*auto*)  
**done**

**lemma** *top\_on\_Step: fixes C :: 'av st option acom*  
**assumes**  $!!x \in S. \llbracket \text{top\_on\_opt } S \ X; x \notin X; \text{vars } e \subseteq -X \rrbracket \implies \text{top\_on\_opt } (f \ x \ e \ S) \ X$   
 $!!b \ S. \text{top\_on\_opt } S \ X \implies \text{vars } b \subseteq -X \implies \text{top\_on\_opt } (g \ b \ S) \ X$   
**shows**  $\llbracket \text{vars } C \subseteq -X; \text{top\_on\_opt } S \ X; \text{top\_on\_acom } C \ X \rrbracket \implies \text{top\_on\_acom } (\text{Step } f \ g \ S \ C) \ X$   
**proof**(*induction C arbitrary: S*)  
**qed** (*auto simp: top\_on\_acom\_simps vars\_acom\_def top\_on\_post top\_on\_sup assms*)

**lemma** *m1:  $x \leq y \implies m \ x \geq m \ y$*   
**by**(*auto simp: le\_less m2*)

**lemma** *m\_s2\_rep: assumes finite(X) and S1 = S2 on -X and  $\forall x. S1 \ x \leq S2 \ x$  and  $S1 \neq S2$*   
**shows**  $(\sum x \in X. m \ (S2 \ x)) < (\sum x \in X. m \ (S1 \ x))$   
**proof**–  
**from** *assms(3) have 1:  $\forall x \in X. m(S1 \ x) \geq m(S2 \ x)$  by (simp add: m1)*  
**from** *assms(2,3,4) have  $\exists x \in X. S1 \ x < S2 \ x$*   
**by**(*simp add: fun\_eq\_iff*) (*metis Compl\_iff le\_neq\_trans*)  
**hence** *2:  $\exists x \in X. m(S1 \ x) > m(S2 \ x)$  by (metis m2)*  
**from** *sum\_strict\_mono\_ex1[OF (finite X) 1 2]*  
**show**  $(\sum x \in X. m \ (S2 \ x)) < (\sum x \in X. m \ (S1 \ x)) .$   
**qed**

**lemma** *m\_s2: finite(X)  $\implies S1 = S2$  on  $-X \implies S1 < S2 \implies m\_s \ S1 \ X > m\_s \ S2 \ X$*   
**apply**(*auto simp add: less\_fun\_def m\_s\_def*)  
**apply**(*simp add: m\_s2\_rep le\_fun\_def*)  
**done**

**lemma** *m\_o2: finite X  $\implies \text{top\_on\_opt } o1 \ (-X) \implies \text{top\_on\_opt } o2 \ (-X) \implies$*   
 $o1 < o2 \implies m\_o \ o1 \ X > m\_o \ o2 \ X$   
**proof**(*induction o1 o2 rule: less\_eq\_option.induct*)  
**case 1 thus** *?case by (auto simp: m\_s2 less\_option\_def)*  
**next**  
**case 2 thus** *?case by (auto simp: less\_option\_def le\_imp\_less\_Suc m\_s\_h)*  
**next**  
**case 3 thus** *?case by (auto simp: less\_option\_def)*

qed

**lemma** *m\_o1*: *finite X*  $\implies$  *top\_on\_opt o1 (-X)*  $\implies$  *top\_on\_opt o2 (-X)*  
 $\implies$   
*o1*  $\leq$  *o2*  $\implies$  *m\_o o1 X*  $\geq$  *m\_o o2 X*  
**by**(*auto simp: le\_less m\_o2*)

**lemma** *m\_c2*: *top\_on\_acom C1 (-vars C1)*  $\implies$  *top\_on\_acom C2 (-vars C2)*  $\implies$

*C1*  $<$  *C2*  $\implies$  *m\_c C1*  $>$  *m\_c C2*

**proof**(*auto simp add: le\_iff\_le\_annos size\_annos\_same[of C1 C2] vars\_acom\_def less\_acom\_def*)

**let** *?X* = *vars(strip C2)*

**assume** *top*: *top\_on\_acom C1 (- vars(strip C2)) top\_on\_acom C2 (- vars(strip C2))*

**and** *strip\_eq*: *strip C1 = strip C2*

**and** *0*:  $\forall i < \text{size}(\text{annos } C2). \text{annos } C1 ! i \leq \text{annos } C2 ! i$

**hence** *1*:  $\forall i < \text{size}(\text{annos } C2). m_o (\text{annos } C1 ! i) ?X \geq m_o (\text{annos } C2 ! i) ?X$

**apply** (*auto simp: all\_set\_conv\_all\_nth vars\_acom\_def top\_on\_acom\_def*)

**by** (*metis (lifting, no\_types) finite\_cvars m\_o1 size\_annos\_same2*)

**fix** *i* **assume** *i*:  $i < \text{size}(\text{annos } C2) \neg \text{annos } C2 ! i \leq \text{annos } C1 ! i$

**have** *topo1*: *top\_on\_opt (annos C1 ! i) (- ?X)*

**using** *i(1) topo1* **by**(*simp add: top\_on\_acom\_def size\_annos\_same[OF strip\_eq]*)

**have** *topo2*: *top\_on\_opt (annos C2 ! i) (- ?X)*

**using** *i(1) topo2* **by**(*simp add: top\_on\_acom\_def size\_annos\_same[OF strip\_eq]*)

**from** *i* **have**  $m_o (\text{annos } C1 ! i) ?X > m_o (\text{annos } C2 ! i) ?X$  (**is** *?P i*)

**by** (*metis 0 less\_option\_def m\_o2[OF finite\_cvars topo1] topo2*)

**hence** *2*:  $\exists i < \text{size}(\text{annos } C2). ?P i$  **using**  $\langle i < \text{size}(\text{annos } C2) \rangle$  **by** *blast*

**have**  $(\sum i < \text{size}(\text{annos } C2). m_o (\text{annos } C2 ! i) ?X)$

$< (\sum i < \text{size}(\text{annos } C2). m_o (\text{annos } C1 ! i) ?X)$

**apply**(*rule sum\_strict\_mono\_ex1*) **using** *1 2* **by** (*auto*)

**thus** *?thesis*

**by**(*simp add: m\_c\_def vars\_acom\_def strip\_eq sum\_list\_sum\_nth atLeast0LessThan size\_annos\_same[OF strip\_eq]*)

qed

end

**locale** *Abs\_Int\_fun\_measure* =

```

  Abs_Int_fun_mono where  $\gamma = \gamma + \text{Measure\_fun}$  where  $m = m$ 
  for  $\gamma :: 'av :: \text{semilattice\_sup\_top} \Rightarrow \text{val set}$  and  $m :: 'av \Rightarrow \text{nat}$ 
begin

lemma top_on_step':  $\text{top\_on\_acom } C \ (-\text{vars } C) \Longrightarrow \text{top\_on\_acom } (\text{step}' \top C) \ (-\text{vars } C)$ 
unfolding step'_def
by(rule top_on_Step)
  (auto simp add: top_option_def asem_def split: option.splits)

lemma AI.Some_measure:  $\exists C. \text{AI } c = \text{Some } C$ 
unfolding AI_def
apply(rule pfp_termination[where  $I = \lambda C. \text{top\_on\_acom } C \ (-\text{vars } C)$ 
and  $m = m\_c$ ])
apply(simp_all add: m_c2 mono_step'_top bot_least top_on_bot)
using top_on_step' apply(auto simp add: vars_acom_def)
done

end

```

Problem: not executable because of the comparison of abstract states, i.e. functions, in the pre-fixpoint computation.

**end**

## 14.7 Computable State

```

theory Abs_State
imports Abs_Int0
begin

```

```

type_synonym 'a st_rep = (vname * 'a) list

```

```

fun fun_rep :: ('a :: top) st_rep  $\Rightarrow$  vname  $\Rightarrow$  'a where
fun_rep [] = ( $\lambda x. \top$ ) |
fun_rep ((x, a)#ps) = (fun_rep ps) ( $x := a$ )

```

```

lemma fun_rep_map_of[code]: — original def is too slow
  fun_rep ps = ( $\%x. \text{case map\_of } ps \ x \ \text{of } \text{None} \Rightarrow \top \mid \text{Some } a \Rightarrow a$ )
by(induction ps rule: fun_rep.induct) auto

```

```

definition eq_st :: ('a :: top) st_rep  $\Rightarrow$  'a st_rep  $\Rightarrow$  bool where
eq_st S1 S2 = (fun_rep S1 = fun_rep S2)

```

```

hide_type st — hide previous def to avoid long names

```

**declare**  $[[\text{typedef\_overloaded}]]$  — allow quotient types to depend on classes

**quotient\_type**  $'a\ st = ('a::\text{top})\ st\_rep / eq\_st$   
**morphisms**  $rep\_st\ St$   
**by**  $(metis\ eq\_st\_def\ equivpI\ reflpI\ sympI\ transpI)$

**lift\_definition**  $update :: ('a::\text{top})\ st \Rightarrow vname \Rightarrow 'a \Rightarrow 'a\ st$   
**is**  $\lambda ps\ x\ a.\ (x,a)\#ps$   
**by** $(auto\ simp:\ eq\_st\_def)$

**lift\_definition**  $fun :: ('a::\text{top})\ st \Rightarrow vname \Rightarrow 'a\ \text{is}\ fun\_rep$   
**by** $(simp\ add:\ eq\_st\_def)$

**definition**  $show\_st :: vname\ set \Rightarrow ('a::\text{top})\ st \Rightarrow (vname * 'a)\ set$  **where**  
 $show\_st\ X\ S = (\lambda x.\ (x,\ fun\ S\ x))\ 'X$

**definition**  $show\_acom\ C = map\_acom\ (map\_option\ (show\_st\ (vars(strip\ C))))\ C$

**definition**  $show\_acom\_opt = map\_option\ show\_acom$

**lemma**  $fun\_update[simp]:\ fun\ (update\ S\ x\ y) = (fun\ S)(x:=y)$   
**by**  $transfer\ auto$

**definition**  $\gamma\_st :: (('a::\text{top}) \Rightarrow 'b\ set) \Rightarrow 'a\ st \Rightarrow (vname \Rightarrow 'b)\ set$  **where**  
 $\gamma\_st\ \gamma\ F = \{f.\ \forall x.\ f\ x \in \gamma(fun\ F\ x)\}$

**instantiation**  $st :: (order\_top)\ order$   
**begin**

**definition**  $less\_eq\_st\_rep :: 'a\ st\_rep \Rightarrow 'a\ st\_rep \Rightarrow bool$  **where**  
 $less\_eq\_st\_rep\ ps1\ ps2 =$   
 $((\forall x \in set(map\ fst\ ps1) \cup set(map\ fst\ ps2).\ fun\_rep\ ps1\ x \leq fun\_rep\ ps2\ x))$

**lemma**  $less\_eq\_st\_rep\_iff:$

$less\_eq\_st\_rep\ r1\ r2 = (\forall x.\ fun\_rep\ r1\ x \leq fun\_rep\ r2\ x)$

**apply** $(auto\ simp:\ less\_eq\_st\_rep\_def\ fun\_rep\_map\_of\ split:\ option.\ split)$

**apply**  $(metis\ Un\_iff\ map\_of\_eq\_None\_iff\ option.\ distinct(1))$

**apply**  $(metis\ Un\_iff\ map\_of\_eq\_None\_iff\ option.\ distinct(1))$

**done**

**corollary**  $less\_eq\_st\_rep\_iff\_fun:$

$less\_eq\_st\_rep\ r1\ r2 = (fun\_rep\ r1 \leq fun\_rep\ r2)$

**by**  $(metis\ less\_eq\_st\_rep\_iff\ le\_fun\_def)$

**lift\_definition** *less\_eq\_st* :: 'a st  $\Rightarrow$  'a st  $\Rightarrow$  bool **is** *less\_eq\_st\_rep*  
**by**(*auto simp add: eq\_st\_def less\_eq\_st\_rep\_iff*)

**definition** *less\_st* **where**  $F < (G::'a\ st) = (F \leq G \wedge \neg G \leq F)$

**instance**

**proof** (*standard, goal\_cases*)

**case 1 show** ?*case* **by**(*rule less\_st\_def*)

**next**

**case 2 show** ?*case* **by** *transfer (auto simp: less\_eq\_st\_rep\_def)*

**next**

**case 3 thus** ?*case* **by** *transfer (metis less\_eq\_st\_rep\_iff order\_trans)*

**next**

**case 4 thus** ?*case*

**by** *transfer (metis less\_eq\_st\_rep\_iff eq\_st\_def fun\_eq\_iff antisym)*

**qed**

**end**

**lemma** *le\_st\_iff*:  $(F \leq G) = (\forall x. \text{fun } F\ x \leq \text{fun } G\ x)$

**by** *transfer (rule less\_eq\_st\_rep\_iff)*

**fun** *map2\_st\_rep* :: ('a::top  $\Rightarrow$  'a  $\Rightarrow$  'a)  $\Rightarrow$  'a st\_rep  $\Rightarrow$  'a st\_rep  $\Rightarrow$  'a st\_rep

**where**

*map2\_st\_rep* f [] *ps2* = *map* (%(x,y). (x, f  $\top$  y)) *ps2* |

*map2\_st\_rep* f ((x,y)#*ps1*) *ps2* =

  (*let* *y2* = *fun\_rep ps2 x*

*in* (x,f y *y2*) # *map2\_st\_rep* f *ps1 ps2*)

**lemma** *fun\_rep\_map2\_rep[simp]*:  $f \top \top = \top \Longrightarrow$

*fun\_rep (map2\_st\_rep f ps1 ps2) = ( $\lambda x. f (\text{fun\_rep } ps1\ x) (\text{fun\_rep } ps2\ x)$ )*

**apply**(*induction f ps1 ps2 rule: map2\_st\_rep.induct*)

**apply**(*simp add: fun\_rep\_map\_of map\_of\_map fun\_eq\_iff split: option.split*)

**apply**(*fastforce simp: fun\_rep\_map\_of fun\_eq\_iff split:option.splits*)

**done**

**instantiation** *st* :: (*semilattice\_sup\_top*) *semilattice\_sup\_top*

**begin**

**lift\_definition** *sup\_st* :: 'a st  $\Rightarrow$  'a st  $\Rightarrow$  'a st **is** *map2\_st\_rep* ( $\sqcup$ )

**by** (*simp add: eq\_st\_def*)

**lift\_definition** *top\_st* :: 'a st **is** [] .

```

instance
proof (standard, goal_cases)
  case 1 show ?case by transfer (simp add:less_eq_st_rep_iff)
next
  case 2 show ?case by transfer (simp add:less_eq_st_rep_iff)
next
  case 3 thus ?case by transfer (simp add:less_eq_st_rep_iff)
next
  case 4 show ?case by transfer (simp add:less_eq_st_rep_iff fun_rep_map_of)
qed

end

lemma fun_top: fun  $\top$  = ( $\lambda x. \top$ )
by transfer simp

lemma mono_update[simp]:
 $a1 \leq a2 \implies S1 \leq S2 \implies \text{update } S1 \ x \ a1 \leq \text{update } S2 \ x \ a2$ 
by transfer (auto simp add: less_eq_st_rep_def)

lemma mono_fun:  $S1 \leq S2 \implies \text{fun } S1 \ x \leq \text{fun } S2 \ x$ 
by transfer (simp add: less_eq_st_rep_iff)

locale Gamma_semilattice = Val_semilattice where  $\gamma = \gamma$ 
  for  $\gamma :: 'av :: \text{semilattice\_sup\_top} \Rightarrow \text{val set}$ 
begin

abbreviation  $\gamma_s :: 'av \ \text{st} \Rightarrow \text{state set}$ 
where  $\gamma_s == \gamma\_st \ \gamma$ 

abbreviation  $\gamma_o :: 'av \ \text{st option} \Rightarrow \text{state set}$ 
where  $\gamma_o == \gamma\_option \ \gamma_s$ 

abbreviation  $\gamma_c :: 'av \ \text{st option acom} \Rightarrow \text{state set acom}$ 
where  $\gamma_c == \text{map\_acom} \ \gamma_o$ 

lemma gamma_s_top[simp]:  $\gamma_s \top = UNIV$ 
by(auto simp:  $\gamma\_st\_def$  fun_top)

lemma gamma_o_Top[simp]:  $\gamma_o \top = UNIV$ 
by (simp add: top_option_def)

lemma mono_gamma_s:  $f \leq g \implies \gamma_s \ f \subseteq \gamma_s \ g$ 

```

**by**(*simp add:  $\gamma$ \_st\_def le\_st\_iff subset\_iff*) (*metis mono\_gamma\_subsetD*)

**lemma** *mono\_gamma\_o*:

$S1 \leq S2 \implies \gamma_o S1 \subseteq \gamma_o S2$

**by**(*induction S1 S2 rule: less\_eq\_option.induct*)(*simp\_all add: mono\_gamma\_s*)

**lemma** *mono\_gamma\_c*:  $C1 \leq C2 \implies \gamma_c C1 \leq \gamma_c C2$

**by** (*simp add: less\_eq\_acom\_def mono\_gamma\_o size\_annos anno\_map\_acom size\_annos\_same*[of *C1 C2*])

**lemma** *in\_gamma\_option\_iff*:

$x \in \gamma\_option\ r\ u \iff (\exists u'. u = \text{Some } u' \wedge x \in r\ u')$

**by** (*cases u*) *auto*

**end**

**end**

## 14.8 Computable Abstract Interpretation

**theory** *Abs\_Int1*

**imports** *Abs\_State*

**begin**

Abstract interpretation over type *st* instead of functions.

**context** *Gamma\_semilattice*

**begin**

**fun** *aval'* :: *aexp*  $\Rightarrow$  '*av st*  $\Rightarrow$  '*av* **where**

*aval'* (*N i*) *S* = *num'* *i* |

*aval'* (*V x*) *S* = *fun* *S x* |

*aval'* (*Plus a1 a2*) *S* = *plus'* (*aval'* *a1 S*) (*aval'* *a2 S*)

**lemma** *aval'\_correct*:  $s \in \gamma_s S \implies \text{aval } a\ s \in \gamma(\text{aval}'\ a\ S)$

**by** (*induction a*) (*auto simp: gamma\_num' gamma\_plus'  $\gamma$ \_st\_def*)

**lemma** *gamma\_Step\_subcomm*: **fixes** *C1 C2* :: '*a*::*semilattice\_sup* *acom*

**assumes**  $\forall x\ e\ S. f1\ x\ e\ (\gamma_o S) \subseteq \gamma_o (f2\ x\ e\ S)$

$\forall b\ S. g1\ b\ (\gamma_o S) \subseteq \gamma_o (g2\ b\ S)$

**shows** *Step f1 g1* ( $\gamma_o S$ ) ( $\gamma_c C$ )  $\leq \gamma_c$  (*Step f2 g2 S C*)

**proof**(*induction C arbitrary: S*)

**qed** (*auto simp: assms intro!: mono\_gamma\_o sup\_ge1 sup\_ge2*)

**lemma** *in\_gamma\_update*:  $\llbracket s \in \gamma_s S; i \in \gamma a \rrbracket \implies s(x := i) \in \gamma_s(\text{update}$

$S x a$   
**by**(*simp add:  $\gamma$ \_st\_def*)

**end**

**locale** *Abs\_Int* = *Gamma\_semilattice* **where**  $\gamma = \gamma$   
**for**  $\gamma :: 'av :: semilattice\_sup\_top \Rightarrow val\ set$   
**begin**

**definition** *step'* = *Step*  
 $(\lambda x e S. case\ S\ of\ None \Rightarrow None \mid Some\ S \Rightarrow Some(update\ S\ x\ (aval'\ e\ S)))$   
 $(\lambda b S. S)$

**definition** *AI* ::  $com \Rightarrow 'av\ st\ option\ acom\ option$  **where**  
 $AI\ c = pfp\ (step'\ \top)\ (bot\ c)$

**lemma** *strip\_step'[simp]*:  $strip(step'\ S\ C) = strip\ C$   
**by**(*simp add: step'\_def*)

Correctness:

**lemma** *step\_step'*:  $step\ (\gamma_o\ S)\ (\gamma_c\ C) \leq \gamma_c\ (step'\ S\ C)$   
**unfolding** *step\_def step'\_def*  
**by**(*rule gamma\_Step\_subcomm*)  
 $(auto\ simp: intro!: aval\_correct\ in\_gamma\_update\ split: option.splits)$

**lemma** *AI\_correct*:  $AI\ c = Some\ C \Longrightarrow CS\ c \leq \gamma_c\ C$   
**proof**(*simp add: CS\_def AI\_def*)

**assume** *1*:  $pfp\ (step'\ \top)\ (bot\ c) = Some\ C$   
**have** *1*:  $pfp'\ step'\ \top\ C \leq C$  **by**(*rule pfp\_pfp[OF 1]*)  
**have** *2*:  $step\ (\gamma_o\ \top)\ (\gamma_c\ C) \leq \gamma_c\ C$  — transfer the pfp'  
**proof**(*rule order\_trans*)  
**show**  $step\ (\gamma_o\ \top)\ (\gamma_c\ C) \leq \gamma_c\ (step'\ \top\ C)$  **by**(*rule step\_step'*)  
**show**  $\dots \leq \gamma_c\ C$  **by** (*metis mono\_gamma\_c[OF pfp']*)  
**qed**  
**have** *3*:  $strip\ (\gamma_c\ C) = c$  **by**(*simp add: strip\_pfp[OF - 1] step'\_def*)  
**have** *lfp*  $c\ (step\ (\gamma_o\ \top)) \leq \gamma_c\ C$   
**by**(*rule lfp\_lowerbound[simplified, where f=step (\gamma\_o \top), OF 3 2]*)  
**thus**  $lfp\ c\ (step\ UNIV) \leq \gamma_c\ C$  **by** *simp*

**qed**

**end**



### 14.8.1 Monotonicity

**locale** *Abs\_Int\_mono* = *Abs\_Int* +  
**assumes** *mono\_plus'*:  $a1 \leq b1 \implies a2 \leq b2 \implies plus' a1 a2 \leq plus' b1 b2$   
**begin**

**lemma** *mono\_aval'*:  $S1 \leq S2 \implies aval' e S1 \leq aval' e S2$   
**by**(*induction e*) (*auto simp: mono\_plus' mono\_fun*)

**theorem** *mono\_step'*:  $S1 \leq S2 \implies C1 \leq C2 \implies step' S1 C1 \leq step' S2 C2$

**unfolding** *step'\_def*  
**by**(*rule mono2\_Step*) (*auto simp: mono\_aval' split: option.split*)

**lemma** *mono\_step'\_top*:  $C \leq C' \implies step' \top C \leq step' \top C'$   
**by** (*metis mono\_step' order\_refl*)

**lemma** *AI\_least\_pfp*: **assumes**  $AI\ c = Some\ C\ step' \top C' \leq C'$  *strip*  $C' = c$   
**shows**  $C \leq C'$   
**by**(*rule pfp\_bot\_least[OF \_ \_ assms(2,3) assms(1)[unfolded AI\_def]]*)  
(*simp\_all add: mono\_step'\_top*)

**end**

### 14.8.2 Termination

**locale** *Measure1* =  
**fixes**  $m :: 'av::order\_top \Rightarrow nat$   
**fixes**  $h :: nat$   
**assumes**  $h: m\ x \leq h$   
**begin**

**definition**  $m\_s :: 'av\ st \Rightarrow vname\ set \Rightarrow nat\ (m_s)$  **where**  
 $m\_s\ S\ X = (\sum\ x \in X. m(\text{fun } S\ x))$

**lemma**  $m\_s.h: finite\ X \implies m\_s\ S\ X \leq h * card\ X$   
**by**(*simp add: m\_s\_def*) (*metis mult.commute of\_nat\_id sum\_bounded\_above[OF h]*)

**definition**  $m\_o :: 'av\ st\ option \Rightarrow vname\ set \Rightarrow nat\ (m_o)$  **where**  
 $m\_o\ opt\ X = (case\ opt\ of\ None \Rightarrow h * card\ X + 1 \mid Some\ S \Rightarrow m\_s\ S\ X)$

**lemma**  $m\_o.h: finite\ X \implies m\_o\ opt\ X \leq (h * card\ X + 1)$

**by**(*auto simp add: m\_o\_def m\_s\_h le\_SucI split: option.split dest:m\_s\_h*)

**definition**  $m_c :: 'av\ st\ option\ acom \Rightarrow nat\ (m_c)$  **where**  
 $m_c\ C = sum\_list\ (map\ (\lambda a. m\_o\ a\ (vars\ C))\ (annos\ C))$

Upper complexity bound:

**lemma**  $m_c\_h: m_c\ C \leq size(annos\ C) * (h * card(vars\ C) + 1)$

**proof**–

**let**  $?X = vars\ C$  **let**  $?n = card\ ?X$  **let**  $?a = size(annos\ C)$   
**have**  $m_c\ C = (\sum\ i < ?a. m\_o\ (annos\ C\ !\ i)\ ?X)$   
**by**(*simp add: m\_c\_def sum\_list\_sum\_nth atLeast0LessThan*)  
**also have**  $\dots \leq (\sum\ i < ?a. h * ?n + 1)$   
**apply**(*rule sum\_mono*) **using**  $m\_o\_h[OF\ finite\_Cvars]$  **by** *simp*  
**also have**  $\dots = ?a * (h * ?n + 1)$  **by** *simp*  
**finally show**  $?thesis$  .

**qed**

**end**

**fun**  $top\_on\_st :: 'a::order\_top\ st \Rightarrow vname\ set \Rightarrow bool\ (top'\_on_s)$  **where**  
 $top\_on\_st\ S\ X = (\forall x \in X. fun\ S\ x = \top)$

**fun**  $top\_on\_opt :: 'a::order\_top\ st\ option \Rightarrow vname\ set \Rightarrow bool\ (top'\_on_o)$   
**where**  
 $top\_on\_opt\ (Some\ S)\ X = top\_on\_st\ S\ X$  |  
 $top\_on\_opt\ None\ X = True$

**definition**  $top\_on\_acom :: 'a::order\_top\ st\ option\ acom \Rightarrow vname\ set \Rightarrow bool$   
 $(top'\_on_c)$  **where**  
 $top\_on\_acom\ C\ X = (\forall a \in set(annos\ C). top\_on\_opt\ a\ X)$

**lemma**  $top\_on\_top: top\_on\_opt\ (\top::\_st\ option)\ X$   
**by**(*auto simp: top\_option\_def fun\_top*)

**lemma**  $top\_on\_bot: top\_on\_acom\ (bot\ c)\ X$   
**by**(*auto simp add: top\_on\_acom\_def bot\_def*)

**lemma**  $top\_on\_post: top\_on\_acom\ C\ X \Longrightarrow top\_on\_opt\ (post\ C)\ X$   
**by**(*simp add: top\_on\_acom\_def post\_in\_annos*)

**lemma**  $top\_on\_acom\_simps:$   
 $top\_on\_acom\ (SKIP\ \{Q\})\ X = top\_on\_opt\ Q\ X$   
 $top\_on\_acom\ (x ::= e\ \{Q\})\ X = top\_on\_opt\ Q\ X$   
 $top\_on\_acom\ (C1;;C2)\ X = (top\_on\_acom\ C1\ X \wedge top\_on\_acom\ C2\ X)$

$top\_on\_acom (IF\ b\ THEN\ \{P1\}\ C1\ ELSE\ \{P2\}\ C2\ \{Q\})\ X =$   
 $(top\_on\_opt\ P1\ X \wedge top\_on\_acom\ C1\ X \wedge top\_on\_opt\ P2\ X \wedge top\_on\_acom$   
 $C2\ X \wedge top\_on\_opt\ Q\ X)$   
 $top\_on\_acom (\{I\}\ WHILE\ b\ DO\ \{P\}\ C\ \{Q\})\ X =$   
 $(top\_on\_opt\ I\ X \wedge top\_on\_acom\ C\ X \wedge top\_on\_opt\ P\ X \wedge top\_on\_opt\ Q$   
 $X)$   
**by**(*auto simp add: top\_on\_acom\_def*)

**lemma** *top\_on\_sup*:

$top\_on\_opt\ o1\ X \implies top\_on\_opt\ o2\ X \implies top\_on\_opt\ (o1\ \sqcup\ o2\ ::\ \_)\ st$   
 $option)\ X$

**apply**(*induction o1 o2 rule: sup\_option.induct*)

**apply**(*auto*)

**by** *transfer simp*

**lemma** *top\_on\_Step*: **fixes**  $C :: ('a::semilattice\_sup\_top)st\ option\ acom$

**assumes**  $!!x\ e\ S. \llbracket top\_on\_opt\ S\ X; x \notin X; vars\ e \subseteq -X \rrbracket \implies top\_on\_opt$   
 $(f\ x\ e\ S)\ X$

$!!b\ S. top\_on\_opt\ S\ X \implies vars\ b \subseteq -X \implies top\_on\_opt\ (g\ b\ S)\ X$

**shows**  $\llbracket vars\ C \subseteq -X; top\_on\_opt\ S\ X; top\_on\_acom\ C\ X \rrbracket \implies top\_on\_acom$   
 $(Step\ f\ g\ S\ C)\ X$

**proof**(*induction C arbitrary: S*)

**qed** (*auto simp: top\_on\_acom\_simps vars\_acom\_def top\_on\_post top\_on\_sup*  
*assms*)

**locale** *Measure* = *Measure1* +

**assumes**  $m2: x < y \implies m\ x > m\ y$

**begin**

**lemma**  $m1: x \leq y \implies m\ x \geq m\ y$

**by**(*auto simp: le\_less m2*)

**lemma**  $m\_s2\_rep$ : **assumes**  $finite(X)$  **and**  $S1 = S2\ on\ -X$  **and**  $\forall x. S1\ x$   
 $\leq S2\ x$  **and**  $S1 \neq S2$

**shows**  $(\sum_{x \in X}. m\ (S2\ x)) < (\sum_{x \in X}. m\ (S1\ x))$

**proof**–

**from** *assms*(3) **have**  $1: \forall x \in X. m(S1\ x) \geq m(S2\ x)$  **by** (*simp add: m1*)

**from** *assms*(2,3,4) **have**  $\exists x \in X. S1\ x < S2\ x$

**by**(*simp add: fun\_eq\_iff*) (*metis Compl\_iff le\_neq\_trans*)

**hence**  $2: \exists x \in X. m(S1\ x) > m(S2\ x)$  **by** (*metis m2*)

**from** *sum\_strict\_mono\_ex1*[*OF* (*finite X*) 1 2]

**show**  $(\sum_{x \in X}. m\ (S2\ x)) < (\sum_{x \in X}. m\ (S1\ x))$  .

**qed**

```

lemma m_s2: finite(X)  $\implies$  fun S1 = fun S2 on  $\neg$ X
 $\implies$  S1 < S2  $\implies$  m_s S1 X > m_s S2 X
apply(auto simp add: less_st_def m_s_def)
apply (transfer fixing: m)
apply(simp add: less_eq_st_rep_iff eq_st_def m_s2_rep)
done

```

```

lemma m_o2: finite X  $\implies$  top_on_opt o1 ( $\neg$ X)  $\implies$  top_on_opt o2 ( $\neg$ X)
 $\implies$ 
  o1 < o2  $\implies$  m_o o1 X > m_o o2 X
proof(induction o1 o2 rule: less_eq_option.induct)
  case 1 thus ?case by (auto simp: m_o_def m_s2 less_option_def)
next
  case 2 thus ?case by(auto simp: m_o_def less_option_def le_imp_less_Suc
m_s.h)
next
  case 3 thus ?case by (auto simp: less_option_def)
qed

```

```

lemma m_o1: finite X  $\implies$  top_on_opt o1 ( $\neg$ X)  $\implies$  top_on_opt o2 ( $\neg$ X)
 $\implies$ 
  o1  $\leq$  o2  $\implies$  m_o o1 X  $\geq$  m_o o2 X
by(auto simp: le_less m_o2)

```

```

lemma m_c2: top_on_acom C1 ( $\neg$ vars C1)  $\implies$  top_on_acom C2 ( $\neg$ vars
C2)  $\implies$ 
  C1 < C2  $\implies$  m_c C1 > m_c C2
proof(auto simp add: le_iff_le_annos size_annos_same[of C1 C2] vars_acom_def
less_acom_def)
  let ?X = vars(strip C2)
  assume top: top_on_acom C1 ( $\neg$  vars(strip C2)) top_on_acom C2 ( $\neg$ 
vars(strip C2))
  and strip_eq: strip C1 = strip C2
  and 0:  $\forall i < \text{size}(\text{annos } C2). \text{annos } C1 ! i \leq \text{annos } C2 ! i$ 
  hence 1:  $\forall i < \text{size}(\text{annos } C2). m_o (\text{annos } C1 ! i) ?X \geq m_o (\text{annos } C2$ 
   $! i) ?X$ 
  apply (auto simp: all_set_conv_all_nth vars_acom_def top_on_acom_def)
  by (metis finite_cvars m_o1 size_annos_same2)
  fix i assume i:  $i < \text{size}(\text{annos } C2) \neg \text{annos } C2 ! i \leq \text{annos } C1 ! i$ 
  have topo1: top_on_opt (annos C1 ! i) ( $\neg$  ?X)
  using i(1) top(1) by(simp add: top_on_acom_def size_annos_same[OF
strip_eq])

```

```

have topo2: top_on_opt (annos C2 ! i) (- ?X)
  using i(1) top(2) by(simp add: top_on_acom_def size_annos_same[OF
strip_eq])
from i have m_o (annos C1 ! i) ?X > m_o (annos C2 ! i) ?X (is ?P i)
  by (metis 0 less_option_def m_o2[OF finite_cvars topo1] topo2)
hence 2:  $\exists i < \text{size}(\text{annos } C2). ?P i$  using (i < size(annos C2)) by blast
have ( $\sum i < \text{size}(\text{annos } C2). m_o (\text{annos } C2 ! i) ?X$ )
  < ( $\sum i < \text{size}(\text{annos } C2). m_o (\text{annos } C1 ! i) ?X$ )
  apply(rule sum_strict_mono_ex1) using 1 2 by (auto)
thus ?thesis
  by(simp add: m_c_def vars_acom_def strip_eq sum_list_sum_nth atLeast0LessThan
size_annos_same[OF strip_eq])
qed

```

**end**

```

locale Abs_Int_measure =
  Abs_Int_mono where  $\gamma = \gamma + \text{Measure}$  where  $m = m$ 
  for  $\gamma :: 'av :: \text{semilattice\_sup\_top} \Rightarrow \text{val set}$  and  $m :: 'av \Rightarrow \text{nat}$ 
begin

```

```

lemma top_on_step':  $\llbracket \text{top\_on\_acom } C (-\text{vars } C) \rrbracket \Longrightarrow \text{top\_on\_acom } (\text{step}'$ 
 $\top C) (-\text{vars } C)$ 
unfolding step'_def
by(rule top_on_Step)
  (auto simp add: top_option_def fun_top split: option.splits)

```

```

lemma AI_Some_measure:  $\exists C. AI\ c = \text{Some } C$ 
unfolding AI_def
apply(rule pfp_termination[where  $I = \lambda C. \text{top\_on\_acom } C (-\text{vars } C)$ 
and  $m = m_c$ ])
apply(simp_all add: m_c2 mono_step'_top bot_least top_on_bot)
using top_on_step' apply(auto simp add: vars_acom_def)
done

```

**end**

**end**

## 14.9 Constant Propagation

```

theory Abs_Int1_const
imports Abs_Int1

```

**begin**

**datatype** *const* = *Const val* | *Any*

**fun** *γ\_const* **where**

*γ\_const* (*Const i*) = {*i*} |

*γ\_const* (*Any*) = *UNIV*

**fun** *plus\_const* **where**

*plus\_const* (*Const i*) (*Const j*) = *Const(i+j)* |

*plus\_const* \_ \_ = *Any*

**lemma** *plus\_const\_cases*: *plus\_const a1 a2* =

(*case (a1,a2) of (Const i, Const j) ⇒ Const(i+j) | \_ ⇒ Any*)

**by**(*auto split: prod.split const.split*)

**instantiation** *const* :: *semilattice\_sup\_top*

**begin**

**fun** *less\_eq\_const* **where**  $x \leq y = (y = \text{Any} \mid x=y)$

**definition**  $x < (y::\text{const}) = (x \leq y \ \& \ \neg y \leq x)$

**fun** *sup\_const* **where**  $x \sqcup y = (\text{if } x=y \text{ then } x \text{ else } \text{Any})$

**definition**  $\top = \text{Any}$

**instance**

**proof** (*standard, goal\_cases*)

**case 1 thus ?case by** (*rule less\_const\_def*)

**next**

**case (2 x) show ?case by** (*cases x simp\_all*)

**next**

**case (3 x y z) thus ?case by**(*cases z, cases y, cases x, simp\_all*)

**next**

**case (4 x y) thus ?case by**(*cases x, cases y, simp\_all, cases y, simp\_all*)

**next**

**case (6 x y) thus ?case by**(*cases x, cases y, simp\_all*)

**next**

**case (5 x y) thus ?case by**(*cases y, cases x, simp\_all*)

**next**

**case (7 x y z) thus ?case by**(*cases z, cases y, cases x, simp\_all*)

**next**

**case 8 thus ?case by**(*simp add: top\_const\_def*)

qed

end

```
global_interpretation Val_semilattice
where  $\gamma = \gamma\_const$  and  $num' = Const$  and  $plus' = plus\_const$ 
proof (standard, goal_cases)
  case (1 a b) thus ?case
    by(cases a, cases b, simp, simp, cases b, simp, simp)
next
  case 2 show ?case by(simp add: top_const_def)
next
  case 3 show ?case by simp
next
  case 4 thus ?case by(auto simp: plus_const_cases split: const.split)
qed
```

```
global_interpretation Abs_Int
where  $\gamma = \gamma\_const$  and  $num' = Const$  and  $plus' = plus\_const$ 
defines  $AI\_const = AI$  and  $step\_const = step'$  and  $aval'\_const = aval'$ 
..
```

#### 14.9.1 Tests

**definition**  $steps\ c\ i = (step\_const \top \hat{\hat{}} i)$  (bot c)

```
value show_acom (steps test1_const 0)
value show_acom (steps test1_const 1)
value show_acom (steps test1_const 2)
value show_acom (steps test1_const 3)
value show_acom (the(AI_const test1_const))
```

```
value show_acom (the(AI_const test2_const))
value show_acom (the(AI_const test3_const))
```

```
value show_acom (steps test4_const 0)
value show_acom (steps test4_const 1)
value show_acom (steps test4_const 2)
value show_acom (steps test4_const 3)
value show_acom (steps test4_const 4)
value show_acom (the(AI_const test4_const))
```

```
value show_acom (steps test5_const 0)
```

```

value show_acom (steps test5_const 1)
value show_acom (steps test5_const 2)
value show_acom (steps test5_const 3)
value show_acom (steps test5_const 4)
value show_acom (steps test5_const 5)
value show_acom (steps test5_const 6)
value show_acom (the(AI_const test5_const))

```

```

value show_acom (steps test6_const 0)
value show_acom (steps test6_const 1)
value show_acom (steps test6_const 2)
value show_acom (steps test6_const 3)
value show_acom (steps test6_const 4)
value show_acom (steps test6_const 5)
value show_acom (steps test6_const 6)
value show_acom (steps test6_const 7)
value show_acom (steps test6_const 8)
value show_acom (steps test6_const 9)
value show_acom (steps test6_const 10)
value show_acom (steps test6_const 11)
value show_acom (steps test6_const 12)
value show_acom (steps test6_const 13)
value show_acom (the(AI_const test6_const))

```

Monotonicity:

```

global interpretation Abs_Int_mono
where  $\gamma = \gamma\_const$  and  $num' = Const$  and  $plus' = plus\_const$ 
proof (standard, goal_cases)
  case 1 thus ?case by(auto simp: plus_const_cases split: const.split)
qed

```

Termination:

```

definition m_const ::  $const \Rightarrow nat$  where
  m_const x = (if x = Any then 0 else 1)

global interpretation Abs_Int_measure
where  $\gamma = \gamma\_const$  and  $num' = Const$  and  $plus' = plus\_const$ 
and  $m = m\_const$  and  $h = 1$ 
proof (standard, goal_cases)
  case 1 thus ?case by(auto simp: m_const_def split: const.splits)
next
  case 2 thus ?case by(auto simp: m_const_def less_const_def split: const.splits)
qed

```



**thm** *AI\_Some\_measure*

**end**

## 14.10 Parity Analysis

**theory** *Abs\_Int1\_parity*

**imports** *Abs\_Int1*

**begin**

**datatype** *parity* = *Even* | *Odd* | *Either*

Instantiation of class *order* with type *parity*:

**instantiation** *parity* :: *order*

**begin**

First the definition of the interface function  $\leq$ . Note that the header of the definition must refer to the ascii name ( $\leq$ ) of the constants as *less\_eq\_parity* and the definition is named *less\_eq\_parity\_def*. Inside the definition the symbolic names can be used.

**definition** *less\_eq\_parity* **where**

$x \leq y = (y = \textit{Either} \vee x=y)$

We also need  $<$ , which is defined canonically:

**definition** *less\_parity* **where**

$x < y = (x \leq y \wedge \neg y \leq (x::\textit{parity}))$

(The type annotation is necessary to fix the type of the polymorphic predicates.)

Now the instance proof, i.e. the proof that the definition fulfills the axioms (assumptions) of the class. The initial proof-step generates the necessary proof obligations.

**instance**

**proof**

**fix**  $x::\textit{parity}$  **show**  $x \leq x$  **by**(*auto simp: less\_eq\_parity\_def*)

**next**

**fix**  $x y z :: \textit{parity}$  **assume**  $x \leq y$   $y \leq z$  **thus**  $x \leq z$   
**by**(*auto simp: less\_eq\_parity\_def*)

**next**

**fix**  $x y :: \textit{parity}$  **assume**  $x \leq y$   $y \leq x$  **thus**  $x = y$   
**by**(*auto simp: less\_eq\_parity\_def*)

**next**

**fix**  $x y :: \textit{parity}$  **show**  $(x < y) = (x \leq y \wedge \neg y \leq x)$  **by**(*rule less\_parity\_def*)

**qed**

**end**

Instantiation of class *semilattice\_sup\_top* with type *parity*:

**instantiation** *parity* :: *semilattice\_sup\_top*  
**begin**

**definition** *sup\_parity* **where**  
 $x \sqcup y = (\text{if } x = y \text{ then } x \text{ else } \textit{Either})$

**definition** *top\_parity* **where**  
 $\top = \textit{Either}$

Now the instance proof. This time we take a shortcut with the help of proof method *goal\_cases*: it creates cases 1 ... n for the subgoals 1 ... n; in case i, i is also the name of the assumptions of subgoal i and *case?* refers to the conclusion of subgoal i. The class axioms are presented in the same order as in the class definition.

**instance**  
**proof** (*standard*, *goal\_cases*)  
  **case** 1 **show** ?*case* **by**(*auto simp: less\_eq\_parity\_def sup\_parity\_def*)  
**next**  
  **case** 2 **show** ?*case* **by**(*auto simp: less\_eq\_parity\_def sup\_parity\_def*)  
**next**  
  **case** 3 **thus** ?*case* **by**(*auto simp: less\_eq\_parity\_def sup\_parity\_def*)  
**next**  
  **case** 4 **show** ?*case* **by**(*auto simp: less\_eq\_parity\_def top\_parity\_def*)  
**qed**  
**end**

Now we define the functions used for instantiating the abstract interpretation locales. Note that the Isabelle terminology is *interpretation*, not *instantiation* of locales, but we use instantiation to avoid confusion with abstract interpretation.

**fun**  $\gamma_{\text{parity}}$  :: *parity*  $\Rightarrow$  *val set* **where**  
 $\gamma_{\text{parity}} \textit{Even} = \{i. i \bmod 2 = 0\} \mid$   
 $\gamma_{\text{parity}} \textit{Odd} = \{i. i \bmod 2 = 1\} \mid$   
 $\gamma_{\text{parity}} \textit{Either} = \textit{UNIV}$

**fun** *num\_parity* :: *val*  $\Rightarrow$  *parity* **where**  
*num\_parity* *i* = (*if*  $i \bmod 2 = 0$  *then* *Even* *else* *Odd*)

**fun** *plus\_parity* :: *parity*  $\Rightarrow$  *parity*  $\Rightarrow$  *parity* **where**  
*plus\_parity* *Even* *Even* = *Even*  $\mid$

```

plus_parity Odd Odd = Even |
plus_parity Even Odd = Odd |
plus_parity Odd Even = Odd |
plus_parity Either y = Either |
plus_parity x Either = Either

```

First we instantiate the abstract value interface and prove that the functions on type *parity* have all the necessary properties:

```

global_interpretation Val_semilattice
where  $\gamma = \gamma\_parity$  and  $num' = num\_parity$  and  $plus' = plus\_parity$ 
proof (standard, goal_cases)

```

subgoals are the locale axioms

```

case 1 thus ?case by(auto simp: less_eq_parity_def)
next
case 2 show ?case by(auto simp: top_parity_def)
next
case 3 show ?case by auto
next
case (4 - a1 - a2) thus ?case
by (induction a1 a2 rule: plus_parity.induct)
(auto simp add: mod_add_eq [symmetric])
qed

```

In case 4 we needed to refer to particular variables. Writing (i x y z) fixes the names of the variables in case i to be x, y and z in the left-to-right order in which the variables occur in the subgoal. Underscores are anonymous placeholders for variable names we don't care to fix.

Instantiating the abstract interpretation locale requires no more proofs (they happened in the instantiation above) but delivers the instantiated abstract interpreter which we call *AI\_parity*:

```

global_interpretation Abs_Int
where  $\gamma = \gamma\_parity$  and  $num' = num\_parity$  and  $plus' = plus\_parity$ 
defines  $aval\_parity = aval'$  and  $step\_parity = step'$  and  $AI\_parity = AI$ 
..

```

#### 14.10.1 Tests

```

definition test1_parity =
  "x" ::= N 1;;
  WHILE Less (V "x") (N 100) DO "x" ::= Plus (V "x") (N 2)
value show_acom (the(AI_parity test1_parity))

```

```

definition test2_parity =

```

```

"x" ::= N 1;;
WHILE Less (V "x") (N 100) DO "x" ::= Plus (V "x") (N 3)

```

**definition** *steps c i* = ((*step\_parity*  $\top$ )  $\wedge$  *i*) (*bot c*)

```

value show_acom (steps test2_parity 0)
value show_acom (steps test2_parity 1)
value show_acom (steps test2_parity 2)
value show_acom (steps test2_parity 3)
value show_acom (steps test2_parity 4)
value show_acom (steps test2_parity 5)
value show_acom (steps test2_parity 6)
value show_acom (the(AI_parity test2_parity))

```

### 14.10.2 Termination

```

global_interpretation Abs_Int_mono
where  $\gamma = \gamma\_parity$  and  $num' = num\_parity$  and  $plus' = plus\_parity$ 
proof (standard, goal_cases)
  case (1 - a1 - a2) thus ?case
    by(induction a1 a2 rule: plus_parity.induct)
      (auto simp add:less_eq_parity_def)
qed

```

**definition** *m\_parity* :: *parity*  $\Rightarrow$  *nat* **where**  
*m\_parity x* = (if *x* = *Either* then 0 else 1)

```

global_interpretation Abs_Int_measure
where  $\gamma = \gamma\_parity$  and  $num' = num\_parity$  and  $plus' = plus\_parity$ 
and  $m = m\_parity$  and  $h = 1$ 
proof (standard, goal_cases)
  case 1 thus ?case by(auto simp add: m_parity_def less_eq_parity_def)
next
  case 2 thus ?case by(auto simp add: m_parity_def less_eq_parity_def less_parity_def)
qed

```

**thm** *AI\_Some\_measure*

**end**

## 14.11 Backward Analysis of Expressions

```

theory Abs_Int2
imports Abs_Int1

```

```

begin

instantiation prod :: (order,order) order
begin

definition less_eq_prod p1 p2 = (fst p1 ≤ fst p2 ∧ snd p1 ≤ snd p2)
definition less_prod p1 p2 = (p1 ≤ p2 ∧ ¬ p2 ≤ (p1::'a*'b))

instance
proof (standard, goal_cases)
  case 1 show ?case by(rule less_prod_def)
next
  case 2 show ?case by(simp add: less_eq_prod_def)
next
  case 3 thus ?case unfolding less_eq_prod_def by(metis order_trans)
next
  case 4 thus ?case by(simp add: less_eq_prod_def)(metis eq_iff surjective_pairing)
qed

end

```

#### 14.11.1 Extended Framework

```

subclass (in bounded_lattice) semilattice_sup_top ..

locale Val_lattice_gamma = Gamma_semilattice where  $\gamma = \gamma$ 
  for  $\gamma :: 'a :: bounded\_lattice \Rightarrow val\ set +$ 
assumes inter_gamma_subset_gamma_inf:
   $\gamma\ a1 \sqcap \gamma\ a2 \subseteq \gamma(a1 \sqcap a2)$ 
and gamma_bot[simp]:  $\gamma \perp = \{\}$ 
begin

lemma in_gamma_inf:  $x \in \gamma\ a1 \Longrightarrow x \in \gamma\ a2 \Longrightarrow x \in \gamma(a1 \sqcap a2)$ 
by (metis IntI inter_gamma_subset_gamma_inf set_mp)

lemma gamma_inf:  $\gamma(a1 \sqcap a2) = \gamma\ a1 \sqcap \gamma\ a2$ 
by(rule equalityI[OF _ inter_gamma_subset_gamma_inf])
  (metis inf_le1 inf_le2 le_inf_iff mono_gamma)

end

locale Val_inv = Val_lattice_gamma where  $\gamma = \gamma$ 

```

```

for  $\gamma :: 'av::bounded\_lattice \Rightarrow val\ set +$ 
fixes  $test\_num' :: val \Rightarrow 'av \Rightarrow bool$ 
and  $inv\_plus' :: 'av \Rightarrow 'av \Rightarrow 'av \Rightarrow 'av * 'av$ 
and  $inv\_less' :: bool \Rightarrow 'av \Rightarrow 'av \Rightarrow 'av * 'av$ 
assumes  $test\_num': test\_num' i a = (i \in \gamma a)$ 
and  $inv\_plus': inv\_plus' a a1 a2 = (a1', a2') \Longrightarrow$ 
 $i1 \in \gamma a1 \Longrightarrow i2 \in \gamma a2 \Longrightarrow i1+i2 \in \gamma a \Longrightarrow i1 \in \gamma a1' \wedge i2 \in \gamma a2'$ 
and  $inv\_less': inv\_less' (i1 < i2) a1 a2 = (a1', a2') \Longrightarrow$ 
 $i1 \in \gamma a1 \Longrightarrow i2 \in \gamma a2 \Longrightarrow i1 \in \gamma a1' \wedge i2 \in \gamma a2'$ 

```

```

locale  $Abs\_Int\_inv = Val\_inv$  where  $\gamma = \gamma$ 

```

```

for  $\gamma :: 'av::bounded\_lattice \Rightarrow val\ set$ 
begin

```

```

lemma  $in\_gamma\_sup\_UpI:$ 

```

```

 $s \in \gamma_o S1 \vee s \in \gamma_o S2 \Longrightarrow s \in \gamma_o(S1 \sqcup S2)$ 

```

```

by ( $metis (hide\_lams, no\_types) sup\_ge1 sup\_ge2 mono\_gamma\_o subsetD$ )

```

```

fun  $aval'' :: aexp \Rightarrow 'av\ st\ option \Rightarrow 'av$  where

```

```

 $aval'' e\ None = \perp \mid$ 

```

```

 $aval'' e\ (Some\ S) = aval'\ e\ S$ 

```

```

lemma  $aval''\_correct: s \in \gamma_o S \Longrightarrow aval\ a\ s \in \gamma(aval'' a\ S)$ 

```

```

by( $cases\ S$ )( $auto\ simp\ add: aval'\_correct\ split: option.splits$ )

```

### 14.11.2 Backward analysis

```

fun  $inv\_aval' :: aexp \Rightarrow 'av \Rightarrow 'av\ st\ option \Rightarrow 'av\ st\ option$  where

```

```

 $inv\_aval' (N\ n) a\ S = (if\ test\_num'\ n\ a\ then\ S\ else\ None) \mid$ 

```

```

 $inv\_aval' (V\ x) a\ S = (case\ S\ of\ None \Rightarrow None \mid Some\ S \Rightarrow$ 

```

```

 $let\ a' = fun\ S\ x \sqcap a\ in$ 

```

```

 $if\ a' = \perp\ then\ None\ else\ Some(update\ S\ x\ a')) \mid$ 

```

```

 $inv\_aval' (Plus\ e1\ e2) a\ S =$ 

```

```

 $(let\ (a1, a2) = inv\_plus'\ a\ (aval''\ e1\ S)\ (aval''\ e2\ S)$ 

```

```

 $in\ inv\_aval'\ e1\ a1\ (inv\_aval'\ e2\ a2\ S))$ 

```

The test for *bot* in the *V*-case is important: *bot* indicates that a variable has no possible values, i.e. that the current program point is unreachable. But then the abstract state should collapse to *None*. Put differently, we maintain the invariant that in an abstract state of the form *Some s*, all variables are mapped to non-*bot* values. Otherwise the (pointwise) sup of two abstract states, one of which contains *bot* values, may produce too large a result, thus making the analysis less precise.

```

fun inv_bval' :: bexp  $\Rightarrow$  bool  $\Rightarrow$  'av st option  $\Rightarrow$  'av st option where
inv_bval' (Bc v) res S = (if v=res then S else None) |
inv_bval' (Not b) res S = inv_bval' b ( $\neg$  res) S |
inv_bval' (And b1 b2) res S =
  (if res then inv_bval' b1 True (inv_bval' b2 True S)
   else inv_bval' b1 False S  $\sqcup$  inv_bval' b2 False S) |
inv_bval' (Less e1 e2) res S =
  (let (a1,a2) = inv_less' res (aval'' e1 S) (aval'' e2 S)
   in inv_aval' e1 a1 (inv_aval' e2 a2 S))

```

**lemma** inv\_aval'\_correct:  $s \in \gamma_o S \implies \text{aval } e \ s \in \gamma a \implies s \in \gamma_o (\text{inv\_aval}' e a S)$

**proof**(induction e arbitrary: a S)

**case** N **thus** ?case **by** simp (metis test\_num')

**next**

**case** (V x)

**obtain** S' **where** S = Some S' **and**  $s \in \gamma_s S'$  **using**  $\langle s \in \gamma_o S \rangle$

**by**(auto simp: in\_gamma\_option\_iff)

**moreover hence**  $s \ x \in \gamma (\text{fun } S' x)$

**by**(simp add:  $\gamma$ -st\_def)

**moreover have**  $s \ x \in \gamma a$  **using** V(2) **by** simp

**ultimately show** ?case

**by**(simp add: Let\_def  $\gamma$ -st\_def)

  (metis mono\_gamma\_emptyE in\_gamma\_inf gamma\_bot subset\_empty)

**next**

**case** (Plus e1 e2) **thus** ?case

**using** inv\_plus'[OF \_ aval''\_correct aval''\_correct]

**by** (auto split: prod.split)

**qed**

**lemma** inv\_bval'\_correct:  $s \in \gamma_o S \implies bv = \text{bval } b \ s \implies s \in \gamma_o (\text{inv\_bval}' b bv S)$

**proof**(induction b arbitrary: S bv)

**case** Bc **thus** ?case **by** simp

**next**

**case** (Not b) **thus** ?case **by** simp

**next**

**case** (And b1 b2) **thus** ?case

**by** simp (metis And(1) And(2) in\_gamma\_sup-UpI)

**next**

**case** (Less e1 e2) **thus** ?case

**apply** hypsubst\_thin

**apply** (auto split: prod.split)

**apply** (metis (lifting) inv\_aval'\_correct aval''\_correct inv\_less')

**done**  
**qed**

**definition**  $step' = Step$

$(\lambda x e S. case\ S\ of\ None \Rightarrow None \mid Some\ S \Rightarrow Some(update\ S\ x\ (aval'\ e\ S)))$   
 $(\lambda b S. inv\_bval'\ b\ True\ S)$

**definition**  $AI :: com \Rightarrow 'av\ st\ option\ acom\ option$  **where**  
 $AI\ c = pfp\ (step'\ \top)\ (bot\ c)$

**lemma**  $strip\_step'[simp]: strip(step'\ S\ c) = strip\ c$   
**by**( $simp\ add: step'\_def$ )

**lemma**  $top\_on\_inv\_aval': \llbracket top\_on\_opt\ S\ X; vars\ e \subseteq -X \rrbracket \Longrightarrow top\_on\_opt$   
 $(inv\_aval'\ e\ a\ S)\ X$   
**by**( $induction\ e\ arbitrary: a\ S$ ) ( $auto\ simp: Let\_def\ split: option.splits\ prod.split$ )

**lemma**  $top\_on\_inv\_bval': \llbracket top\_on\_opt\ S\ X; vars\ b \subseteq -X \rrbracket \Longrightarrow top\_on\_opt$   
 $(inv\_bval'\ b\ r\ S)\ X$   
**by**( $induction\ b\ arbitrary: r\ S$ ) ( $auto\ simp: top\_on\_inv\_aval'\ top\_on\_sup\ split: prod.split$ )

**lemma**  $top\_on\_step': top\_on\_acom\ C\ (-\ vars\ C) \Longrightarrow top\_on\_acom\ (step'\ \top\ C)\ (-\ vars\ C)$   
**unfolding**  $step'\_def$   
**by**( $rule\ top\_on\_Step$ )  
 $(auto\ simp\ add: top\_on\_top\ top\_on\_inv\_bval'\ split: option.split)$

### 14.11.3 Correctness

**lemma**  $step\_step': step\ (\gamma_o\ S)\ (\gamma_c\ C) \leq \gamma_c\ (step'\ S\ C)$

**unfolding**  $step\_def\ step'\_def$

**by**( $rule\ gamma\_Step\_subcomm$ )

$(auto\ simp: intro!: aval'\_correct\ inv\_bval'\_correct\ in\_gamma\_update\ split: option.splits)$

**lemma**  $AI\_correct: AI\ c = Some\ C \Longrightarrow CS\ c \leq \gamma_c\ C$

**proof**( $simp\ add: CS\_def\ AI\_def$ )

**assume**  $1: pfp\ (step'\ \top)\ (bot\ c) = Some\ C$

**have**  $pfp': step'\ \top\ C \leq C$  **by**( $rule\ pfp\_pfp[OF\ 1]$ )

**have**  $2: step\ (\gamma_o\ \top)\ (\gamma_c\ C) \leq \gamma_c\ C$  — transfer the pfp'

**proof**( $rule\ order\_trans$ )

**show**  $step\ (\gamma_o\ \top)\ (\gamma_c\ C) \leq \gamma_c\ (step'\ \top\ C)$  **by**( $rule\ step\_step'$ )



```

  show ... ≤ γc C by (metis mono_gamma_c[OF pfp'])
qed
have 3: strip (γc C) = c by (simp add: strip_pfp[OF - 1] step'_def)
have lfp c (step (γo ⊤)) ≤ γc C
  by (rule lfp_lowerbound[simplified, where f=step (γo ⊤), OF 3 2])
thus lfp c (step UNIV) ≤ γc C by simp
qed

end

```

#### 14.11.4 Monotonicity

```

locale Abs_Int_inv_mono = Abs_Int_inv +
assumes mono_plus': a1 ≤ b1 ⇒ a2 ≤ b2 ⇒ plus' a1 a2 ≤ plus' b1 b2
and mono_inv_plus': a1 ≤ b1 ⇒ a2 ≤ b2 ⇒ r ≤ r' ⇒
  inv_plus' r a1 a2 ≤ inv_plus' r' b1 b2
and mono_inv_less': a1 ≤ b1 ⇒ a2 ≤ b2 ⇒
  inv_less' bv a1 a2 ≤ inv_less' bv b1 b2
begin

```

```

lemma mono_aval':
  S1 ≤ S2 ⇒ aval' e S1 ≤ aval' e S2
by (induction e) (auto simp: mono_plus' mono_fun)

```

```

lemma mono_aval'':
  S1 ≤ S2 ⇒ aval'' e S1 ≤ aval'' e S2
apply (cases S1)
  apply simp
apply (cases S2)
  apply simp
by (simp add: mono_aval')

```

```

lemma mono_inv_aval': r1 ≤ r2 ⇒ S1 ≤ S2 ⇒ inv_aval' e r1 S1 ≤
  inv_aval' e r2 S2
apply (induction e arbitrary: r1 r2 S1 S2)
  apply (auto simp: test_num' Let_def inf_mono split: option.splits prod.splits)
  apply (metis mono_gamma_subsetD)
  apply (metis le_bot inf_mono le_st_iff)
  apply (metis inf_mono mono_update le_st_iff)
apply (metis mono_aval'' mono_inv_plus'[simplified less_eq_prod_def] fst_conv
  snd_conv)
done

```

```

lemma mono_inv_bval': S1 ≤ S2 ⇒ inv_bval' b bv S1 ≤ inv_bval' b bv S2

```

```

apply(induction b arbitrary: bv S1 S2)
  apply(simp)
  apply(simp)
  apply simp
  apply(metis order_trans[OF - sup_ge1] order_trans[OF - sup_ge2])
apply (simp split: prod.splits)
apply(metis mono_aval'' mono_inv_aval' mono_inv_less'[simplified less_eq_prod_def]
fst_conv snd_conv)
done

```

**theorem** *mono\_step'*:  $S1 \leq S2 \implies C1 \leq C2 \implies \text{step}' S1 C1 \leq \text{step}' S2 C2$

```

unfolding step'_def
by(rule mono2_Step) (auto simp: mono_aval' mono_inv_bval' split: option.split)

```

**lemma** *mono\_step'\_top*:  $C1 \leq C2 \implies \text{step}' \top C1 \leq \text{step}' \top C2$   
**by** (*metis mono\_step' order\_refl*)

**end**

**end**

## 14.12 Interval Analysis

```

theory Abs_Int2_ivl
imports Abs_Int2
begin

```

```

type_synonym eint = int extended
type_synonym eint2 = eint * eint

```

**definition** *γ\_rep* :: *eint2*  $\Rightarrow$  *int set* **where**  
 $\gamma\_rep\ p = (\text{let } (l,h) = p \text{ in } \{i. l \leq \text{Fin } i \wedge \text{Fin } i \leq h\})$

**definition** *eq\_ivl* :: *eint2*  $\Rightarrow$  *eint2*  $\Rightarrow$  *bool* **where**  
 $eq\_ivl\ p1\ p2 = (\gamma\_rep\ p1 = \gamma\_rep\ p2)$

**lemma** *refl\_eq\_ivl[simp]*:  $eq\_ivl\ p\ p$   
**by**(*auto simp: eq\_ivl\_def*)

**quotient\_type** *ivl* = *eint2* / *eq\_ivl*  
**by**(*rule equivpI*)(*auto simp: reflp\_def symp\_def transp\_def eq\_ivl\_def*)

**abbreviation** *ivl\_abbrev* :: *eint*  $\Rightarrow$  *eint*  $\Rightarrow$  *ivl* (*[-, -]*) **where**

$[l,h] == \text{abs\_ivl}(l,h)$

**lift\_definition**  $\gamma\_ivl :: ivl \Rightarrow int \text{ set is } \gamma\_rep$   
**by**(*simp add: eq\_ivl\_def*)

**lemma**  $\gamma\_ivl\_nice: \gamma\_ivl[l,h] = \{i. l \leq Fin\ i \wedge Fin\ i \leq h\}$   
**by** *transfer (simp add:  $\gamma\_rep\_def$ )*

**lift\_definition**  $num\_ivl :: int \Rightarrow ivl \text{ is } \lambda i. (Fin\ i, Fin\ i) .$

**lift\_definition**  $in\_ivl :: int \Rightarrow ivl \Rightarrow bool$   
**is**  $\lambda i (l,h). l \leq Fin\ i \wedge Fin\ i \leq h$   
**by**(*auto simp: eq\_ivl\_def  $\gamma\_rep\_def$* )

**lemma**  $in\_ivl\_nice: in\_ivl\ i\ [l,h] = (l \leq Fin\ i \wedge Fin\ i \leq h)$   
**by** *transfer simp*

**definition**  $is\_empty\_rep :: eint2 \Rightarrow bool \text{ where}$   
 $is\_empty\_rep\ p = (let\ (l,h) = p\ in\ l > h \mid l = Pinf \ \&\ h = Pinf \mid l = Minf \ \&$   
 $h = Minf)$

**lemma**  $\gamma\_rep\_cases: \gamma\_rep\ p = (case\ p\ of\ (Fin\ i, Fin\ j) \Rightarrow \{i..j\} \mid (Fin$   
 $i, Pinf) \Rightarrow \{i.. \} \mid$   
 $(Minf, Fin\ i) \Rightarrow \{..i\} \mid (Minf, Pinf) \Rightarrow UNIV \mid \_ \Rightarrow \{\})$   
**by**(*auto simp add:  $\gamma\_rep\_def\ split: prod.splits\ extended.splits$* )

**lift\_definition**  $is\_empty\_ivl :: ivl \Rightarrow bool \text{ is } is\_empty\_rep$   
**apply**(*auto simp: eq\_ivl\_def  $\gamma\_rep\_cases\ is\_empty\_rep\_def$* )  
**apply**(*auto simp: not\_less less\_eq\_extended\_case split: extended.splits*)  
**done**

**lemma**  $eq\_ivl\_iff: eq\_ivl\ p1\ p2 = (is\_empty\_rep\ p1 \ \&\ is\_empty\_rep\ p2 \mid p1$   
 $= p2)$   
**by**(*auto simp: eq\_ivl\_def is\_empty\_rep\_def  $\gamma\_rep\_cases\ Icc\_eq\_Icc\ split: prod.splits$*   
*extended.splits*)

**definition**  $empty\_rep :: eint2 \text{ where } empty\_rep = (Pinf, Minf)$

**lift\_definition**  $empty\_ivl :: ivl \text{ is } empty\_rep .$

**lemma**  $is\_empty\_empty\_rep[simp]: is\_empty\_rep\ empty\_rep$   
**by**(*auto simp add: is\_empty\_rep\_def empty\_rep\_def*)

**lemma**  $is\_empty\_rep\_iff: is\_empty\_rep\ p = (\gamma\_rep\ p = \{\})$

**by**(*auto simp add:  $\gamma$ \_rep\_cases is\_empty\_rep\_def split: prod.splits extended.splits*)

**declare** *is\_empty\_rep\_iff*[*THEN iffD1, simp*]

**instantiation** *ivl* :: *semilattice\_sup\_top*  
**begin**

**definition** *le\_rep* :: *eint2*  $\Rightarrow$  *eint2*  $\Rightarrow$  *bool* **where**  
*le\_rep* *p1* *p2* = (let (*l1*,*h1*) = *p1*; (*l2*,*h2*) = *p2* in  
  if *is\_empty\_rep*(*l1*,*h1*) then *True* else  
  if *is\_empty\_rep*(*l2*,*h2*) then *False* else *l1*  $\geq$  *l2* & *h1*  $\leq$  *h2*)

**lemma** *le\_iff\_subset*: *le\_rep* *p1* *p2*  $\longleftrightarrow$   $\gamma$ \_rep *p1*  $\subseteq$   $\gamma$ \_rep *p2*

**apply** *rule*

**apply**(*auto simp: is\_empty\_rep\_def le\_rep\_def  $\gamma$ \_rep\_def split: if\_splits prod.splits*)[1]

**apply**(*auto simp: is\_empty\_rep\_def  $\gamma$ \_rep\_cases le\_rep\_def*)

**apply**(*auto simp: not\_less split: extended.splits*)

**done**

**lift\_definition** *less\_eq\_ivl* :: *ivl*  $\Rightarrow$  *ivl*  $\Rightarrow$  *bool* **is** *le\_rep*

**by**(*auto simp: eq\_ivl\_def le\_iff\_subset*)

**definition** *less\_ivl* **where** *i1* < *i2* = (*i1*  $\leq$  *i2*  $\wedge$   $\neg$  *i2*  $\leq$  (*i1*::*ivl*))

**lemma** *le\_ivl\_iff\_subset*: *iv1*  $\leq$  *iv2*  $\longleftrightarrow$   $\gamma$ \_ivl *iv1*  $\subseteq$   $\gamma$ \_ivl *iv2*

**by** *transfer* (*rule le\_iff\_subset*)

**definition** *sup\_rep* :: *eint2*  $\Rightarrow$  *eint2*  $\Rightarrow$  *eint2* **where**

*sup\_rep* *p1* *p2* = (if *is\_empty\_rep* *p1* then *p2* else if *is\_empty\_rep* *p2* then *p1*  
  else let (*l1*,*h1*) = *p1*; (*l2*,*h2*) = *p2* in (min *l1* *l2*, max *h1* *h2*))

**lift\_definition** *sup\_ivl* :: *ivl*  $\Rightarrow$  *ivl*  $\Rightarrow$  *ivl* **is** *sup\_rep*

**by**(*auto simp: eq\_ivl\_iff sup\_rep\_def*)

**lift\_definition** *top\_ivl* :: *ivl* **is** (*Minf*,*Pinf*) .

**lemma** *is\_empty\_min\_max*:

$\neg$  *is\_empty\_rep* (*l1*,*h1*)  $\Longrightarrow$   $\neg$  *is\_empty\_rep* (*l2*, *h2*)  $\Longrightarrow$   $\neg$  *is\_empty\_rep*  
(min *l1* *l2*, max *h1* *h2*)

**by**(*auto simp add: is\_empty\_rep\_def max\_def min\_def split: if\_splits*)

**instance**

**proof** (*standard, goal\_cases*)

```

    case 1 show ?case by (rule less_ivl_def)
next
    case 2 show ?case by transfer (simp add: le_rep_def split: prod.splits)
next
    case 3 thus ?case by transfer (auto simp: le_rep_def split: if_splits)
next
    case 4 thus ?case by transfer (auto simp: le_rep_def eq_ivl_iff split:
if_splits)
next
    case 5 thus ?case by transfer (auto simp add: le_rep_def sup_rep_def
is_empty_min_max)
next
    case 6 thus ?case by transfer (auto simp add: le_rep_def sup_rep_def
is_empty_min_max)
next
    case 7 thus ?case by transfer (auto simp add: le_rep_def sup_rep_def)
next
    case 8 show ?case by transfer (simp add: le_rep_def is_empty_rep_def)
qed

end

```

Implement (naive) executable equality:

```

instantiation ivl :: equal
begin

```

```

definition equal_ivl where
equal_ivl i1 (i2::ivl) = (i1 ≤ i2 ∧ i2 ≤ i1)

```

```

instance

```

```

proof (standard, goal_cases)

```

```

    case 1 show ?case by (simp add: equal_ivl_def eq_iff)
qed

```

```

end

```

```

lemma [simp]: fixes x :: 'a::linorder extended shows (¬ x < Pinf) = (x
= Pinf)

```

```

by(simp add: not_less)

```

```

lemma [simp]: fixes x :: 'a::linorder extended shows (¬ Minf < x) = (x
= Minf)

```

```

by(simp add: not_less)

```

```

instantiation ivl :: bounded_lattice

```

**begin**

**definition**  $inf\_rep :: eint2 \Rightarrow eint2 \Rightarrow eint2$  **where**

$inf\_rep\ p1\ p2 = (let\ (l1,h1) = p1;\ (l2,h2) = p2\ in\ (max\ l1\ l2,\ min\ h1\ h2))$

**lemma**  $\gamma\_inf\_rep: \gamma\_rep(inf\_rep\ p1\ p2) = \gamma\_rep\ p1 \cap \gamma\_rep\ p2$

**by**(*auto simp: inf\_rep\_def  $\gamma\_rep\_cases$  split: prod.splits extended.splits*)

**lift\_definition**  $inf\_ivl :: ivl \Rightarrow ivl \Rightarrow ivl$  **is**  $inf\_rep$

**by**(*auto simp:  $\gamma\_inf\_rep$  eq\_ivl\_def*)

**lemma**  $\gamma\_inf: \gamma\_ivl\ (iv1 \sqcap iv2) = \gamma\_ivl\ iv1 \cap \gamma\_ivl\ iv2$

**by** *transfer (rule  $\gamma\_inf\_rep$ )*

**definition**  $\perp = empty\_ivl$

**instance**

**proof** (*standard, goal\_cases*)

**case** 1 **thus** *?case* **by** (*simp add:  $\gamma\_inf$  le\_ivl\_iff\_subset*)

**next**

**case** 2 **thus** *?case* **by** (*simp add:  $\gamma\_inf$  le\_ivl\_iff\_subset*)

**next**

**case** 3 **thus** *?case* **by** (*simp add:  $\gamma\_inf$  le\_ivl\_iff\_subset*)

**next**

**case** 4 **show** *?case*

**unfolding** *bot\_ivl\_def* **by** *transfer (auto simp: le\_iff\_subset)*

**qed**

**end**

**lemma**  $eq\_ivl\_empty: eq\_ivl\ p\ empty\_rep = is\_empty\_rep\ p$

**by** (*metis eq\_ivl\_iff is\_empty\_empty\_rep*)

**lemma**  $le\_ivl\_nice: [l1,h1] \leq [l2,h2] \longleftrightarrow$

*(if  $[l1,h1] = \perp$  then True else*

*if  $[l2,h2] = \perp$  then False else  $l1 \geq l2 \ \& \ h1 \leq h2$ )*

**unfolding** *bot\_ivl\_def* **by** *transfer (simp add: le\_rep\_def eq\_ivl\_empty)*

**lemma**  $sup\_ivl\_nice: [l1,h1] \sqcup [l2,h2] =$

*(if  $[l1,h1] = \perp$  then  $[l2,h2]$  else*

*if  $[l2,h2] = \perp$  then  $[l1,h1]$  else  $[min\ l1\ l2, max\ h1\ h2]$ )*

**unfolding** *bot\_ivl\_def* **by** *transfer (simp add: sup\_rep\_def eq\_ivl\_empty)*

**lemma** *inf\_ivl\_nice*:  $[l1, h1] \sqcap [l2, h2] = [\max l1\ l2, \min h1\ h2]$   
**by** *transfer (simp add: inf\_rep\_def)*

**lemma** *top\_ivl\_nice*:  $\top = [-\infty, \infty]$   
**by** *(simp add: top\_ivl\_def)*

**instantiation** *ivl :: plus*  
**begin**

**definition** *plus\_rep :: eint2  $\Rightarrow$  eint2  $\Rightarrow$  eint2* **where**  
*plus\_rep p1 p2 =*  
*(if is\_empty\_rep p1  $\vee$  is\_empty\_rep p2 then empty\_rep else*  
*let (l1, h1) = p1; (l2, h2) = p2 in (l1+l2, h1+h2))*

**lift\_definition** *plus\_ivl :: ivl  $\Rightarrow$  ivl  $\Rightarrow$  ivl* **is** *plus\_rep*  
**by***(auto simp: plus\_rep\_def eq\_ivl\_iff)*

**instance** ..  
**end**

**lemma** *plus\_ivl\_nice*:  $[l1, h1] + [l2, h2] =$   
*(if  $[l1, h1] = \perp \vee [l2, h2] = \perp$  then  $\perp$  else  $[l1+l2, h1+h2]$ )*  
**unfolding** *bot\_ivl\_def* **by** *transfer (auto simp: plus\_rep\_def eq\_ivl\_empty)*

**lemma** *uminus\_eq\_Min**f* $[simp]$ :  $-x = Min\ f \iff x = Pin\ f$   
**by***(cases x) auto*

**lemma** *uminus\_eq\_Pinf* $[simp]$ :  $-x = Pin\ f \iff x = Min\ f$   
**by***(cases x) auto*

**lemma** *uminus\_le\_Fin\_iff*:  $-x \leq Fin(-y) \iff Fin\ y \leq (x::'a::ordered\_ab\_group\_add\ extended)$

**by***(cases x) auto*

**lemma** *Fin\_uminus\_le\_iff*:  $Fin(-y) \leq -x \iff x \leq ((Fin\ y)::'a::ordered\_ab\_group\_add\ extended)$

**by***(cases x) auto*

**instantiation** *ivl :: uminus*  
**begin**

**definition** *uminus\_rep :: eint2  $\Rightarrow$  eint2* **where**  
*uminus\_rep p = (let (l, h) = p in (-h, -l))*

**lemma**  *$\gamma$ \_uminus\_rep*:  $i \in \gamma\_rep\ p \implies -i \in \gamma\_rep(uminus\_rep\ p)$

**by**(*auto simp: uminus\_rep\_def  $\gamma$ \_rep\_def image\_def uminus\_le\_Fin\_iff Fin\_uminus\_le\_iff split: prod.split*)

**lift\_definition** *uminus\_ivl* :: *ivl*  $\Rightarrow$  *ivl* **is** *uminus\_rep*  
**by** (*auto simp: uminus\_rep\_def eq\_ivl\_def  $\gamma$ \_rep\_cases*)  
(*auto simp: Icc\_eq\_Icc split: extended.splits*)

**instance ..**  
**end**

**lemma**  *$\gamma$ \_uminus*:  $i \in \gamma\_ivl\ iv \Longrightarrow -i \in \gamma\_ivl(-\ iv)$   
**by** *transfer (rule  $\gamma$ \_uminus\_rep)*

**lemma** *uminus\_nice*:  $-[l,h] = [-h,-l]$   
**by** *transfer (simp add: uminus\_rep\_def)*

**instantiation** *ivl* :: *minus*  
**begin**

**definition** *minus\_ivl* :: *ivl*  $\Rightarrow$  *ivl*  $\Rightarrow$  *ivl* **where**  
(*iv1::ivl*) - *iv2* = *iv1* + -*iv2*

**instance ..**  
**end**

**definition** *inv\_plus\_ivl* :: *ivl*  $\Rightarrow$  *ivl*  $\Rightarrow$  *ivl*  $\Rightarrow$  *ivl\*ivl* **where**  
*inv\_plus\_ivl* *iv* *iv1* *iv2* = (*iv1*  $\sqcap$  (*iv* - *iv2*), *iv2*  $\sqcap$  (*iv* - *iv1*))

**definition** *above\_rep* :: *eint2*  $\Rightarrow$  *eint2* **where**  
*above\_rep* *p* = (*if is\_empty\_rep p then empty\_rep else let (l,h) = p in (l, $\infty$ )*)

**definition** *below\_rep* :: *eint2*  $\Rightarrow$  *eint2* **where**  
*below\_rep* *p* = (*if is\_empty\_rep p then empty\_rep else let (l,h) = p in ( $-\infty$ ,h)*)

**lift\_definition** *above* :: *ivl*  $\Rightarrow$  *ivl* **is** *above\_rep*  
**by**(*auto simp: above\_rep\_def eq\_ivl\_iff*)

**lift\_definition** *below* :: *ivl*  $\Rightarrow$  *ivl* **is** *below\_rep*  
**by**(*auto simp: below\_rep\_def eq\_ivl\_iff*)

**lemma**  *$\gamma$ \_aboveI*:  $i \in \gamma\_ivl\ iv \Longrightarrow i \leq j \Longrightarrow j \in \gamma\_ivl(\text{above } iv)$   
**by** *transfer*  
(*auto simp add: above\_rep\_def  $\gamma$ \_rep\_cases is\_empty\_rep\_def*)



*split: extended.splits*)

**lemma**  $\gamma\_belowI: i \in \gamma\_ivl\ iv \implies j \leq i \implies j \in \gamma\_ivl(below\ iv)$   
**by** *transfer*  
(*auto simp add: below\_rep\_def  $\gamma\_rep\_cases$  is\_empty\_rep\_def*  
*split: extended.splits*)

**definition**  $inv\_less\_ivl :: bool \Rightarrow ivl \Rightarrow ivl \Rightarrow ivl * ivl$  **where**  
 $inv\_less\_ivl\ res\ iv1\ iv2 =$   
(*if* *res*  
  *then* ( $iv1 \sqcap (below\ iv2 - [1,1]),$   
     $iv2 \sqcap (above\ iv1 + [1,1])$ )  
  *else* ( $iv1 \sqcap above\ iv2, iv2 \sqcap below\ iv1$ ))

**lemma** *above\_nice*:  $above[l,h] = (if\ [l,h] = \perp\ then\ \perp\ else\ [l,\infty])$   
**unfolding** *bot\_ivl\_def* **by** *transfer* (*simp add: above\_rep\_def eq\_ivl\_empty*)

**lemma** *below\_nice*:  $below[l,h] = (if\ [l,h] = \perp\ then\ \perp\ else\ [-\infty,h])$   
**unfolding** *bot\_ivl\_def* **by** *transfer* (*simp add: below\_rep\_def eq\_ivl\_empty*)

**lemma** *add\_mono\_le\_Fin*:  
 $\llbracket x1 \leq Fin\ y1; x2 \leq Fin\ y2 \rrbracket \implies x1 + x2 \leq Fin\ (y1 + (y2::'a::ordered\_ab\_group\_add))$   
**by**(*drule* (1) *add\_mono*) *simp*

**lemma** *add\_mono\_Fin\_le*:  
 $\llbracket Fin\ y1 \leq x1; Fin\ y2 \leq x2 \rrbracket \implies Fin(y1 + y2::'a::ordered\_ab\_group\_add)$   
 $\leq x1 + x2$   
**by**(*drule* (1) *add\_mono*) *simp*

**global\\_interpretation** *Val\\_semilattice*  
**where**  $\gamma = \gamma\_ivl$  **and**  $num' = num\_ivl$  **and**  $plus' = (+)$   
**proof** (*standard, goal\_cases*)  
  **case** 1 **thus** ?*case* **by** *transfer* (*simp add: le\_iff\_subset*)  
**next**  
  **case** 2 **show** ?*case* **by** *transfer* (*simp add:  $\gamma\_rep\_def$* )  
**next**  
  **case** 3 **show** ?*case* **by** *transfer* (*simp add:  $\gamma\_rep\_def$* )  
**next**  
  **case** 4 **thus** ?*case*  
    **apply** *transfer*  
    **apply**(*auto simp:  $\gamma\_rep\_def$  plus\_rep\_def add\_mono\_le\_Fin add\_mono\_Fin\_le*)  
    **by**(*auto simp: empty\_rep\_def is\_empty\_rep\_def*)  
**qed**

```

global_interpretation Val_lattice_gamma
where  $\gamma = \gamma_{ivl}$  and  $num' = num_{ivl}$  and  $plus' = (+)$ 
defines  $aval_{ivl} = aval'$ 
proof (standard, goal_cases)
  case 1 show ?case by(simp add:  $\gamma_{inf}$ )
next
  case 2 show ?case unfolding bot_ivl_def by transfer simp
qed

```

```

global_interpretation Val_inv
where  $\gamma = \gamma_{ivl}$  and  $num' = num_{ivl}$  and  $plus' = (+)$ 
and  $test\_num' = in_{ivl}$ 
and  $inv\_plus' = inv\_plus_{ivl}$  and  $inv\_less' = inv\_less_{ivl}$ 
proof (standard, goal_cases)
  case 1 thus ?case by transfer (auto simp:  $\gamma_{rep\_def}$ )
next
  case (2 _ _ _ _ i1 i2) thus ?case
    unfolding inv_plus_ivl_def minus_ivl_def
    apply(clarsimp simp add:  $\gamma_{inf}$ )
    using  $\gamma_{plus'}$ [of i1+i2 - -i1]  $\gamma_{plus'}$ [of i1+i2 - -i2]
    by(simp add:  $\gamma_{uminus}$ )
next
  case (3 i1 i2) thus ?case
    unfolding inv_less_ivl_def minus_ivl_def one_extended_def
    apply(clarsimp simp add:  $\gamma_{inf}$  split: if_splits)
    using  $\gamma_{plus'}$ [of i1+1 - -1]  $\gamma_{plus'}$ [of i2 - 1 - 1]
    apply(simp add:  $\gamma_{belowI}$ [of i2]  $\gamma_{aboveI}$ [of i1]
      uminus_ivl.abs_eq uminus_rep_def  $\gamma_{ivl\_nice}$ )
    apply(simp add:  $\gamma_{aboveI}$ [of i2]  $\gamma_{belowI}$ [of i1])
    done
qed

```

```

global_interpretation Abs_Int_inv
where  $\gamma = \gamma_{ivl}$  and  $num' = num_{ivl}$  and  $plus' = (+)$ 
and  $test\_num' = in_{ivl}$ 
and  $inv\_plus' = inv\_plus_{ivl}$  and  $inv\_less' = inv\_less_{ivl}$ 
defines  $inv\_aval_{ivl} = inv\_aval'$ 
and  $inv\_bval_{ivl} = inv\_bval'$ 
and  $step_{ivl} = step'$ 
and  $AI_{ivl} = AI$ 
and  $aval_{ivl}' = aval''$ 
..

```

Monotonicity:

```
lemma mono_plus_ivl:  $iv1 \leq iv2 \implies iv3 \leq iv4 \implies iv1 + iv3 \leq iv2 + (iv4::ivl)$   
apply transfer  
apply (auto simp: plus_rep_def le_iff_subset split: if_splits)  
by (auto simp: is_empty_rep_iff  $\gamma$ _rep_cases split: extended_splits)
```

```
lemma mono_minus_ivl:  $iv1 \leq iv2 \implies -iv1 \leq -(iv2::ivl)$   
apply transfer  
apply (auto simp: uminus_rep_def le_iff_subset split: if_splits prod.split)  
by (auto simp:  $\gamma$ _rep_cases split: extended_splits)
```

```
lemma mono_above:  $iv1 \leq iv2 \implies \text{above } iv1 \leq \text{above } iv2$   
apply transfer  
apply (auto simp: above_rep_def le_iff_subset split: if_splits prod.split)  
by (auto simp: is_empty_rep_iff  $\gamma$ _rep_cases split: extended_splits)
```

```
lemma mono_below:  $iv1 \leq iv2 \implies \text{below } iv1 \leq \text{below } iv2$   
apply transfer  
apply (auto simp: below_rep_def le_iff_subset split: if_splits prod.split)  
by (auto simp: is_empty_rep_iff  $\gamma$ _rep_cases split: extended_splits)
```

```
global interpretation Abs_Int_inv_mono  
where  $\gamma = \gamma_{ivl}$  and  $num' = num_{ivl}$  and  $plus' = (+)$   
and  $test\_num' = in_{ivl}$   
and  $inv\_plus' = inv\_plus_{ivl}$  and  $inv\_less' = inv\_less_{ivl}$   
proof (standard, goal_cases)  
  case 1 thus ?case by (rule mono_plus_ivl)  
next  
  case 2 thus ?case  
    unfolding inv_plus_ivl_def minus_ivl_def less_eq_prod_def  
    by (auto simp: le_infI1 le_infI2 mono_plus_ivl mono_minus_ivl)  
next  
  case 3 thus ?case  
    unfolding less_eq_prod_def inv_less_ivl_def minus_ivl_def  
    by (auto simp: le_infI1 le_infI2 mono_plus_ivl mono_above mono_below)  
qed
```

### 14.12.1 Tests

```
value show_acom_opt (AI_ivl test1_ivl)
```

Better than *AI.const*:

```
value show_acom_opt (AI_ivl test3_const)  
value show_acom_opt (AI_ivl test4_const)
```

```
value show_acom_opt (AI_ivl test6_const)
```

```
definition steps c i = (step_ivl  $\top$   $\wedge$  i) (bot c)
```

```
value show_acom_opt (AI_ivl test2_ivl)
```

```
value show_acom (steps test2_ivl 0)
```

```
value show_acom (steps test2_ivl 1)
```

```
value show_acom (steps test2_ivl 2)
```

```
value show_acom (steps test2_ivl 3)
```

Fixed point reached in 2 steps. Not so if the start value of x is known:

```
value show_acom_opt (AI_ivl test3_ivl)
```

```
value show_acom (steps test3_ivl 0)
```

```
value show_acom (steps test3_ivl 1)
```

```
value show_acom (steps test3_ivl 2)
```

```
value show_acom (steps test3_ivl 3)
```

```
value show_acom (steps test3_ivl 4)
```

```
value show_acom (steps test3_ivl 5)
```

Takes as many iterations as the actual execution. Would diverge if loop did not terminate. Worse still, as the following example shows: even if the actual execution terminates, the analysis may not. The value of y keeps decreasing as the analysis is iterated, no matter how long:

```
value show_acom (steps test4_ivl 50)
```

Relationships between variables are NOT captured:

```
value show_acom_opt (AI_ivl test5_ivl)
```

Again, the analysis would not terminate:

```
value show_acom (steps test6_ivl 50)
```

```
end
```

### 14.13 Widening and Narrowing

```
theory Abs_Int3
```

```
imports Abs_Int2_ivl
```

```
begin
```

```
class widen =
```

```
fixes widen :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a (infix  $\nabla$  65)
```

```
class narrow =
```

```
fixes narrow :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a (infix  $\Delta$  65)
```

```

class wn = widen + narrow + order +
assumes widen1:  $x \leq x \nabla y$ 
assumes widen2:  $y \leq x \nabla y$ 
assumes narrow1:  $y \leq x \implies y \leq x \Delta y$ 
assumes narrow2:  $y \leq x \implies x \Delta y \leq x$ 
begin

lemma narrowid[simp]:  $x \Delta x = x$ 
by (metis eq_iff narrow1 narrow2)

end

lemma top_widen_top[simp]:  $\top \nabla \top = (\top :: \{wn, order\_top\})$ 
by (metis eq_iff top_greatest widen2)

instantiation ivl :: wn
begin

definition widen_rep p1 p2 =
  (if is_empty_rep p1 then p2 else if is_empty_rep p2 then p1 else
   let (l1,h1) = p1; (l2,h2) = p2
   in (if l2 < l1 then Minf else l1, if h1 < h2 then Pinf else h1))

lift_definition widen_ivl :: ivl  $\Rightarrow$  ivl  $\Rightarrow$  ivl is widen_rep
by(auto simp: widen_rep_def eq_ivl_iff)

definition narrow_rep p1 p2 =
  (if is_empty_rep p1  $\vee$  is_empty_rep p2 then empty_rep else
   let (l1,h1) = p1; (l2,h2) = p2
   in (if l1 = Minf then l2 else l1, if h1 = Pinf then h2 else h1))

lift_definition narrow_ivl :: ivl  $\Rightarrow$  ivl  $\Rightarrow$  ivl is narrow_rep
by(auto simp: narrow_rep_def eq_ivl_iff)

instance
proof
qed (transfer, auto simp: widen_rep_def narrow_rep_def le_iff_subset  $\gamma$ _rep_def
  subset_eq is_empty_rep_def empty_rep_def eq_ivl_def split: if_splits extended.splits) +

end

instantiation st :: (order_top, wn) wn
begin

```

**lift\_definition** *widen\_st* :: 'a st  $\Rightarrow$  'a st  $\Rightarrow$  'a st **is** *map2\_st\_rep* ( $\nabla$ )  
**by**(*auto simp: eq\_st\_def*)

**lift\_definition** *narrow\_st* :: 'a st  $\Rightarrow$  'a st  $\Rightarrow$  'a st **is** *map2\_st\_rep* ( $\Delta$ )  
**by**(*auto simp: eq\_st\_def*)

**instance**

**proof** (*standard, goal\_cases*)

**case** 1 **thus** ?*case* **by** *transfer* (*simp add: less\_eq\_st\_rep\_iff widen1*)  
**next**

**case** 2 **thus** ?*case* **by** *transfer* (*simp add: less\_eq\_st\_rep\_iff widen2*)  
**next**

**case** 3 **thus** ?*case* **by** *transfer* (*simp add: less\_eq\_st\_rep\_iff narrow1*)  
**next**

**case** 4 **thus** ?*case* **by** *transfer* (*simp add: less\_eq\_st\_rep\_iff narrow2*)  
**qed**

**end**

**instantiation** *option* :: (wn)wn

**begin**

**fun** *widen\_option* **where**

*None*  $\nabla$  *x* = *x* |

*x*  $\nabla$  *None* = *x* |

(*Some* *x*)  $\nabla$  (*Some* *y*) = *Some*(*x*  $\nabla$  *y*)

**fun** *narrow\_option* **where**

*None*  $\Delta$  *x* = *None* |

*x*  $\Delta$  *None* = *None* |

(*Some* *x*)  $\Delta$  (*Some* *y*) = *Some*(*x*  $\Delta$  *y*)

**instance**

**proof** (*standard, goal\_cases*)

**case** (1 *x y*) **thus** ?*case*

**by**(*induct x y rule: widen\_option.induct*)(*simp\_all add: widen1*)

**next**

**case** (2 *x y*) **thus** ?*case*

**by**(*induct x y rule: widen\_option.induct*)(*simp\_all add: widen2*)

**next**

**case** (3 *x y*) **thus** ?*case*

**by**(*induct x y rule: narrow\_option.induct*) (*simp\_all add: narrow1*)

**next**

**case** ( $\lambda y x$ ) **thus** ?case  
**by**(*induct*  $x y$  *rule: narrow\_option.induct*) (*simp\_all* *add: narrow2*)  
**qed**

**end**

**definition** *map2\_acom* :: ( $'a \Rightarrow 'a \Rightarrow 'a$ )  $\Rightarrow 'a \text{ acom} \Rightarrow 'a \text{ acom} \Rightarrow 'a \text{ acom}$   
**where**  
*map2\_acom*  $f C1 C2 = \text{annotate } (\lambda p. f (\text{anno } C1 p) (\text{anno } C2 p)) (\text{strip } C1)$

**instantiation** *acom* :: (*widen*)*widen*

**begin**

**definition** *widen\_acom* = *map2\_acom* ( $\nabla$ )

**instance** ..

**end**

**instantiation** *acom* :: (*narrow*)*narrow*

**begin**

**definition** *narrow\_acom* = *map2\_acom* ( $\Delta$ )

**instance** ..

**end**

**lemma** *strip\_map2\_acom*[*simp*]:

*strip*  $C1 = \text{strip } C2 \implies \text{strip}(\text{map2\_acom } f C1 C2) = \text{strip } C1$

**by**(*simp* *add: map2\_acom\_def*)

**lemma** *strip\_widen\_acom*[*simp*]:

*strip*  $C1 = \text{strip } C2 \implies \text{strip}(C1 \nabla C2) = \text{strip } C1$

**by**(*simp* *add: widen\_acom\_def*)

**lemma** *strip\_narrow\_acom*[*simp*]:

*strip*  $C1 = \text{strip } C2 \implies \text{strip}(C1 \Delta C2) = \text{strip } C1$

**by**(*simp* *add: narrow\_acom\_def*)

**lemma** *narrow1\_acom*:  $C2 \leq C1 \implies C2 \leq C1 \Delta (C2::'a::\text{wn } \text{acom})$

**by**(*simp* *add: narrow\_acom\_def narrow1 map2\_acom\_def less\_eq\_acom\_def size\_annos*)

**lemma** *narrow2\_acom*:  $C2 \leq C1 \implies C1 \Delta (C2::'a::\text{wn } \text{acom}) \leq C1$

**by**(*simp* *add: narrow\_acom\_def narrow2 map2\_acom\_def less\_eq\_acom\_def size\_annos*)

### 14.13.1 Pre-fixpoint computation

**definition**  $iter\_widen :: ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow ('a::\{order,widen\})option$   
**where**  $iter\_widen\ f = while\_option\ (\lambda x. \neg f\ x \leq x)\ (\lambda x. x \nabla f\ x)$

**definition**  $iter\_narrow :: ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow ('a::\{order,narrow\})option$   
**where**  $iter\_narrow\ f = while\_option\ (\lambda x. x \triangle f\ x < x)\ (\lambda x. x \triangle f\ x)$

**definition**  $pf\_wn :: ('a::\{order,widen,narrow\} \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ option$   
**where**  $pf\_wn\ f\ x =$   
*(case iter\_widen f x of None  $\Rightarrow$  None | Some p  $\Rightarrow$  iter\_narrow f p)*

**lemma**  $iter\_widen\_pf\_p$ :  $iter\_widen\ f\ x = Some\ p \Longrightarrow f\ p \leq p$   
**by** *(auto simp add: iter\_widen\_def dest: while\_option\_stop)*

**lemma**  $iter\_widen\_inv$ :  
**assumes**  $!!x. P\ x \Longrightarrow P(f\ x)\ !!x1\ x2. P\ x1 \Longrightarrow P\ x2 \Longrightarrow P(x1 \nabla x2)$  **and**  
 $P\ x$   
**and**  $iter\_widen\ f\ x = Some\ y$  **shows**  $P\ y$   
**using**  $while\_option\_rule$  **where**  $P = P, OF\_assms(4)[unfolding\ iter\_widen\_def]$   
**by** *(blast intro: assms(1-3))*

**lemma**  $strip\_while$ : **fixes**  $f :: 'a\ acom \Rightarrow 'a\ acom$   
**assumes**  $\forall C. strip\ (f\ C) = strip\ C$  **and**  $while\_option\ P\ f\ C = Some\ C'$   
**shows**  $strip\ C' = strip\ C$   
**using**  $while\_option\_rule$  **where**  $P = \lambda C'. strip\ C' = strip\ C, OF\_assms(2)$   
**by** *(metis assms(1))*

**lemma**  $strip\_iter\_widen$ : **fixes**  $f :: 'a::\{order,widen\}\ acom \Rightarrow 'a\ acom$   
**assumes**  $\forall C. strip\ (f\ C) = strip\ C$  **and**  $iter\_widen\ f\ C = Some\ C'$   
**shows**  $strip\ C' = strip\ C$   
**proof**–  
**have**  $\forall C. strip(C \nabla f\ C) = strip\ C$   
**by** *(metis assms(1) strip\_map2\_acom widen\_acom\_def)*  
**from**  $strip\_while$  *[OF this] assms(2)* **show**  $?thesis$  **by** *(simp add: iter\_widen\_def)*  
**qed**

**lemma**  $iter\_narrow\_pf\_p$ :  
**assumes**  $mono: !!x1\ x2:::wn\ acom. P\ x1 \Longrightarrow P\ x2 \Longrightarrow x1 \leq x2 \Longrightarrow f\ x1 \leq f\ x2$   
**and**  $Pinv: !!x. P\ x \Longrightarrow P(f\ x)\ !!x1\ x2. P\ x1 \Longrightarrow P\ x2 \Longrightarrow P(x1 \triangle x2)$   
**and**  $P\ p0$  **and**  $f\ p0 \leq p0$  **and**  $iter\_narrow\ f\ p0 = Some\ p$   
**shows**  $P\ p \wedge f\ p \leq p$



**proof**–  
**let**  $?Q = \%p. P p \wedge f p \leq p \wedge p \leq p0$   
**have**  $?Q (p \triangle f p)$  **if**  $Q: ?Q p$  **for**  $p$   
**proof** *auto*  
**note**  $P = \text{conjunction1}[OF Q]$  **and**  $12 = \text{conjunction2}[OF Q]$   
**note**  $1 = \text{conjunction1}[OF 12]$  **and**  $2 = \text{conjunction2}[OF 12]$   
**let**  $?p' = p \triangle f p$   
**show**  $P ?p'$  **by** (*blast intro: P Pinv*)  
**have**  $f ?p' \leq f p$  **by**(*rule mono[OF ⟨P (p Δ f p)⟩ P narrow2\_acom[OF 1]]*)  
**also have**  $\dots \leq ?p'$  **by**(*rule narrow1\_acom[OF 1]*)  
**finally show**  $f ?p' \leq ?p'$  .  
**have**  $?p' \leq p$  **by** (*rule narrow2\_acom[OF 1]*)  
**also have**  $p \leq p0$  **by**(*rule 2*)  
**finally show**  $?p' \leq p0$  .  
**qed**  
**thus** *?thesis*  
**using** *while\_option\_rule[where P = ?Q, OF \_ assms(6)[simplified\_iter\_narrow\_def]]*  
**by** (*blast intro: assms(4,5) le\_refl*)  
**qed**

**lemma** *pf<sub>p</sub>-wn-pf<sub>p</sub>*:  
**assumes** *mono: !!x1 x2:::wn acom. P x1  $\implies$  P x2  $\implies$  x1  $\leq$  x2  $\implies$  f x1  $\leq$  f x2*  
**and** *Pinv: P x !!x. P x  $\implies$  P(f x)*  
*!!x1 x2. P x1  $\implies$  P x2  $\implies$  P(x1  $\nabla$  x2)*  
*!!x1 x2. P x1  $\implies$  P x2  $\implies$  P(x1  $\triangle$  x2)*  
**and** *pf<sub>p</sub>-wn: pf<sub>p</sub>-wn f x = Some p shows P p  $\wedge$  f p  $\leq$  p*  
**proof**–  
**from** *pf<sub>p</sub>-wn* **obtain**  $p0$   
**where** *its: iter\_widen f x = Some p0 iter\_narrow f p0 = Some p*  
**by**(*auto simp: pf<sub>p</sub>-wn\_def split: option.splits*)  
**have**  $P p0$  **by** (*blast intro: iter\_widen\_inv[where P=P] its(1) Pinv(1-3)*)  
**thus** *?thesis*  
**by** – (*assumption |*  
*rule iter\_narrow\_pf<sub>p</sub>[where P=P] mono Pinv(2,4) iter\_widen\_pf<sub>p</sub>*  
*its)+*  
**qed**

**lemma** *strip\_pf<sub>p</sub>-wn*:  
 $\llbracket \forall C. \text{strip}(f C) = \text{strip } C; \text{pf}_p\text{-wn } f C = \text{Some } C' \rrbracket \implies \text{strip } C' = \text{strip } C$   
**by**(*auto simp add: pf<sub>p</sub>-wn\_def iter\_narrow\_def split: option.splits*)  
(*metis (mono\_tags) strip\_iter\_widen strip\_narrow\_acom strip\_while*)

```

locale Abs_Int_wn = Abs_Int_inv_mono where  $\gamma = \gamma$ 
  for  $\gamma :: 'av :: \{wn, bounded\_lattice\} \Rightarrow val\ set$ 
begin

definition AI_wn ::  $com \Rightarrow 'av\ st\ option\ acom\ option$  where
  AI_wn c = pfp_wn (step'  $\top$ ) (bot c)

lemma AI_wn_correct: AI_wn c = Some C  $\implies CS\ c \leq \gamma_c\ C$ 
proof(simp add: CS_def AI_wn_def)
  assume 1: pfp_wn (step'  $\top$ ) (bot c) = Some C
  have 2: strip C = c  $\wedge$  step'  $\top$  C  $\leq$  C
    by(rule pfp_wn_pfp[where x=bot c]) (simp_all add: 1 mono_step'_top)
  have pfp: step ( $\gamma_o\ \top$ ) ( $\gamma_c\ C$ )  $\leq \gamma_c\ C$ 
proof(rule order_trans)
  show step ( $\gamma_o\ \top$ ) ( $\gamma_c\ C$ )  $\leq \gamma_c$  (step'  $\top$  C)
    by(rule step_step')
  show ...  $\leq \gamma_c\ C$ 
    by(rule mono_gamma_c[OF conjunct2[OF 2]])
qed
  have 3: strip ( $\gamma_c\ C$ ) = c by(simp add: strip_pfp_wn[OF _ 1])
  have lfp c (step ( $\gamma_o\ \top$ ))  $\leq \gamma_c\ C$ 
    by(rule lfp_lowerbound[simplified,where f=step ( $\gamma_o\ \top$ ), OF 3 pfp])
  thus lfp c (step UNIV)  $\leq \gamma_c\ C$  by simp
qed

end

global interpretation Abs_Int_wn
where  $\gamma = \gamma_{ivl}$  and num' = num_ivl and plus' = (+)
and test_num' = in_ivl
and inv_plus' = inv_plus_ivl and inv_less' = inv_less_ivl
defines AI_wn_ivl = AI_wn
..

```

### 14.13.2 Tests

```

definition step_up_ivl n = (( $\lambda C. C \nabla step\_ivl\ \top\ C$ )  $\hat{\ }^n$ )
definition step_down_ivl n = (( $\lambda C. C \Delta step\_ivl\ \top\ C$ )  $\hat{\ }^n$ )

```

For *test3\_ivl*, *AI\_ivl* needed as many iterations as the loop took to execute. In contrast, *AI\_wn\_ivl* converges in a constant number of steps:

```

value show_acom (step_up_ivl 1 (bot test3_ivl))

```

```

value show_acom (step_up_ivl 2 (bot test3_ivl))
value show_acom (step_up_ivl 3 (bot test3_ivl))
value show_acom (step_up_ivl 4 (bot test3_ivl))
value show_acom (step_up_ivl 5 (bot test3_ivl))
value show_acom (step_up_ivl 6 (bot test3_ivl))
value show_acom (step_up_ivl 7 (bot test3_ivl))
value show_acom (step_up_ivl 8 (bot test3_ivl))
value show_acom (step_down_ivl 1 (step_up_ivl 8 (bot test3_ivl)))
value show_acom (step_down_ivl 2 (step_up_ivl 8 (bot test3_ivl)))
value show_acom (step_down_ivl 3 (step_up_ivl 8 (bot test3_ivl)))
value show_acom (step_down_ivl 4 (step_up_ivl 8 (bot test3_ivl)))
value show_acom_opt (AI_wn_ivl test3_ivl)

```

Now all the analyses terminate:

```

value show_acom_opt (AI_wn_ivl test4_ivl)
value show_acom_opt (AI_wn_ivl test5_ivl)
value show_acom_opt (AI_wn_ivl test6_ivl)

```

### 14.13.3 Generic Termination Proof

**lemma** *top\_on\_opt\_widen*:

$top\_on\_opt\ o1\ X \implies top\_on\_opt\ o2\ X \implies top\_on\_opt\ (o1 \nabla o2 :: \_ st\ option)\ X$

**apply** (*induct o1 o2 rule: widen\_option.induct*)

**apply** (*auto*)

**by** *transfer simp*

**lemma** *top\_on\_opt\_narrow*:

$top\_on\_opt\ o1\ X \implies top\_on\_opt\ o2\ X \implies top\_on\_opt\ (o1 \triangle o2 :: \_ st\ option)\ X$

**apply** (*induct o1 o2 rule: narrow\_option.induct*)

**apply** (*auto*)

**by** *transfer simp*

**lemma** *annos\_map2\_acom[simp]*:  $strip\ C2 = strip\ C1 \implies$

$annos(map2\_acom\ f\ C1\ C2) = map\ (\% (x,y).f\ x\ y)\ (zip\ (annos\ C1)\ (annos\ C2))$

**by** (*simp add: map2\_acom\_def list\_eq\_iff\_nth\_eq size\_annos anno\_def[symmetric] size\_annos\_same[of C1 C2]*)

**lemma** *top\_on\_acom\_widen*:

$\llbracket top\_on\_acom\ C1\ X; strip\ C1 = strip\ C2; top\_on\_acom\ C2\ X \rrbracket$

$\implies top\_on\_acom\ (C1 \nabla C2 :: \_ st\ option\ acom)\ X$

**by**(*auto simp add: widen\_acom\_def top\_on\_acom\_def*)(*metis top\_on\_opt\_widen in\_set\_zipE*)

**lemma** *top\_on\_acom\_narrow*:

$\llbracket \text{top\_on\_acom } C1 \ X; \text{ strip } C1 = \text{ strip } C2; \text{ top\_on\_acom } C2 \ X \rrbracket$   
 $\implies \text{top\_on\_acom } (C1 \ \Delta \ C2 \ :: \ \_ \ \text{st option acom}) \ X$

**by**(*auto simp add: narrow\_acom\_def top\_on\_acom\_def*)(*metis top\_on\_opt\_narrow in\_set\_zipE*)

The assumptions for widening and narrowing differ because during narrowing we have the invariant  $y \leq x$  (where  $y$  is the next iterate), but during widening there is no such invariant, there we only have that not yet  $y \leq x$ . This complicates the termination proof for widening.

**locale** *Measure\_wn = Measure1* **where**  $m=m$

**for**  $m :: 'av :: \{order\_top, wn\} \Rightarrow nat +$

**fixes**  $n :: 'av \Rightarrow nat$

**assumes** *m\_anti\_mono*:  $x \leq y \implies m \ x \geq m \ y$

**assumes** *m\_widen*:  $\sim y \leq x \implies m(x \ \nabla \ y) < m \ x$

**assumes** *n\_narrow*:  $y \leq x \implies x \ \Delta \ y < x \implies n(x \ \Delta \ y) < n \ x$

**begin**

**lemma** *m\_s\_anti\_mono\_rep*: **assumes**  $\forall x. S1 \ x \leq S2 \ x$

**shows**  $(\sum x \in X. m \ (S2 \ x)) \leq (\sum x \in X. m \ (S1 \ x))$

**proof**–

**from** *assms* **have**  $\forall x. m(S1 \ x) \geq m(S2 \ x)$  **by** (*metis m\_anti\_mono*)

**thus**  $(\sum x \in X. m \ (S2 \ x)) \leq (\sum x \in X. m \ (S1 \ x))$  **by** (*metis sum\_mono*)

**qed**

**lemma** *m\_s\_anti\_mono*:  $S1 \leq S2 \implies m\_s \ S1 \ X \geq m\_s \ S2 \ X$

**unfolding** *m\_s\_def*

**apply** (*transfer fixing: m*)

**apply**(*simp add: less\_eq\_st\_rep\_iff eq\_st\_def m\_s\_anti\_mono\_rep*)

**done**

**lemma** *m\_s\_widen\_rep*: **assumes** *finite X S1 = S2 on -X*  $\neg S2 \ x \leq S1 \ x$

**shows**  $(\sum x \in X. m \ (S1 \ x \ \nabla \ S2 \ x)) < (\sum x \in X. m \ (S1 \ x))$

**proof**–

**have**  $1: \forall x \in X. m(S1 \ x) \geq m(S1 \ x \ \nabla \ S2 \ x)$

**by** (*metis m\_anti\_mono wn\_class.widen1*)

**have**  $x \in X$  **using** *assms*(2,3)

**by**(*auto simp add: Ball\_def*)

**hence**  $2: \exists x \in X. m(S1 \ x) > m(S1 \ x \ \nabla \ S2 \ x)$

**using** *assms*(3) *m\_widen* **by** *blast*

```

from sum_strict_mono_ex1 [OF ⟨finite X⟩ 1 2]
show ?thesis .
qed

```

```

lemma m_s_widen: finite X  $\implies$  fun S1 = fun S2 on -X  $\implies$ 
   $\sim S2 \leq S1 \implies m_s (S1 \nabla S2) X < m_s S1 X$ 
apply (auto simp add: less_st_def m_s_def)
apply (transfer fixing: m)
apply (auto simp add: less_eq_st_rep_iff m_s_widen_rep)
done

```

```

lemma m_o_anti_mono: finite X  $\implies$  top_on_opt o1 (-X)  $\implies$  top_on_opt
o2 (-X)  $\implies$ 
  o1 \leq o2 \implies m_o o1 X \geq m_o o2 X
proof (induction o1 o2 rule: less_eq_option.induct)
  case 1 thus ?case by (simp add: m_o_def)(metis m_s_anti_mono)
next
  case 2 thus ?case
    by (simp add: m_o_def le_SucI m_s_h split: option.splits)
next
  case 3 thus ?case by simp
qed

```

```

lemma m_o_widen:  $\llbracket$  finite X; top_on_opt S1 (-X); top_on_opt S2 (-X);
 $\neg S2 \leq S1 \rrbracket \implies$ 
   $m_o (S1 \nabla S2) X < m_o S1 X$ 
by (auto simp: m_o_def m_s_h less_Suc_eq_le m_s_widen split: option.split)

```

```

lemma m_c_widen:
  strip C1 = strip C2  $\implies$  top_on_acom C1 (-vars C1)  $\implies$  top_on_acom
C2 (-vars C2)
   $\implies \neg C2 \leq C1 \implies m_c (C1 \nabla C2) < m_c C1$ 
apply (auto simp: m_c_def widen_acom_def map2_acom_def size_annos[symmetric]
anno_def[symmetric] sum_list_sum_nth)
apply (subgoal_tac length(annos C2) = length(annos C1))
  prefer 2 apply (simp add: size_annos_same2)
apply (auto)
apply (rule sum_strict_mono_ex1)
  apply (auto simp add: m_o_anti_mono vars_acom_def anno_def top_on_acom_def
top_on_opt_widen widen1 less_eq_acom_def listrel_iff_nth)
apply (rule_tac x=p in bexI)
  apply (auto simp: vars_acom_def m_o_widen top_on_acom_def)
done

```

**definition**  $n_s$  :: 'av st  $\Rightarrow$  vname set  $\Rightarrow$  nat ( $n_s$ ) **where**  
 $n_s$  S X = ( $\sum x \in X. n(\text{fun } S \ x)$ )

**lemma**  $n_s$ \_narrow\_rep:

**assumes** finite X S1 = S2 on -X  $\forall x. S2 \ x \leq S1 \ x \ \forall x. S1 \ x \ \Delta \ S2 \ x \leq S1 \ x$

$S1 \ x \neq S1 \ x \ \Delta \ S2 \ x$

**shows** ( $\sum x \in X. n(S1 \ x \ \Delta \ S2 \ x)$ ) < ( $\sum x \in X. n(S1 \ x)$ )

**proof**–

**have** 1:  $\forall x. n(S1 \ x \ \Delta \ S2 \ x) \leq n(S1 \ x)$

**by** (metis assms(3) assms(4) eq\_iff\_less\_le\_not\_le n\_narrow)

**have**  $x \in X$  **by** (metis Compl\_iff assms(2) assms(5) narrowid)

**hence** 2:  $\exists x \in X. n(S1 \ x \ \Delta \ S2 \ x) < n(S1 \ x)$

**by** (metis assms(3–5) eq\_iff\_less\_le\_not\_le n\_narrow)

**show** ?thesis

**apply**(rule sum\_strict\_mono\_ex1[OF ⟨finite X⟩]) **using** 1 2 **by** blast+  
**qed**

**lemma**  $n_s$ \_narrow: finite X  $\implies$  fun S1 = fun S2 on -X  $\implies$  S2  $\leq$  S1  
 $\implies$  S1  $\Delta$  S2 < S1

$\implies n_s(S1 \ \Delta \ S2) \ X < n_s \ S1 \ X$

**apply**(auto simp add: less\_st\_def  $n_s$ \_def)

**apply** (transfer fixing: n)

**apply**(auto simp add: less\_eq\_st\_rep\_iff eq\_st\_def fun\_eq\_iff  $n_s$ \_narrow\_rep)

**done**

**definition**  $n_o$  :: 'av st option  $\Rightarrow$  vname set  $\Rightarrow$  nat ( $n_o$ ) **where**  
 $n_o$  opt X = (case opt of None  $\Rightarrow$  0 | Some S  $\Rightarrow$   $n_s$  S X + 1)

**lemma**  $n_o$ \_narrow:

top\_on\_opt S1 (-X)  $\implies$  top\_on\_opt S2 (-X)  $\implies$  finite X

$\implies S2 \leq S1 \implies S1 \ \Delta \ S2 < S1 \implies n_o(S1 \ \Delta \ S2) \ X < n_o \ S1 \ X$

**apply**(induction S1 S2 rule: narrow\_option.induct)

**apply**(auto simp:  $n_o$ \_def  $n_s$ \_narrow)

**done**

**definition**  $n_c$  :: 'av st option acom  $\Rightarrow$  nat ( $n_c$ ) **where**  
 $n_c$  C = sum\_list (map ( $\lambda a. n_o \ a \ (\text{vars } C)$ )) (annos C)

**lemma** less\_annos\_iff: (C1 < C2) = (C1  $\leq$  C2  $\wedge$

( $\exists i < \text{length}(\text{annos } C1). \text{annos } C1 \ ! \ i < \text{annos } C2 \ ! \ i$ )

**by**(metis (hide\_lams, no\_types) less\_le\_not\_le le\_iff\_le\_annos\_size\_annos\_same2)

```

lemma n_c_narrow: strip C1 = strip C2
   $\implies$  top_on_acom C1 (- vars C1)  $\implies$  top_on_acom C2 (- vars C2)
   $\implies$  C2  $\leq$  C1  $\implies$  C1  $\Delta$  C2 < C1  $\implies$  n_c (C1  $\Delta$  C2) < n_c C1
apply(auto simp: n_c_def narrow_acom_def sum_list_sum_nth)
apply(subgoal_tac length(annos C2) = length(annos C1))
prefer 2 apply (simp add: size_annos_same2)
apply (auto)
apply(simp add: less_annos_iff le_iff_le_annos)
apply(rule sum_strict_mono_ex1)
apply (auto simp: vars_acom_def top_on_acom_def)
apply (metis n_o_narrow nth_mem finite_cvars less_imp_le le_less order_refl)
apply(rule_tac x=i in bexI)
prefer 2 apply simp
apply(rule n_o_narrow[where X = vars(strip C2)])
apply (simp-all)
done

```

**end**

```

lemma iter_widen_termination:
fixes m :: 'a::wn acom  $\Rightarrow$  nat
assumes P_f:  $\bigwedge C. P C \implies P(f C)$ 
and P_widen:  $\bigwedge C1 C2. P C1 \implies P C2 \implies P(C1 \nabla C2)$ 
and m_widen:  $\bigwedge C1 C2. P C1 \implies P C2 \implies \sim C2 \leq C1 \implies m(C1 \nabla C2) < m C1$ 
and P C shows  $\exists C'. iter_widen f C = Some C'$ 
proof(simp add: iter_widen_def,
  rule measure_while_option_Some[where P = P and f=m])
  show P C by(rule  $\langle P C \rangle$ )
next
  fix C assume P C  $\neg$  f C  $\leq$  C thus P (C  $\nabla$  f C)  $\wedge$  m (C  $\nabla$  f C) < m C
  by(simp add: P_f P_widen m_widen)
qed

```

```

lemma iter_narrow_termination:
fixes n :: 'a::wn acom  $\Rightarrow$  nat
assumes P_f:  $\bigwedge C. P C \implies P(f C)$ 
and P_narrow:  $\bigwedge C1 C2. P C1 \implies P C2 \implies P(C1 \Delta C2)$ 
and mono:  $\bigwedge C1 C2. P C1 \implies P C2 \implies C1 \leq C2 \implies f C1 \leq f C2$ 
and n_narrow:  $\bigwedge C1 C2. P C1 \implies P C2 \implies C2 \leq C1 \implies C1 \Delta C2 < C1 \implies n(C1 \Delta C2) < n C1$ 

```

**and** *init*:  $P \ C \ f \ C \leq C$  **shows**  $\exists C'. \text{iter\_narrow } f \ C = \text{Some } C'$   
**proof**(*simp add*: *iter\_narrow\_def*,  
*rule measure\_while\_option\_Some*[**where**  $f=n$  **and**  $P = \%C. P \ C \wedge f$   
 $C \leq C$ ])  
**show**  $P \ C \wedge f \ C \leq C$  **using** *init* **by** *blast*  
**next**  
**fix**  $C$  **assume**  $1: P \ C \wedge f \ C \leq C$  **and**  $2: C \ \Delta \ f \ C < C$   
**hence**  $P \ (C \ \Delta \ f \ C)$  **by**(*simp add*: *P-f P-narrow*)  
**moreover then have**  $f \ (C \ \Delta \ f \ C) \leq C \ \Delta \ f \ C$   
**by** (*metis narrow1\_acom narrow2\_acom 1 mono order\_trans*)  
**moreover have**  $n \ (C \ \Delta \ f \ C) < n \ C$  **using**  $1 \ 2$  **by**(*simp add*: *n\_narrow*  
*P-f*)  
**ultimately show**  $(P \ (C \ \Delta \ f \ C) \wedge f \ (C \ \Delta \ f \ C) \leq C \ \Delta \ f \ C) \wedge n(C \ \Delta$   
 $f \ C) < n \ C$   
**by** *blast*  
**qed**

**locale** *Abs\_Int\_wn\_measure* = *Abs\_Int\_wn* **where**  $\gamma=\gamma + \text{Measure\_wn}$  **where**  
 $m=m$   
**for**  $\gamma :: 'av::\{wn, bounded\_lattice\} \Rightarrow \text{val set}$  **and**  $m :: 'av \Rightarrow \text{nat}$

#### 14.13.4 Termination: Intervals

**definition**  $m\_rep :: \text{eint2} \Rightarrow \text{nat}$  **where**  
 $m\_rep \ p = (\text{if } \text{is\_empty\_rep } \ p \ \text{then } 3 \ \text{else}$   
 $\text{let } (l,h) = p \ \text{in } (\text{case } l \ \text{of } \text{Minf} \Rightarrow 0 \mid \_ \Rightarrow 1) + (\text{case } h \ \text{of } \text{Pinf} \Rightarrow 0 \mid \_$   
 $\Rightarrow 1))$

**lift\_definition**  $m\_ivl :: \text{ivl} \Rightarrow \text{nat}$  **is**  $m\_rep$   
**by**(*auto simp*: *m\_rep\_def eq\_ivl\_iff*)

**lemma**  $m\_ivl\_nice$ :  $m\_ivl[l,h] = (\text{if } [l,h] = \perp \ \text{then } 3 \ \text{else}$   
 $(\text{if } l = \text{Minf} \ \text{then } 0 \ \text{else } 1) + (\text{if } h = \text{Pinf} \ \text{then } 0 \ \text{else } 1))$

**unfolding** *bot\_ivl\_def*  
**by** *transfer* (*auto simp*: *m\_rep\_def eq\_ivl\_empty split: extended.split*)

**lemma**  $m\_ivl\_height$ :  $m\_ivl \ iv \leq 3$   
**by** *transfer* (*simp add*: *m\_rep\_def split: prod.split extended.split*)

**lemma**  $m\_ivl\_anti\_mono$ :  $y \leq x \Longrightarrow m\_ivl \ x \leq m\_ivl \ y$   
**by** *transfer*  
(*auto simp*: *m\_rep\_def is\_empty\_rep\_def \gamma\_rep\_cases le\_iff\_subset*  
*split: prod.split extended.splits if\_splits*)



**lemma** *m\_ivl\_widen*:  
 $\sim y \leq x \implies m\_ivl(x \nabla y) < m\_ivl x$   
**by** *transfer*  
*(auto simp: m\_rep\_def widen\_rep\_def is\_empty\_rep\_def  $\gamma$ \_rep\_cases le\_iff\_subset split: prod.split extended.splits if\_splits)*

**definition** *n\_ivl* :: *ivl*  $\Rightarrow$  *nat* **where**  
*n\_ivl iv* =  $3 - m\_ivl iv$

**lemma** *n\_ivl\_narrow*:  
 $x \Delta y < x \implies n\_ivl(x \Delta y) < n\_ivl x$   
**unfolding** *n\_ivl\_def*  
**apply** (*subst (asm) less\_le\_not\_le*)  
**apply** *transfer*  
**by** (*auto simp add: m\_rep\_def narrow\_rep\_def is\_empty\_rep\_def empty\_rep\_def  $\gamma$ \_rep\_cases le\_iff\_subset split: prod.splits if\_splits extended.split*)

**global interpretation** *Abs\_Int\_wn\_measure*  
**where**  $\gamma = \gamma\_ivl$  **and**  $num' = num\_ivl$  **and**  $plus' = (+)$   
**and**  $test\_num' = in\_ivl$   
**and**  $inv\_plus' = inv\_plus\_ivl$  **and**  $inv\_less' = inv\_less\_ivl$   
**and**  $m = m\_ivl$  **and**  $n = n\_ivl$  **and**  $h = 3$   
**proof** (*standard, goal\_cases*)  
**case** 2 **thus** ?*case* **by** (*rule m\_ivl\_anti\_mono*)  
**next**  
**case** 1 **thus** ?*case* **by** (*rule m\_ivl\_height*)  
**next**  
**case** 3 **thus** ?*case* **by** (*rule m\_ivl\_widen*)  
**next**  
**case** 4 **from** 4(2) **show** ?*case* **by** (*rule n\_ivl\_narrow*)  
— note that the first assms is unnecessary for intervals  
**qed**

**lemma** *iter\_widen\_step\_ivl\_termination*:  
 $\exists C. iter\_widen (step\_ivl \top) (bot c) = Some C$   
**apply** (*rule iter\_widen\_termination[where m = m\_c and P = %C. strip C = c  $\wedge$  top\_on\_acom C (- vars C)]*)  
**apply** (*auto simp add: m\_c\_widen top\_on\_bot top\_on\_step'[simplified comp\_def vars\_acom\_def] vars\_acom\_def top\_on\_acom\_widen*)  
**done**

```

lemma iter_narrow_step_ivl_termination:
  top_on_acom C (- vars C)  $\implies$  step_ivl  $\top$  C  $\leq$  C  $\implies$ 
   $\exists$  C'. iter_narrow (step_ivl  $\top$ ) C = Some C'
apply(rule iter_narrow_termination[where n = n_c and P = %C'. strip
C = strip C'  $\wedge$  top_on_acom C' (-vars C')])
apply(auto simp: top_on_step'[simplified comp_def vars_acom_def]
mono_step'_top n_c_narrow vars_acom_def top_on_acom_narrow)
done

theorem AI_wn_ivl_termination:
   $\exists$  C. AI_wn_ivl c = Some C
apply(auto simp: AI_wn_def pfp_wn_def iter_widen_step_ivl_termination
split: option.split)
apply(rule iter_narrow_step_ivl_termination)
apply(rule conjunct2)
apply(rule iter_widen_inv[where f = step'  $\top$  and P = %C. c = strip C
& top_on_acom C (- vars C)])
apply(auto simp: top_on_acom_widen top_on_step'[simplified comp_def vars_acom_def]
iter_widen_pfp top_on_bot vars_acom_def)
done

```

### 14.13.5 Counterexamples

Widening is increasing by assumption, but  $x \leq f x$  is not an invariant of widening. It can already be lost after the first step:

```

lemma assumes !!x y::'a::wn. x  $\leq$  y  $\implies$  f x  $\leq$  f y
and x  $\leq$  f x and  $\neg$  f x  $\leq$  x shows x  $\nabla$  f x  $\leq$  f(x  $\nabla$  f x)
nitpick[card = 3, expect = genuine, show_consts, timeout = 120]

```

**oops**

Widening terminates but may converge more slowly than Kleene iteration. In the following model, Kleene iteration goes from 0 to the least pfp in one step but widening takes 2 steps to reach a strictly larger pfp:

```

lemma assumes !!x y::'a::wn. x  $\leq$  y  $\implies$  f x  $\leq$  f y
and x  $\leq$  f x and  $\neg$  f x  $\leq$  x and f(f x)  $\leq$  f x
shows f(x  $\nabla$  f x)  $\leq$  x  $\nabla$  f x
nitpick[card = 4, expect = genuine, show_consts, timeout = 120]

```

**oops**

**end**

## References

- [1] T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *Lect. Notes in Comp. Sci.*, pages 180–192. Springer-Verlag, 1996.
- [2] T. Nipkow and G. Klein. *Concrete Semantics with Isabelle/HOL*. Springer, 2014. <http://concrete-semantics.org>.