

Concrete Semantics

Tobias Nipkow & Gerwin Klein

October 16, 2017

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

1	Arithmetic and Boolean Expressions	4
1.1	Arithmetic Expressions	4
1.2	Constant Folding	5
1.3	Boolean Expressions	6
1.4	Constant Folding	6
2	Stack Machine and Compilation	7
2.1	Stack Machine	7
2.2	Compilation	8
3	IMP — A Simple Imperative Language	9
3.1	Big-Step Semantics of Commands	9
3.2	Rule inversion	11
3.3	Command Equivalence	13
3.4	Execution is deterministic	15
4	Small-Step Semantics of Commands	16
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

5	Compiler for IMP	20
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
6	Compiler Correctness, Reverse Direction	26
6.1	Definitions	26
6.2	Basic properties of <i>exec_n</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
7	A Typed Language	40
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
8	Security Type Systems	44
8.1	Security Levels and Expressions	45
8.2	Syntax Directed Typing	46
8.3	The Standard Typing System	50
8.4	A Bottom-Up Typing System	51
8.5	A Termination-Sensitive Syntax Directed System	52
8.6	The Standard Termination-Sensitive System	55
9	Definite Initialization Analysis	56
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
10	Constant Folding	64
10.1	Semantic Equivalence up to a Condition	64
10.2	Simple folding of arithmetic expressions	68

11 Live Variable Analysis	72
11.1 Liveness Analysis	72
11.2 Correctness	73
11.3 Program Optimization	75
11.4 True Liveness Analysis	78
11.5 Correctness	79
11.6 Executability	81
11.7 Limiting the number of iterations	82
12 Denotational Semantics of Commands	83
12.1 Continuity	84
12.2 The denotational semantics is deterministic	86
13 Hoare Logic	87
13.1 Hoare Logic for Partial Correctness	87
13.2 Soundness and Completeness	90
13.3 Verification Conditions	92
13.4 Hoare Logic for Total Correctness	94
14 Abstract Interpretation	99
14.1 Annotated Commands	100
14.2 The generic Step function	103
14.3 Collecting Semantics of Commands	104
14.4 Pretty printing state sets	109
14.5 Examples	109
14.6 Test Programs	110
14.7 Orderings	112
14.8 Abstract Interpretation	115
14.9 Computable Abstract Interpretation	127
14.10 Constant Propagation	133
14.11 Backward Analysis of Expressions	137
14.12 Interval Analysis	142
14.13 Widening and Narrowing	152

1 Arithmetic and Boolean Expressions

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

1.1 Arithmetic Expressions

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 τ_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*₁ *a*₂) *s* = *aval* *a*₁ *s* + *aval* *a*₂ *s*

apply(*induction* *a*₁ *a*₂ *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*₁ *a*₂) = *plus* (*asimp* *a*₁) (*asimp* *a*₂)

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
theory BExp imports AExp begin

```

1.3 Boolean Expressions

```

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

```

```

abbreviation hd2 xs == hd(tl xs)

```

```

abbreviation tl2 xs == tl(tl xs)

```

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

```

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

```

$exec1 \text{ ADD } _ stk = (hd2 \text{ stk} + hd \text{ stk}) \# tl2 \text{ stk}$

fun $exec :: instr \text{ list} \Rightarrow state \Rightarrow stack \Rightarrow stack$ **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 $comp :: aexp \Rightarrow instr \text{ list}$ **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 ⇒ 'a*'b ⇒ bool, split_format(complete)]

declare star.refl[simp,intro]

lemma star_step1[simp, intro]: r x y ⇒ 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 axp      (_ ::= _ [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

```

```

theory Big-Step imports Com begin

```

3.1 Big-Step Semantics of Commands

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 × state ⇒ state ⇒ bool (infix ⇒ 55)

```

```

where

```

```

  Skip: (SKIP,s) ⇒ s |

```

```

  Assign: (x ::= a,s) ⇒ s(x := aval a s) |

```

```

  Seq: [ (c1,s1) ⇒ s2; (c2,s2) ⇒ s3 ] ⇒⇒ (c1;;c2, s1) ⇒ s3 |

```

IfTrue: $\llbracket \text{bval } b \text{ } s; (c_1, s) \Rightarrow t \rrbracket \Longrightarrow (IF \text{ } b \text{ } THEN \text{ } c_1 \text{ } ELSE \text{ } c_2, s) \Rightarrow t \mid$
IfFalse: $\llbracket \neg \text{bval } b \text{ } s; (c_2, s) \Rightarrow t \rrbracket \Longrightarrow (IF \text{ } b \text{ } THEN \text{ } c_1 \text{ } ELSE \text{ } c_2, s) \Rightarrow t \mid$
WhileFalse: $\neg \text{bval } b \text{ } s \Longrightarrow (WHILE \text{ } b \text{ } DO \text{ } c, s) \Rightarrow s \mid$
WhileTrue:
 $\llbracket \text{bval } b \text{ } s_1; (c, s_1) \Rightarrow s_2; (WHILE \text{ } b \text{ } DO \text{ } c, s_2) \Rightarrow s_3 \rrbracket$
 $\Longrightarrow (WHILE \text{ } b \text{ } DO \text{ } c, s_1) \Rightarrow s_3$

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 $\{t. (SKIP, \lambda_. 0) \Rightarrow t\}$

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

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

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

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

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

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

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

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*

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

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!]: $(x ::= a, s) \Rightarrow t$

thm *AssignE*

inductive_cases *SeqE*[elim!]: $(c1;;c2,s1) \Rightarrow s3$

thm *SeqE*

inductive_cases *IfE*[elim!]: $(IF\ b\ THEN\ c1\ ELSE\ c2,s) \Rightarrow t$

thm *IfE*

inductive_cases *WhileE*[elim]: $(WHILE\ b\ DO\ c,s) \Rightarrow t$

thm *WhileE*

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

An automatic example:

lemma $(IF\ b\ THEN\ SKIP\ ELSE\ SKIP, s) \Rightarrow t \Longrightarrow t = s$

by *blast*

Rule inversion by hand via the “cases” method:

lemma *assumes* $(IF\ b\ THEN\ SKIP\ ELSE\ SKIP, s) \Rightarrow t$

shows $t = s$

proof—

from *assms* **show** *?thesis*

proof *cases* — inverting *assms*

case *IfTrue* **thm** *IfTrue*

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

next

case *IfFalse* **thus** *?thesis* **by** *blast*

qed

qed

lemma *assign_simp*:

$(x ::= a, s) \Rightarrow s' \longleftrightarrow (s' = s(x := \text{aval } a\ s))$

by *auto*

An example combining rule inversion and derivations

lemma *Seq_assoc*:

$(c1;; c2;; c3, s) \Rightarrow s' \longleftrightarrow (c1;; (c2;; c3), s) \Rightarrow s'$

proof

assume $(c1;; c2;; c3, s) \Rightarrow s'$

then obtain $s1\ s2$ **where**
 $c1: (c1, s) \Rightarrow s1$ **and**
 $c2: (c2, s1) \Rightarrow s2$ **and**
 $c3: (c3, s2) \Rightarrow s'$ **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** ~ 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

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

hence $(?iw, s) \Rightarrow t$

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
}
moreover
 — now the other direction:
{ fix $s\ t$ **assume** $(?iw, s) \Rightarrow t$
hence $(?w, s) \Rightarrow t$
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*
with $\langle (?iw, s) \Rightarrow t \rangle$ **have** $(c;; ?w, s) \Rightarrow t$ **by** *auto*
 — and for this, only the Seq-rule is applicable:
then obtain s' **where**
 $(c, s) \Rightarrow s'$ **and** $(?w, s') \Rightarrow t$ **by** *auto*
 — with this information, we can build a derivation tree for the *WHILE*

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

$(WHILE\ b\ DO\ c, s) \Rightarrow t \Longrightarrow c \sim c' \Longrightarrow (WHILE\ b\ DO\ c', s) \Rightarrow t$
apply (*induction WHILE b DO c s t arbitrary: b c rule: big_step_induct*)
apply *blast*
apply *blast*
done

lemma *sim_while_cong*: $c \sim c' \Longrightarrow WHILE\ b\ DO\ c \sim WHILE\ b\ DO\ c'$
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*: $c \sim c' \Longrightarrow c' \sim c'' \Longrightarrow c \sim c''$ **by** *auto*

3.4 Execution is deterministic

This proof is automatic.

theorem *big_step_determ*: $\llbracket (c, s) \Rightarrow t; (c, s) \Rightarrow u \rrbracket \Longrightarrow u = t$
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

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

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

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

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

assume $IHw: \wedge t'. (WHILE\ b\ DO\ c, s_1) \Rightarrow t' \Longrightarrow t' = t$

— Premise of implication:

assume $(WHILE\ b\ DO\ c, s) \Rightarrow t'$

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

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

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

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: $(x ::= a, s) \rightarrow (SKIP, s(x := aval a s))$ |

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

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

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

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

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

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

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:
  [ (c1,s1)  $\rightarrow^*$  (SKIP,s2); (c2,s2)  $\rightarrow^*$  (SKIP,s3) ]
   $\implies$  (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:
   $cs \rightarrow cs' \Longrightarrow cs' \Rightarrow t \Longrightarrow cs \Rightarrow t$ 
apply (induction arbitrary: t rule: small_step.induct)
apply auto
done

```

```

lemma small_to_big:
   $cs \rightarrow^* (SKIP, t) \Longrightarrow cs \Rightarrow t$ 
apply (induction cs (SKIP, t) rule: star.induct)
apply (auto intro: small1_big_continue)
done

```

Finally, the equivalence theorem:

```

theorem big_iff_small:
   $cs \Rightarrow t = cs \rightarrow^* (SKIP, t)$ 
by(metis big_to_small small_to_big)

```

4.5 Final configurations and infinite reductions

```

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

```

```

lemma finalD:  $final (c, s) \Longrightarrow c = SKIP$ 
apply(simp add: final_def)
apply(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*:
 $(EX\ t.\ cs \Rightarrow t) \longleftrightarrow (EX\ cs'.\ cs \rightarrow^* cs' \wedge final\ cs')$
by(*simp add: big_iff_small final_iff_SKIP*)

This is the same as saying that the absence of a big step result is equivalent with absence of a terminating small step sequence, i.e. with nontermination. Since \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*]:
 $0 \leq i \Longrightarrow$
 $(xs @ ys) !! i = (if\ i < size\ xs\ then\ xs\ !!\ i\ else\ ys\ !!\ (i - size\ xs))$
by (*induction xs arbitrary: i*) (*auto simp: algebra_simps*)

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

abbreviation (**output**)

$isize\ xs == int\ (length\ xs)$

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 $config = int \times state \times stack$

abbreviation $hd2\ xs == hd(tl\ xs)$

abbreviation $tl2\ xs == tl(tl\ xs)$

fun $iexec :: instr \Rightarrow config \Rightarrow config\ where$

$iexec\ instr\ (i,s,stk) = (case\ instr\ of$

$LOADI\ n \Rightarrow (i+1,s, n\#\ stk) \mid$

$LOAD\ x \Rightarrow (i+1,s, s\ x\ \#\ stk) \mid$

$ADD \Rightarrow (i+1,s, (hd2\ stk + hd\ stk)\ \#\ tl2\ stk) \mid$

$STORE\ x \Rightarrow (i+1,s(x := hd\ stk),tl\ stk) \mid$

$JMP\ n \Rightarrow (i+1+n,s,stk) \mid$

$JMPLESS\ n \Rightarrow (if\ hd2\ stk < hd\ stk\ then\ i+1+n\ else\ i+1,s,tl2\ stk) \mid$

$JMPGE\ n \Rightarrow (if\ hd2\ stk \geq hd\ stk\ then\ i+1+n\ else\ i+1,s,tl2\ stk))$

definition

$exec1 :: instr\ list \Rightarrow config \Rightarrow config \Rightarrow bool$

$((-/ \vdash (- \rightarrow / -)) [59,0,59] 60)$

where

$P \vdash c \rightarrow c' =$

$(\exists i\ s\ stk. c = (i,s,stk) \wedge c' = iexec(P!!i)\ (i,s,stk) \wedge 0 \leq i \wedge i < size\ P)$

lemma $exec1I$ [$intro, code_pred_intro$]:

$c' = iexec\ (P!!i)\ (i,s,stk) \Longrightarrow 0 \leq i \Longrightarrow i < size\ P$

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

by ($simp\ add: exec1_def$)

abbreviation

$exec :: instr\ list \Rightarrow config \Rightarrow config \Rightarrow bool\ ((-/ \vdash (- \rightarrow*/ -)) 50)$

where

$exec\ P \equiv star\ (exec1\ P)$

lemmas $exec_induct = star.induct$ [$of\ exec1\ P, split_format(complete)$]

code_pred *exec1* **by** (*metis exec1_def*)

values

$\{(i, \text{map } t \text{ [\"x\", \"y\"]}, \text{stk}) \mid i \ t \ \text{stk}.$
 $\text{[LOAD \"y\", STORE \"x\"]} \vdash$
 $(0, <\text{\"x\"} := 3, \text{\"y\"} := 4>, []) \rightarrow^* (i, t, \text{stk})\}$

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 *ieexec_shift* [*simp*]:

$((n+i', s', \text{stk}') = \text{ieexec } x \ (n+i, s, \text{stk})) = ((i', s', \text{stk}') = \text{ieexec } x \ (i, s, \text{stk}))$

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 $i \ i' :: \text{int}$

shows

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

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

unfolding *exec1_def*

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

lemma *exec_appendL*:

fixes $i \ i' :: \text{int}$

shows

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

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

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

$size \ P' \leq i$
 $\implies P \vdash (i - size \ P', s, stk) \rightarrow^* (j, s', stk')$
 $\implies i' = size \ P' + j$
 $\implies P' @ P \vdash (i, s, stk) \rightarrow^* (i', s', stk')$

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

$P \vdash (0, s, stk) \rightarrow^* (i', s', stk') \implies$
 $size \ P \leq i' \implies$
 $P' \vdash (i' - size \ P, s', stk') \rightarrow^* (i'', s'', stk'') \implies$
 $j'' = size \ P + i'$
 \implies
 $P @ P' \vdash (0, s, stk) \rightarrow^* (j'', s'', stk'')$

by(*metis star_trans[OF exec_appendR exec_appendL-if]*)

declare *Let_def[simp]*

5.4 Compilation

fun *acomp* :: $aexp \Rightarrow instr \ list \ \mathbf{where}$

acomp ($N \ n$) = [*LOADI* n] |

acomp ($V \ x$) = [*LOAD* x] |

acomp (*Plus* $a1 \ a2$) = *acomp* $a1$ @ *acomp* $a2$ @ [*ADD*]

lemma *acomp_correct[intro]*:

$acomp \ a \vdash (0, s, stk) \rightarrow^* (size(acomp \ a), s, aval \ a \ s \# stk)$

by (*induction a arbitrary: stk*) *fastforce*+

fun *bcomp* :: $bexp \Rightarrow bool \Rightarrow int \Rightarrow instr \ list \ \mathbf{where}$

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 ≤ n ⇒
bcomp b f n ⊢
(0,s,stk) →* (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* = \sim *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* ⇒ *instr list* **where**

ccomp SKIP = [] |

ccomp (*x ::= a*) = *acom* *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

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 Compiler Correctness, Reverse 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 $exec_exec_n$:

$P \vdash c \rightarrow^* c' \implies \exists n. P \vdash c \rightarrow \hat{n} c'$
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]: $\text{succs } [] \ n = \{\}$
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 $?isuccs = \lambda p \ P \ n \ i::\text{int}. \ 0 \leq i \wedge i < \text{size } P \wedge p \in \text{isuccs } (P!!i) \ (n+i)$

{ **fix** p **assume** $p \in \text{succs } (x\#xs) \ n$

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

unfolding *succs_def* **by** *auto*

have $p \in ?x \cup ?xs$

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

}

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

{ **fix** p **assume** $p \in ?x \vee p \in ?xs$

hence $p \in \text{succs } (x\#xs) \ n$

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

}

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*]:

$\text{inj } (op + (i::int))$

by (*metis add_minus_cancel inj_on_inverseI*)

lemma *succs_set_shift* [*simp*]:

$op + i \text{ ' succs } xs \ 0 = \text{succs } xs \ i$

by (*force simp: succs_shift [where n=i, symmetric] intro: set_eqI*)

lemma *succs_append* [*simp*]:

$\text{succs } (xs \ @ \ ys) \ n = \text{succs } xs \ n \cup \text{succs } ys \ (n + \text{size } xs)$

by (*induct xs arbitrary: n*) (*auto simp: succs_Cons algebra_simps*)

lemma *exits_append* [*simp*]:

$\text{exits } (xs \ @ \ ys) = \text{exits } xs \cup (op + (\text{size } xs)) \text{ ' exits } ys - \{0..<\text{size } xs + \text{size } ys\}$

by (*auto simp: exits_def image_set_diff*)

lemma *exits_single*:

$\text{exits } [x] = \text{isuccs } x \ 0 - \{0\}$

by (*auto simp: exits_def succs_def*)

lemma *exits_Cons*:

$\text{exits } (x \ # \ xs) = (\text{isuccs } x \ 0 - \{0\}) \cup (op + 1) \text{ ' exits } xs - \{0..<1 + \text{size } xs\}$

using *exits_append* [*of* $[x] \ xs$]

by (*simp add: exits_single*)

lemma *exits_empty* [*iff*]: $\text{exits } [] = \{\}$ **by** (*simp add: exits_def*)

lemma *exits_simps* [*simp*]:

$\text{exits } [ADD] = \{1\}$

$\text{exits } [LOADI \ v] = \{1\}$

$\text{exits } [LOAD \ x] = \{1\}$

$\text{exits } [STORE \ x] = \{1\}$

$i \neq -1 \implies \text{exits } [JMP \ i] = \{1 + i\}$

$i \neq -1 \implies \text{exits } [JMPGE \ i] = \{1 + i, 1\}$

$i \neq -1 \implies \text{exits } [JMPLESS \ i] = \{1 + i, 1\}$

by (*auto simp: exits_def*)

lemma *acomps_succs* [*simp*]:
 $\text{succs } (\text{acomps } a) \ n = \{n + 1 .. n + \text{size } (\text{acomps } a)\}$
by (*induct a arbitrary: n*) *auto*

lemma *acomps_size*:
 $(1::\text{int}) \leq \text{size } (\text{acomps } a)$
by (*induct a*) *auto*

lemma *acomps_exits* [*simp*]:
 $\text{exits } (\text{acomps } a) = \{\text{size } (\text{acomps } a)\}$
by (*auto simp: exits_def acomps_size*)

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

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

qed *auto*

lemmas *bcomps_succsD* [*dest!*] = *bcomps_succs* [*THEN subsetD, rotated*]

lemma *bcomps_exits*:
fixes $i :: \text{int}$
shows
 $0 \leq i \implies$
 $\text{exits } (\text{bcomps } b \ f \ i) \subseteq \{\text{size } (\text{bcomps } b \ f \ i), i + \text{size } (\text{bcomps } b \ f \ i)\}$
by (*auto simp: exits_def*)

lemma *bcomps_exitsD* [*dest!*]:
 $p \in \text{exits } (\text{bcomps } b \ f \ i) \implies 0 \leq i \implies$
 $p = \text{size } (\text{bcomps } b \ f \ i) \vee p = i + \text{size } (\text{bcomps } b \ f \ i)$
using *bcomps_exits* **by** *auto*

lemma *ccomps_succs*:
 $\text{succs } (\text{ccomps } c) \ n \subseteq \{n..n + \text{size } (\text{ccomps } c)\}$
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!*]:
 $p \in exits (ccomp c) \implies p = size (ccomp c)$
using *ccomp_exits* **by** *auto*

6.5 Splitting up machine executions

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

lemma *exec_n_split*:
fixes $i j :: int$
assumes $P @ c @ P' \vdash (size P + i, s) \rightarrow^n (j, s')$
 $0 \leq i < size c$
 $j \notin \{size P ..< size P + size c\}$
shows $\exists s'' (i'::int) k m.$
 $c \vdash (i, s) \rightarrow^k (i', s'') \wedge$
 $i' \in exits c \wedge$

$$P @ c @ P' \vdash (\text{size } P + i', s'') \rightarrow \hat{m} (j, s') \wedge n = k + m$$

using *assms* **proof** (*induction n arbitrary: i j s*)
case 0
thus ?*case* **by** *simp*
next
case (*Suc n*)
have *i*: $0 \leq i$ *i* < *size c* **by** *fact+*
from *Suc.prem*s
have *j*: $\neg (\text{size } P \leq j \wedge j < \text{size } P + \text{size } c)$ **by** *simp*
from *Suc.prem*s
obtain *i0 s0* **where**
step: $P @ c @ P' \vdash (\text{size } P + i, s) \rightarrow (i0, s0)$ **and**
rest: $P @ c @ P' \vdash (i0, s0) \rightarrow \hat{n} (j, s')$
by *clarsimp*

from *step i*
have *c*: $c \vdash (i, s) \rightarrow (i0 - \text{size } P, s0)$ **by** (*rule exec1_split*)

have $i0 = \text{size } P + (i0 - \text{size } P)$ **by** *simp*
then obtain *j0::int* **where** $i0 = \text{size } P + j0$..

note *split_paired_Ex* [*simp del*]

{ **assume** $j0 \in \{0 \dots \text{size } c\}$
with *j0 j rest c*
have ?*case*
by (*fastforce dest!: Suc.IH intro!: exec_Suc*)
} **moreover** {
assume $j0 \notin \{0 \dots \text{size } c\}$
moreover
from *c j0* **have** $j0 \in \text{succs } c$ 0
by (*auto dest: succs_iexec1 simp: exec1_def simp del: iexec.simps*)
ultimately
have $j0 \in \text{exits } c$ **by** (*simp add: exits_def*)
with *c j0 rest*
have ?*case* **by** *fastforce*
}
ultimately
show ?*case* **by** *cases*
qed

lemma *exec_n_drop_right*:
fixes *j :: int*

assumes $c @ P' \vdash (0, s) \rightarrow \hat{n} (j, s') \ j \notin \{0..<size\ c\}$
shows $\exists s'' i' k m.$
 (if $c = []$ *then* $s'' = s \wedge i' = 0 \wedge k = 0$
 else $c \vdash (0, s) \rightarrow \hat{k} (i', s'') \wedge$
 $i' \in \text{exits } c) \wedge$
 $c @ P' \vdash (i', s'') \rightarrow \hat{m} (j, s') \wedge$
 $n = k + m$
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 $i n :: int$

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

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

proof –

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

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

moreover

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

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

ultimately

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

by (*clarsimp simp: exec1_def simp del: iexec.simps*)

qed

lemma *exec_n_drop_left:*

fixes $i n :: int$

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

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

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

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* ($size\ P \leq i$)

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

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 $\langle \text{exits } P' \subseteq \{0..\} \rangle$
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 $j0 :: int$ **where** $j = size\ P + j0 ..$
ultimately
show *?thesis* **using** *exits*
by (*fastforce dest: exec_n_drop_left*)
qed

6.6 Correctness theorem

lemma *acomp_neq_Nil [simp]*:

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

lemma *acomp_exec_n [dest!]*:

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

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

case (*Plus a1 a2*)

let $?sz = size\ (acomp\ a1) + (size\ (acomp\ a2) + 1)$

from *Plus.prem*s

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

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

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

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

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

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:  $size\ (bcomp\ (And\ b1\ b2)\ f\ j) \leq i\ 0 \leq j$  by fact+

  from And.prems
  obtain  $s''\ stk''$  and  $i':int$  and  $k\ m$  where
     $b1: ?b1 \vdash (0, s, stk) \rightarrow \hat{k}\ (i', s'', stk'')$ 
     $i' = size\ ?b1 \vee i' = ?m + size\ ?b1$  and
     $b2: ?b2 \vdash (i' - size\ ?b1, s'', stk'') \rightarrow \hat{m}\ (i - size\ ?b1, s', stk')$ 
    by (auto\ dest!: bcomp_split\ dest: exec_n_drop_left)
  from b1\ j
  have  $i' = size\ ?b1 + (if\ \neg bval\ b1\ s\ then\ ?m\ else\ 0) \wedge s'' = s \wedge stk'' =$ 
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!]:
   $ccomp\ c = [] \implies (c, s) \Rightarrow s$ 
  by (induct\ c) auto

declare assign_simp [simp]

```

lemma *ccomp_exec_n*:
 $ccomp\ c \vdash (0, s, stk) \rightarrow \hat{n}\ (size(ccomp\ c), t, stk')$
 $\implies (c, s) \Rightarrow t \wedge stk' = stk$
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 $(?cs \vdash (0, s, stk) \rightarrow \hat{n}\ (size\ ?cs, t, stk'))$
obtain $i' :: int$ **and** $k\ m\ s''\ stk''$ **where**
 $cs: ?cs \vdash (i', s'', stk'') \rightarrow \hat{m}\ (size\ ?cs, t, stk')$ **and**
 $?bcomp \vdash (0, s, stk) \rightarrow \hat{k}\ (i', s'', stk'')$
 $i' = size\ ?bcomp \vee i' = size\ ?bcomp + size\ (ccomp\ c1) + 1$
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' :
 $ccomp\ c1 @ JMP\ (size\ (ccomp\ c2)) \# ccomp\ c2 \vdash$
 $(if\ bval\ b\ s\ then\ 0\ else\ size\ (ccomp\ c1) + 1, s, stk) \rightarrow \hat{m}$
 $(1 + size\ (ccomp\ c1) + size\ (ccomp\ c2), t, stk')$
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 isuccs_def)
qed
next
  case (While b c)

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

  { assume  $\neg$  bval b s
    with 1.prem
    have ?case
      by simp
        (fastforce dest!: bcomp_exec_n bcomp_split simp: exec_n_simps)
  } moreover {
    assume b: bval b s
    let ?c0 = WHILE b DO c
    let ?cs = ccomp ?c0
    let ?bs = bcomp b False (size (ccomp c) + 1)
    let ?jmp = [JMP ( $\neg$ ((size ?bs + size (ccomp c) + 1)))]

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

    have ?case
    proof cases
      assume ccomp c = []
      with cs k
      obtain m where
        ?cs  $\vdash$  (0,s,stk)  $\rightarrow$   $\hat{m}$  (size (ccomp ?c0), t, 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) →  $\hat{m}'$  (size (ccomp c), s'', stk'') and
  rest: ?cs ⊢ (size ?bs + size (ccomp c), s'', stk'') →  $\hat{m}$ 
    (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) →  $\hat{k}'$  (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
}
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*:

$ccomp\ c \vdash (0, s, stk) \rightarrow^* (size(ccomp\ c), t, stk) \longleftrightarrow (c, s) \Rightarrow t$
 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 × state) ⇒ (com × state) ⇒ bool (**infix** → 55)

where

Assign: taval a s v ⇒ (x ::= a, s) → (SKIP, s(x := v)) |

Seq1: (SKIP;;c,s) → (c,s) |

Seq2: (c1,s) → (c1',s') ⇒ (c1;;c2,s) → (c1';;c2,s') |

IfTrue: tbval b s True ⇒ (IF b THEN c1 ELSE c2,s) → (c1,s) |

IfFalse: tbval b s False ⇒ (IF b THEN c1 ELSE c2,s) → (c2,s) |

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

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

7.5 The Type System

datatype ty = *Ity* | *Rty*

type_synonym tyenv = vname ⇒ ty

inductive *atyping* :: tyenv ⇒ aexp ⇒ ty ⇒ bool

((1_/ ⊢/ (- :/ -)) [50,0,50] 50)

where

Ic_ty: Γ ⊢ *Ic* i : *Ity* |

Rc_ty: Γ ⊢ *Rc* r : *Rty* |

V_ty: Γ ⊢ *V* x : Γ x |

Plus_ty: Γ ⊢ a1 : τ ⇒ Γ ⊢ a2 : τ ⇒ Γ ⊢ *Plus* a1 a2 : τ

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

inductive_cases [*elim!*]:

Γ ⊢ *V* x : τ Γ ⊢ *Ic* i : τ Γ ⊢ *Rc* r : τ Γ ⊢ *Plus* a1 a2 : τ

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

```

inductive btyping :: tyenv  $\Rightarrow$  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:  $\Gamma \vdash a1 : \tau \Longrightarrow \Gamma \vdash a2 : \tau \Longrightarrow \Gamma \vdash Less\ a1\ a2$ 

declare btyping.intros [intro!]
inductive_cases [elim!]:  $\Gamma \vdash Not\ b \mid \Gamma \vdash And\ b1\ b2 \mid \Gamma \vdash Less\ a1\ a2$ 

inductive ctyping :: tyenv  $\Rightarrow$  com  $\Rightarrow$  bool (infix  $\vdash$  50) where
  Skip_ty:  $\Gamma \vdash SKIP \mid$ 
  Assign_ty:  $\Gamma \vdash a : \Gamma(x) \Longrightarrow \Gamma \vdash x ::= a \mid$ 
  Seq_ty:  $\Gamma \vdash c1 \Longrightarrow \Gamma \vdash c2 \Longrightarrow \Gamma \vdash c1;;c2 \mid$ 
  If_ty:  $\Gamma \vdash b \Longrightarrow \Gamma \vdash c1 \Longrightarrow \Gamma \vdash c2 \Longrightarrow \Gamma \vdash IF\ b\ THEN\ c1\ ELSE\ c2 \mid$ 
  While_ty:  $\Gamma \vdash b \Longrightarrow \Gamma \vdash c \Longrightarrow \Gamma \vdash WHILE\ b\ DO\ c$ 

declare ctyping.intros [intro!]
inductive_cases [elim!]:
   $\Gamma \vdash x ::= a \mid \Gamma \vdash c1;;c2$ 
   $\Gamma \vdash IF\ b\ THEN\ c1\ ELSE\ c2$ 
   $\Gamma \vdash WHILE\ b\ DO\ c$ 

```

7.6 Well-typed Programs Do Not Get Stuck

```

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

lemma type_eq_Ity[simp]: type v = Ity  $\longleftrightarrow (\exists i. v = Iv\ i)$ 
by (cases v) simp_all

lemma type_eq_Rty[simp]: type v = Rty  $\longleftrightarrow (\exists r. v = Rv\ r)$ 
by (cases v) simp_all

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

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

```

```

lemma aprogess:  $\Gamma \vdash a : \tau \implies \Gamma \vdash s \implies \exists v. \text{taval } a \ s \ v$ 
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 bprogess:  $\Gamma \vdash b \implies \Gamma \vdash s \implies \exists v. \text{tbval } b \ s \ v$ 
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 aprogess)
  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 progess:
   $\Gamma \vdash c \implies \Gamma \vdash s \implies c \neq \text{SKIP} \implies \exists cs'. (c,s) \rightarrow cs'$ 
proof(induction rule: ctyping.induct)
  case Skip_ty thus ?case by simp
next
  case Assign_ty
  thus ?case by (metis Assign aprogess)
next
  case Seq_ty thus ?case by simp (metis Seq1 Seq2)
next
  case (If_ty  $\Gamma \ b \ c1 \ c2$ )

```

```

then obtain  $bv$  where  $t\text{bval } b \ s \ bv$  by (metis bprogress)
show  $?case$ 
proof(cases bv)
  assume  $bv$ 
  with  $\langle t\text{bval } b \ s \ bv \rangle$  show  $?case$  by simp (metis IfTrue)
next
  assume  $\neg bv$ 
  with  $\langle t\text{bval } b \ s \ bv \rangle$  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

```

theory Sec_Type_Expr imports Big_Step
begin

```

8.1 Security Levels and Expressions

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*. *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

theory *Sec_Typing* **imports** *Sec_Type_Expr*
begin

8.2 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:

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

While:

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

code_pred (*expected_modes*: *i* \Rightarrow *i* \Rightarrow *bool*) *sec_type* .

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

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

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

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

An important property: anti-monotonicity.

lemma *anti_mono*: $\llbracket l \vdash c; l' \leq l \rrbracket \implies l' \vdash c$
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*: $\llbracket (c,s) \Rightarrow t; l \vdash c \rrbracket \implies s = t (< l)$

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*:

$\llbracket (c,s) \Rightarrow s'; (c,t) \Rightarrow t'; 0 \vdash c; s = t (\leq l) \rrbracket$
 $\implies s' = t' (\leq l)$

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  $\text{sec } x \geq \text{sec } a$  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  $\text{aval } a \ s = \text{aval } a \ t$ 
    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
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.prem1(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  $\langle (\text{IF } b \ \text{THEN } c1 \ \text{ELSE } c2, t) \Rightarrow t' \rangle 1 \rangle \langle \neg \text{sec } b \leq l \rangle$ ]
    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  $sec\ b \leq l$ 
hence  $s = t (\leq sec\ b)$  using  $\langle s = t (\leq l) \rangle$  by auto
hence  $\neg\ bval\ b\ t$  using  $\langle \neg\ bval\ b\ s \rangle$  by(simp add: bval_eq_if_eq_le)
with IfFalse.IH IfFalse.prems(1,3)  $\langle sec\ b \vdash c2 \rangle$  anti_mono
show ?thesis by auto
next
assume  $\neg\ sec\ b \leq l$ 
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 confinement[OF \langle (IF\ b\ THEN\ c1\ ELSE\ c2, t) \Rightarrow t' \rangle 1]  $\langle \neg\ sec\ b \leq l \rangle$ 
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 (WhileFalse  $b\ s\ c$ )
have  $sec\ b \vdash c$  using WhileFalse.prems(2) by auto
show ?case
proof cases
  assume  $sec\ b \leq l$ 
  hence  $s = t (\leq sec\ b)$  using  $\langle s = t (\leq l) \rangle$  by auto
  hence  $\neg\ bval\ b\ t$  using  $\langle \neg\ bval\ b\ s \rangle$  by(simp add: bval_eq_if_eq_le)
  with WhileFalse.prems(1,3) show ?thesis by auto
next
assume  $\neg\ sec\ b \leq l$ 
have  $1: sec\ b \vdash WHILE\ b\ DO\ c$ 
  by(rule sec.type.intros)(simp_all add: \langle sec\ b \vdash c \rangle)
from confinement[OF \langle (WHILE\ b\ DO\ c, t) \Rightarrow t' \rangle 1]  $\langle \neg\ sec\ b \leq l \rangle$ 
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  $?w = WHILE\ b\ DO\ c$ 
have  $sec\ b \vdash c$  using  $\langle 0 \vdash WHILE\ b\ DO\ c \rangle$  by auto
show ?case
proof cases
  assume  $sec\ b \leq l$ 
  hence  $s1 = t1 (\leq sec\ b)$  using  $\langle s1 = t1 (\leq l) \rangle$  by auto
  hence  $bval\ b\ t1$ 
  using  $\langle bval\ b\ s1 \rangle$  by(simp add: bval_eq_if_eq_le)

```

```

then obtain  $t2$  where  $(c, t1) \Rightarrow t2$   $(?w, t2) \Rightarrow t3$ 
  using  $\langle (?w, t1) \Rightarrow t3 \rangle$  by auto
from WhileTrue.IH(2)[OF  $\langle (?w, t2) \Rightarrow t3 \rangle$   $\langle 0 \vdash ?w \rangle$ ]
  WhileTrue.IH(1)[OF  $\langle (c, t1) \Rightarrow t2 \rangle$  anti_mono[OF  $\langle sec\ b \vdash c \rangle$ ]
     $\langle s1 = t1 (\leq l) \rangle$ ]]
show ?thesis by simp
next
  assume  $\neg sec\ b \leq l$ 
  have  $1: sec\ b \vdash ?w$  by (rule sec_type.intros)(simp_all add:  $\langle sec\ b \vdash c \rangle$ )
  from confinement[OF big_step.WhileTrue[OF WhileTrue.hyps]  $1$ ]  $\langle \neg sec\ b \leq l \rangle$ 
  have  $s1 = s3 (\leq l)$  by auto
  moreover
  from confinement[OF  $\langle (WHILE\ b\ DO\ c,\ t1) \Rightarrow t3 \rangle$   $1$ ]  $\langle \neg sec\ b \leq l \rangle$ 
  have  $t1 = t3 (\leq l)$  by auto
  ultimately show  $s3 = t3 (\leq l)$  using  $\langle s1 = t1 (\leq l) \rangle$  by auto
qed
qed

```

8.3 The Standard Typing System

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

inductive *sec_type'* :: *nat* \Rightarrow *com* \Rightarrow *bool* ($(_ / \vdash' _)$ [$0, 0$] 50) **where**

Skip':

$l \vdash' SKIP \mid$

Assign':

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

Seq':

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

If':

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

While':

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

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*: $l \vdash' c \implies l \vdash c$
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*)

8.4 A Bottom-Up Typing System

inductive *sec_type2* :: *com* \Rightarrow *level* \Rightarrow *bool* ((\vdash _ : _) [0,0] 50) **where**

Skip2:

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

Assign2:

$\text{sec } x \geq \text{sec } a \implies \vdash x ::= a : \text{sec } x \mid$

Seq2:

$\llbracket \vdash c_1 : l_1; \vdash c_2 : l_2 \rrbracket \implies \vdash c_1;;c_2 : \min l_1 l_2 \mid$

If2:

$\llbracket \text{sec } b \leq \min l_1 l_2; \vdash c_1 : l_1; \vdash c_2 : l_2 \rrbracket$
 $\implies \vdash \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 : \min l_1 l_2 \mid$

While2:

$\llbracket \text{sec } b \leq l; \vdash c : l \rrbracket \implies \vdash \text{WHILE } b \text{ DO } c : l$

lemma *sec_type2_sec_type'*: $\vdash c : l \implies l \vdash' c$

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*: $l \vdash' c \implies \exists l' \geq l. \vdash c : l'$

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
theory *Sec_TypingT* **imports** *Sec_Type_Expr*
begin

8.5 A Termination-Sensitive Syntax Directed System

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:

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

If:

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

While:

$\text{sec } b = 0 \Longrightarrow 0 \vdash c \Longrightarrow 0 \vdash \text{WHILE } b \text{ DO } c$

code_pred (*expected_modes: i => i => bool*) *sec_type* .

inductive_cases [*elim!*]:

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

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  $\max (sec\ b) \vdash c1$  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  $\max (sec\ b) \vdash c2$  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  $sec\ x \geq sec\ a$  using  $\langle 0 \vdash x ::= a \rangle$  by auto
  have  $(x ::= a, t) \Rightarrow t(x := aval\ a\ t)$  by auto
  moreover
  have  $s(x := aval\ a\ s) = t(x := aval\ a\ t) (\leq l)$ 
  proof auto
    assume  $sec\ x \leq l$ 
    with  $\langle sec\ x \geq sec\ a \rangle$  have  $sec\ a \leq l$  by arith
    thus  $aval\ a\ s = aval\ a\ t$ 
    by (rule aval_eq_if_eq_le[OF  $\langle s = t (\leq l) \rangle$ ])
  next
  fix y assume  $y \neq x$   $sec\ y \leq l$ 

```

```

    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  $\langle 0 \vdash IF\ b\ THEN\ c1\ ELSE\ c2 \rangle$  by auto
  obtain  $t'$  where  $t': (c1, t) \Rightarrow t' s' = t' (\leq l)$ 
    using IfTrue.IH[OF anti_mono[OF  $\langle sec\ b \vdash c1 \rangle \langle s = t (\leq l) \rangle$ ]] by blast
  show ?case
  proof cases
    assume  $sec\ b \leq l$ 
    hence  $s = t (\leq sec\ b)$  using  $\langle s = t (\leq l) \rangle$  by auto
    hence  $bval\ b\ t$  using  $\langle bval\ b\ s \rangle$  by(simp add: bval_eq_if_eq_le)
    thus ?thesis by (metis t' big_step.IfTrue)
  next
    assume  $\neg sec\ b \leq l$ 
    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.IfTrue[OF IfTrue( $1, 2$ )]  $1$ ]  $\langle \neg sec\ b \leq l \rangle$ 
    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 (IfFalse  $b s c2 s' c1$ )
  have  $sec\ b \vdash c1\ sec\ b \vdash c2$  using  $\langle 0 \vdash IF\ b\ THEN\ c1\ ELSE\ c2 \rangle$  by auto
  obtain  $t'$  where  $t': (c2, t) \Rightarrow t' s' = t' (\leq l)$ 
    using IfFalse.IH[OF anti_mono[OF  $\langle sec\ b \vdash c2 \rangle \langle s = t (\leq l) \rangle$ ]] by blast
  show ?case
  proof cases
    assume  $sec\ b \leq l$ 
    hence  $s = t (\leq sec\ b)$  using  $\langle s = t (\leq l) \rangle$  by auto
    hence  $\neg bval\ b\ t$  using  $\langle \neg bval\ b\ s \rangle$  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  $0: \text{sec } b \neq 0$  by arith
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 big_step.IfFalse[OF IfFalse(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 (WhileFalse  $b \ s \ c$ )
hence [simp]:  $\text{sec } b = 0$  by auto
have  $s = t (\leq \text{sec } b)$  using  $\langle s = t (\leq l) \rangle$  by auto
hence  $\neg \text{bval } b \ t$  using  $\langle \neg \text{bval } b \ s \rangle$  by (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  $?w = \text{WHILE } b \ \text{DO } c$ 
from  $\langle 0 \vdash ?w \rangle$  have [simp]:  $\text{sec } b = 0$  by auto
have  $0 \vdash c$  using  $\langle 0 \vdash \text{WHILE } b \ \text{DO } c \rangle$  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  $\langle \text{bval } b \ s \rangle$  have  $\text{bval } b \ t$ 
  using bval_eq_if_eq_le[OF  $\langle s = t (\leq l) \rangle$ ] by auto
show ?case
  using big_step.WhileTrue[OF  $\langle \text{bval } b \ t \rangle \langle (c, t) \Rightarrow t'' \rangle \langle (?w, t'') \Rightarrow t' \rangle$ ]
  by (metis  $\langle s' = t' (\leq l) \rangle$ )
qed

```

8.6 The Standard Termination-Sensitive System

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

inductive *sec_type'* :: *nat* \Rightarrow *com* \Rightarrow *bool* ($(_ / \vdash'' _)$ [0,0] 50) **where**

Skip':
 $l \vdash' SKIP \mid$
Assign':
 $\llbracket sec\ x \geq sec\ a; \ sec\ x \geq l \rrbracket \Longrightarrow l \vdash' x ::= a \mid$
Seq':
 $l \vdash' c_1 \Longrightarrow l \vdash' c_2 \Longrightarrow l \vdash' c_1; c_2 \mid$
If':
 $\llbracket sec\ b \leq l; \ l \vdash' c_1; \ l \vdash' c_2 \rrbracket \Longrightarrow l \vdash' IF\ b\ THEN\ c_1\ ELSE\ c_2 \mid$
While':
 $\llbracket sec\ b = 0; \ 0 \vdash' c \rrbracket \Longrightarrow 0 \vdash' WHILE\ b\ DO\ c \mid$
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 a₁ a₂) s =
(case (aval a₁ s, aval a₂ s) of
(Some i₁, Some i₂) \Rightarrow Some(i₁+i₂) | _ \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 b₁ b₂) s = (case (bval b₁ s, bval b₂ s) of
(Some bv₁, Some bv₂) \Rightarrow Some(bv₁ & bv₂) | _ \Rightarrow None) |

bval (*Less* a_1 a_2) $s = (\text{case } (\text{aval } a_1 \ s, \text{aval } a_2 \ s) \text{ of}$
(Some $i_1, \text{Some } i_2) \Rightarrow \text{Some}(i_1 < i_2) \mid _ \Rightarrow \text{None}$)

lemma *aval_Some*: $\text{vars } a \subseteq \text{dom } s \Longrightarrow \exists i. \text{aval } a \ s = \text{Some } i$
by (*induct* a) *auto*

lemma *bval_Some*: $\text{vars } b \subseteq \text{dom } s \Longrightarrow \exists bv. \text{bval } b \ s = \text{Some } bv$
by (*induct* b) (*auto dest!*: *aval_Some*)

end

theory *Def_Init*

imports *Vars Com*

begin

9.3 Definite Initialization Analysis

inductive *D* :: $\text{vname set} \Rightarrow \text{com} \Rightarrow \text{vname set} \Rightarrow \text{bool}$ **where**

Skip: $D \ A \ \text{SKIP} \ A \mid$

Assign: $\text{vars } a \subseteq A \Longrightarrow D \ A \ (x ::= a) \ (\text{insert } x \ A) \mid$

Seq: $\llbracket D \ A_1 \ c_1 \ A_2; D \ A_2 \ c_2 \ A_3 \rrbracket \Longrightarrow D \ A_1 \ (c_1;; c_2) \ A_3 \mid$

If: $\llbracket \text{vars } b \subseteq A; D \ A \ c_1 \ A_1; D \ A \ c_2 \ A_2 \rrbracket \Longrightarrow$

$D \ A \ (\text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2) \ (A_1 \ \text{Int } A_2) \mid$

While: $\llbracket \text{vars } b \subseteq A; D \ A \ c \ A' \rrbracket \Longrightarrow D \ A \ (\text{WHILE } b \ \text{DO } c) \ A$

inductive_cases [*elim!*]:

$D \ A \ \text{SKIP} \ A'$

$D \ A \ (x ::= a) \ A'$

$D \ A \ (c_1;;c_2) \ A'$

$D \ A \ (\text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2) \ A'$

$D \ A \ (\text{WHILE } b \ \text{DO } c) \ A'$

lemma *D_incr*:

$D \ A \ c \ A' \Longrightarrow A \subseteq A'$

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 × state option) ⇒ state option ⇒ bool (**infix** ⇒ 55)

where

None: (c, None) ⇒ None |

Skip: (SKIP, s) ⇒ s |

AssignNone: aval a s = None ⇒ (x ::= a, Some s) ⇒ None |

Assign: aval a s = Some i ⇒ (x ::= a, Some s) ⇒ Some(s(x := Some i))

|

Seq: (c₁, s₁) ⇒ s₂ ⇒ (c₂, s₂) ⇒ s₃ ⇒ (c₁;;c₂, s₁) ⇒ s₃ |

IfNone: bval b s = None ⇒ (IF b THEN c₁ ELSE c₂, Some s) ⇒ None |

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

(IF b THEN c₁ ELSE c₂, Some s) ⇒ s' |

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

(IF b THEN c₁ ELSE c₂, Some s) ⇒ s' |

WhileNone: bval b s = None ⇒ (WHILE b DO c, Some s) ⇒ None |

WhileFalse: bval b s = Some False ⇒ (WHILE b DO c, Some s) ⇒ Some s |

WhileTrue:

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

⇒

(WHILE b DO c, Some s) ⇒ 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*:

$\llbracket (c, \text{Some } s) \Rightarrow s'; D \ A \ c \ A'; A \subseteq \text{dom } s \rrbracket$

⇒ ∃ t. s' = Some t ∧ A' ⊆ 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*:

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

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*: $D \ A \ c \ M \implies A \subseteq A' \implies \text{EX } M'. D \ A' \ c \ M' \ \& \ M \leq M'$

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*:

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

proof (*induction arbitrary: A rule: small_step_induct*)

case (*While b c s*)

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

then obtain *A''* **where** $D \ A' \ c \ A''$ **by** (*metis D_incr D_mono*)

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

```

(dom s)
  by (metis D.If[OF ‹vars b ⊆ dom s› D.Seq[OF ‹D (dom s) c A'›
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*:

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

lemma *equiv_up_to_seq*:

$P \models c \sim c' \implies Q \models d \sim d' \implies$
 $(\bigwedge s s'. (c, s) \Rightarrow s' \implies P s \implies Q s') \implies$
 $P \models (c;; d) \sim (c';; d')$
by (*clarsimp simp: equiv_up_to_def*) *blast*

lemma *equiv_up_to_while_lemma_weak*:

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

proof (*induction rule: big_step_induct*)

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

hence *IH*: $P \ s2 \implies (\text{WHILE } b' \ \text{DO } c', s2) \Rightarrow s3$ **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 $\bigwedge s s'. (c, s) \Rightarrow s' \implies P s \implies \text{bval } b \ s \implies P s'$ **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 $I: \bigwedge s s'. (c, s) \Rightarrow s' \implies P s \implies \text{bval } b \ s \implies P s'$

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]*:

$P \models \text{WHILE } Bc \text{ True DO } c \sim \text{WHILE } Bc \text{ True DO SKIP}$

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 (ALL 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 = (λm . if t1 m = t2 m then t1 m 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 $\text{defs } c1 \ t \ |' \ (- \ \text{lvars } c1) \ |' \ (- \ \text{lvars } c2) =$
 $t \ |' \ (- \ \text{lvars } c1) \ |' \ (- \ \text{lvars } c2)$ **by** *simp*
moreover
from *If*
have $\text{defs } c2 \ t \ |' \ (- \ \text{lvars } c2) = t \ |' \ (- \ \text{lvars } c2)$ **by** *simp*
hence $\text{defs } c2 \ t \ |' \ (- \ \text{lvars } c2) \ |' \ (- \ \text{lvars } c1) =$
 $t \ |' \ (- \ \text{lvars } c2) \ |' \ (- \ \text{lvars } c1)$ **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 $\text{approx } (\text{defs } c1 \ t) \ s2$ **by** (*rule Seq.IH(1)[OF Seq.premis]*)
hence $\text{approx } (\text{defs } c2 \ (\text{defs } c1 \ t)) \ s3$ **by** (*rule Seq.IH(2)*)
thus *?case* **by** *simp*
next
case (*IfTrue b s c1 s'*)
hence $\text{approx } (\text{defs } c1 \ t) \ s'$ **by** *simp*
thus *?case* **by** (*simp add: approx_merge*)
next
case (*IfFalse b s c2 s'*)
hence $\text{approx } (\text{defs } c2 \ t) \ s'$ **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 $\text{approx } (\text{defs } c \ t) \ s2$ **by** *simp*
with *WhileTrue*
have $\text{approx } (\text{defs } c \ t \ |' \ (- \ \text{lvars } c)) \ s3$ **by** *simp*
thus *?case* **by** (*simp add: defs_restrict*)
qed

```

lemma big_step_pres_approx_restrict:
   $(c, s) \Rightarrow s' \Longrightarrow \text{approx } (t \mid' (-lvars\ c))\ s \Longrightarrow \text{approx } (t \mid' (-lvars\ c))\ s'$ 
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  $\text{approx } (t \mid' (-lvars\ c2) \mid' (-lvars\ c1))\ s1$ 
    by (simp add: Int_commute)
  hence  $\text{approx } (t \mid' (-lvars\ c2) \mid' (-lvars\ c1))\ s2$ 
    by (rule Seq)
  hence  $\text{approx } (t \mid' (-lvars\ c1) \mid' (-lvars\ c2))\ s2$ 
    by (simp add: Int_commute)
  hence  $\text{approx } (t \mid' (-lvars\ c1) \mid' (-lvars\ c2))\ s3$ 
    by (rule Seq)
  thus ?case by simp
next
  case (IfTrue b s c1 s' c2)
  hence  $\text{approx } (t \mid' (-lvars\ c2) \mid' (-lvars\ c1))\ s$ 
    by (simp add: Int_commute)
  hence  $\text{approx } (t \mid' (-lvars\ c2) \mid' (-lvars\ c1))\ s'$ 
    by (rule IfTrue)
  thus ?case by (simp add: Int_commute)
next
  case (IfFalse b s c2 s' c1)
  hence  $\text{approx } (t \mid' (-lvars\ c1) \mid' (-lvars\ c2))\ s$ 
    by simp
  hence  $\text{approx } (t \mid' (-lvars\ c1) \mid' (-lvars\ c2))\ s'$ 
    by (rule IfFalse)
  thus ?case by simp
qed auto

```

```

declare assign_simp [simp]

```

```

lemma approx_eq:
   $\text{approx } t \models c \sim \text{fold } c\ t$ 
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))  $\models$ 
  WHILE b DO c  $\sim$  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 empty = ( $\lambda$ _. True)
by (auto simp: approx_def)

```

```

theorem constant_folding_equiv:
  fold c empty  $\sim$  c
using approx_eq [of 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_lfp*:

$vars\ b \cup X \cup L\ c\ P \subseteq P \implies L\ (WHILE\ b\ DO\ c)\ X \subseteq P$

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.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 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 by (metis WhileFalse.prems 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.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

11.3 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.premis **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 \langle s = t on L c1 X \rangle] **obtain** t' **where**

(bury c1 X, t) \Rightarrow t' s' = t' on X **by auto**

thus ?case **using** \langle bval b t \rangle **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 \langle s = t on L c2 X \rangle] **obtain** t' **where**

$(\text{bury } c2 \ X, t) \Rightarrow t' \ s' = t' \ \text{on } X$ **by** *auto*
thus *?case using* $\langle \sim \text{bval } b \ t \rangle$ **by** *auto*
next
case $(\text{WhileFalse } b \ s \ c)$
hence $\sim \text{bval } b \ t$ **by** $(\text{metis } L_While_vars \ \text{bval_eq_if_eq_on_vars} \ \text{set_mp})$
thus *?case*
by *simp* $(\text{metis } L_While_X \ \text{WhileFalse.prem} \ \text{big_step} \ \text{WhileFalse} \ \text{set_mp})$
next
case $(\text{WhileTrue } b \ s1 \ c \ s2 \ s3 \ X \ t1)$
let $?w = \text{WHILE } b \ DO \ c$
from $\langle \text{bval } b \ s1 \rangle \ \text{WhileTrue.prem} \ \text{have} \ \text{bval } b \ t1$
by $(\text{metis } L_While_vars \ \text{bval_eq_if_eq_on_vars} \ \text{set_mp})$
have $s1 = t1 \ \text{on } L \ c \ (L \ ?w \ X)$
using $L_While_pfp \ \text{WhileTrue.prem} \ \text{by} \ \text{blast}$
from $\text{WhileTrue.IH}(1)[\text{OF } \text{this}]$ **obtain** $t2$ **where**
 $(\text{bury } c \ (L \ ?w \ X), t1) \Rightarrow t2 \ s2 = t2 \ \text{on } L \ ?w \ X$ **by** *auto*
from $\text{WhileTrue.IH}(2)[\text{OF } \text{this}(2)]$ **obtain** $t3$
where $(\text{bury } ?w \ X, t2) \Rightarrow t3 \ s3 = t3 \ \text{on } X$
by *auto*
with $\langle \text{bval } b \ t1 \rangle \ \langle (\text{bury } c \ (L \ ?w \ X), t1) \Rightarrow t2 \rangle$ **show** *?case* **by** *auto*
qed

corollary *final_bury_correct*: $(c, s) \Rightarrow s' \Longrightarrow (\text{bury } c \ UNIV, s) \Rightarrow s'$
using *bury_correct*[*of* $c \ s \ s' \ UNIV$]
by $(\text{auto } \text{simp: } \text{fun_eq_iff}[\text{symmetric}])$

Now the opposite direction.

lemma *SKIP_bury*[*simp*]:
 $\text{SKIP} = \text{bury } c \ X \longleftrightarrow c = \text{SKIP} \mid (\exists x \ a. \ c = x ::= a \ \& \ x \notin X)$
by $(\text{cases } c) \ \text{auto}$

lemma *Assign_bury*[*simp*]: $x ::= a = \text{bury } c \ X \longleftrightarrow c = x ::= a \ \& \ x : X$
by $(\text{cases } c) \ \text{auto}$

lemma *Seq_bury*[*simp*]: $bc_1 ;; bc_2 = \text{bury } c \ X \longleftrightarrow$
 $(\exists c_1 \ c_2. \ c = c_1 ;; c_2 \ \& \ bc_2 = \text{bury } c_2 \ X \ \& \ bc_1 = \text{bury } c_1 \ (L \ c_2 \ X))$
by $(\text{cases } c) \ \text{auto}$

lemma *If_bury*[*simp*]: $\text{IF } b \ \text{THEN } bc_1 \ \text{ELSE } bc_2 = \text{bury } c \ X \longleftrightarrow$
 $(\exists c_1 \ c_2. \ c = \text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2 \ \& \ bc_1 = \text{bury } c_1 \ X \ \& \ bc_2 = \text{bury } c_2 \ X)$
by $(\text{cases } c) \ \text{auto}$

lemma *While_bury*[*simp*]: $\text{WHILE } b \ DO \ bc' = \text{bury } c \ X \longleftrightarrow$

($EX c'. c = \text{WHILE } b \text{ DO } c' \ \& \ bc' = \text{bury } c' (L (\text{WHILE } b \text{ DO } c')) X$)
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** $c: c = \text{IF } b \ \text{THEN } c1 \ \text{ELSE } c2$

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* ($\text{bval } b \ t$) **by** *auto*

next

case (*IfFalse b s bc2 s' bc1*)

then obtain *c1 c2* **where** $c: c = \text{IF } b \ \text{THEN } c1 \ \text{ELSE } c2$

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* ($\sim \text{bval } b \ t$) **by** *auto*

next

case (*WhileFalse b s c*)

```

hence  $\sim$  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  $\langle$ bval b s1 $\rangle$  WhileTrue.prems 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.prems w by blast
  with IH(1)[OF bc', of t1] w obtain t2 where
     $\langle$ c', t1 $\rangle \Rightarrow$  t2 s2 = t2 on L w X by auto
  from IH(2)[OF WhileTrue.hyps(6), of t2] w this(2) obtain t3
    where  $\langle$ w, t2 $\rangle \Rightarrow$  t3 s3 = t3 on X
    by auto
  with  $\langle$ bval b t1 $\rangle$   $\langle$ c', t1 $\rangle \Rightarrow$  t2 $\rangle$  w show ?case by auto
qed

corollary final_bury_correct2: (bury c UNIV, s)  $\Rightarrow$  s'  $\implies$  (c, s)  $\Rightarrow$  s'
using bury_correct2[of c UNIV]
by (auto simp: fun_eq_iff[symmetric])

corollary bury_sim: bury c UNIV  $\sim$  c
by(metis final_bury_correct final_bury_correct2)

end

```

```

theory Live_True
imports HOL-Library.While_Combinator Vars Big_Step
begin

```

11.4 True Liveness Analysis

```

fun L :: com  $\Rightarrow$  vname set  $\Rightarrow$  vname set where
  L SKIP X = X |
  L (x ::= a) X = (if x  $\in$  X then vars a  $\cup$  (X - {x}) else 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 = lfp( $\lambda$ Y. vars b  $\cup$  X  $\cup$  L c Y)

```

```

lemma L_mono: mono (L c)
proof–
  { fix X Y have  $X \subseteq Y \implies L\ c\ X \subseteq L\ c\ Y$ 
    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]

```

11.5 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.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 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)  $\Rightarrow$  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 using WhileFalse.premis 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.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

11.6 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 *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 $b = Less (N 0) (V \text{"y"}); c = \text{"y"} ::= V \text{"x"}; \text{"x"} ::= V \text{"z"}$
in $L (WHILE b DO c) \{\text{"y"}\} = \{\text{"x"}, \text{"y"}, \text{"z"}\}$
by *eval*

11.7 Limiting the number of iterations

The final parameter is the default value:

fun $iter :: ('a \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ **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 $Lb :: com \Rightarrow vname set \Rightarrow vname set$ **where**
 $Lb SKIP X = X$ |
 $Lb (x ::= a) X = (if x \in X then X - \{x\} \cup vars a else X)$ |
 $Lb (c_1;; c_2) X = (Lb c_1 \circ Lb c_2) X$ |
 $Lb (IF b THEN c_1 ELSE c_2) X = vars b \cup Lb c_1 X \cup Lb c_2 X$ |
 $Lb (WHILE b DO c) X = iter (\lambda A. vars b \cup X \cup Lb c A) 2 \{\}$ ($vars b \cup rvars c \cup X$)

Lb (and $iter$) is not monotone!

lemma let $w = WHILE Bc False DO (\text{"x"} ::= V \text{"y"}; \text{"z"} ::= V \text{"x"})$
in $\neg (Lb w \{\text{"z"}\} \subseteq Lb w \{\text{"y"}, \text{"z"}\})$
by *eval*

lemma lfp_subset_iter :

$\llbracket mono f; !!X. f X \subseteq f' X; lfp f \subseteq D \rrbracket \implies lfp f \subseteq iter f' n A D$

proof(*induction n arbitrary: A*)

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

next

case *Suc* **thus** ?case **by** *simp (metis lfp_lowerbound)*

qed

lemma $L c X \subseteq Lb c X$

proof(*induction c arbitrary: X*)

case (*While b c*)

let ?f = $\lambda A. vars b \cup X \cup L c A$

```

let ?fb =  $\lambda A. \text{vars } b \cup X \cup Lb \ c \ A$ 
show ?case
proof (simp add: L.simps(5), rule lfp_subset_iter[OF mono_union_L])
  show !!X. ?f X  $\subseteq$  ?fb X using While.IH by blast
  show lfp ?f  $\subseteq$  vars b  $\cup$  rvars c  $\cup$  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* \times *state*) *set*

definition *W* :: (*state* \Rightarrow *bool*) \Rightarrow *com_den* \Rightarrow (*com_den* \Rightarrow *com_den*)
where
W db dc = ($\lambda dw. \{(s,t). \text{if } db \ s \ \text{then } (s,t) \in dc \ O \ dw \ \text{else } s=t\}$)

fun *D* :: *com* \Rightarrow *com_den* **where**
D SKIP = *Id* |
D (x ::= a) = $\{(s,t). t = s(x := \text{aval } a \ s)\}$ |
D (c1;;c2) = *D(c1)* *O* *D(c2)* |
D (IF b THEN c1 ELSE c2)
= $\{(s,t). \text{if } bval \ b \ s \ \text{then } (s,t) \in D \ c1 \ \text{else } (s,t) \in D \ c2\}$ |
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*: $(s, t) \in D(c) \implies (s, t) : Big_step\ c$

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*: $(c1 \sim c2) \iff D\ c1 = D\ c2$

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** $f :: 'a \text{ set} \Rightarrow 'b \text{ set}$
assumes $\text{cont } f$ **shows** $\text{mono } f$
proof
fix $a \ b :: 'a \text{ set}$ **assume** $a \subseteq b$
let $?S = \lambda n::\text{nat}. \text{if } n=0 \text{ then } a \text{ else } b$
have $\text{chain } ?S$ **using** $\langle a \subseteq b \rangle$ **by** $(\text{auto simp: chain_def})$
hence $f(\text{UN } n. ?S \ n) = (\text{UN } n. f(?S \ n))$
using *assms* **by** $(\text{simp add: cont_def})$
moreover **have** $(\text{UN } n. ?S \ n) = b$ **using** $\langle a \subseteq b \rangle$ **by** $(\text{auto split: if_splits})$
moreover **have** $(\text{UN } n. f(?S \ n)) = f \ a \cup f \ b$ **by** $(\text{auto split: if_splits})$
ultimately **show** $f \ a \subseteq f \ b$ **by** (metis Un_upper1)
qed

lemma *chain_iterates*: **fixes** $f :: 'a \text{ set} \Rightarrow 'a \text{ set}$
assumes $\text{mono } f$ **shows** $\text{chain}(\lambda n. (f \ ^n) \ \{\})$
proof–
{ fix n **have** $(f \ ^n) \ \{\} \subseteq (f \ ^{Suc \ n}) \ \{\}$ **using** *assms*
by $(\text{induction } n) (\text{auto simp: mono_def}) \}$
thus *?thesis* **by** $(\text{auto simp: chain_def})$
qed

theorem *lfp_if_cont*:
assumes $\text{cont } f$ **shows** $\text{lfp } f = (\text{UN } n. (f \ ^n) \ \{\}) (\text{is } _ = ?U)$
proof
from *assms mono_if_cont*
have $\text{mono: } (f \ ^n) \ \{\} \subseteq (f \ ^{Suc \ n}) \ \{\}$ **for** n
using *funpow_decreasing* $[\text{of } n \ \text{Suc } n]$ **by** *auto*
show $\text{lfp } f \subseteq ?U$
proof $(\text{rule lfp_lowerbound})$
have $f \ ?U = (\text{UN } n. (f \ ^{Suc \ n}) \ \{\})$
using *chain_iterates* $[\text{OF } \text{mono_if_cont}[\text{OF } \text{assms}]]$ *assms*
by $(\text{simp add: cont_def})$
also **have** $\dots = (f \ ^0) \ \{\} \cup \dots$ **by** *simp*
also **have** $\dots = ?U$
using *mono* **by** *auto* $(\text{metis funpow_simps_right}(2) \ \text{funpow_swap1} \ \text{o_apply})$
finally **show** $f \ ?U \subseteq ?U$ **by** *simp*
qed
next
{ fix $n \ p$ **assume** $f \ p \subseteq p$
have $(f \ ^n) \ \{\} \subseteq p$
proof $(\text{induction } n)$
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
  }
  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
  thus ?case by simp

```

qed (*auto simp add: single_valued_def*)

end

13 Hoare Logic

theory *Hoare* **imports** *Big_Step* **begin**

13.1 Hoare Logic for Partial Correctness

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

end

theory *Hoare_Examples* **imports** *Hoare* **begin**

hide_const (**open**) *sum*

Summing up the first x natural numbers in variable y .

fun *sum* :: *int* \Rightarrow *int* **where**
sum $i =$ (*if* $i \leq 0$ *then* 0 *else* *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* ==

WHILE *Less* ($N\ 0$) (*V* "*x*")
DO ("*y*" ::= *Plus* (*V* "*y*") (*V* "*x*"));
"*x*" ::= *Plus* (*V* "*x*") ($N\ (-\ 1)$))

13.1.1 Proof by Operational Semantics

The behaviour of the loop is proved by induction:


```

lemma while_sum:
  (wsum, s)  $\Rightarrow$  t  $\implies$  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

```

13.1.2 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

13.2 Soundness and Completeness

```
theory Hoare_Sound_Complete
imports Hoare
begin
```

13.2.1 Soundness

```
lemma hoare_sound:  $\vdash \{P\}c\{Q\} \implies \models \{P\}c\{Q\}$ 
proof(induction rule: hoare.induct)
  case (While P b c)
  { fix s t
    have (WHILE b DO c,s)  $\Rightarrow$  t  $\implies$  P s  $\implies$  P t  $\wedge$   $\neg$  bval b 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)
```

13.2.2 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 (c1;;c2) Q = wp c1 (wp c2 Q)
by (rule ext) (auto simp: wp_def)
```

```
lemma wp_If[simp]:
wp (IF b THEN c1 ELSE c2) Q =
( $\lambda$ s. if bval b s then wp c1 Q s else wp c2 Q s)
by (rule ext) (auto simp: wp_def)
```

```
lemma wp_While_If:
wp (WHILE b DO c) Q s =
wp (IF b THEN c;;WHILE b DO c ELSE SKIP) Q s
```

unfolding *wp_def* **by** (*metis unfold_while*)

lemma *wp_While_True[simp]*: $bval\ b\ s \implies$
 $wp\ (WHILE\ b\ DO\ c)\ Q\ s = wp\ (c;;\ WHILE\ b\ DO\ c)\ Q\ s$
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*)

13.2.3 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

theory *VCG* **imports** *Hoare* **begin**

13.3 Verification Conditions

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* \Rightarrow *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)

```

Weakest precondition from annotated commands:

fun *pre* :: *acom* \Rightarrow *assn* \Rightarrow *assn* **where**

```

pre SKIP Q = Q |
pre (x ::= a) Q = ( $\lambda s. Q(s(x := \text{aval } a \ s))$ ) |
pre (C1;; C2) Q = pre C1 (pre C2 Q) |
pre (IF b THEN C1 ELSE C2) Q =
  ( $\lambda s. \text{if } \text{bval } b \ s \ \text{then } \text{pre } C_1 \ Q \ s \ \text{else } \text{pre } C_2 \ Q \ s$ ) |
pre ({I} WHILE b DO C) Q = I

```

Verification condition:

fun *vc* :: *acom* \Rightarrow *assn* \Rightarrow *bool* **where**

```

vc SKIP Q = True |
vc (x ::= a) Q = True |
vc (C1;; C2) Q = (vc C1 (pre C2 Q)  $\wedge$  vc C2 Q) |
vc (IF b THEN C1 ELSE C2) Q = (vc C1 Q  $\wedge$  vc C2 Q) |
vc ({I} WHILE b DO C) Q =
  ( $(\forall s. (I \ s \ \wedge \ \text{bval } b \ s \ \longrightarrow \ \text{pre } C \ I \ s) \ \wedge$ 
    ( $I \ s \ \wedge \ \neg \ \text{bval } b \ s \ \longrightarrow \ Q \ s$ ))  $\wedge$ 
   vc C I)

```

Soundness:

lemma *vc_sound*: *vc C Q* \Longrightarrow \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'*:
 $\llbracket vc \ C \ Q; \forall s. \ P \ s \ \longrightarrow \ pre \ C \ Q \ s \rrbracket \Longrightarrow \vdash \{P\} \ strip \ C \ \{Q\}$
by (*metis strengthen_pre vc_sound*)

Completeness:

lemma *pre_mono*:
 $\forall s. \ P \ s \ \longrightarrow \ P' \ s \ \Longrightarrow \ pre \ C \ P \ s \ \Longrightarrow \ pre \ C \ P' \ s$
proof (*induction C arbitrary: P P' s*)
case *Aseq* **thus** *?case* **by** *simp metis*
qed *simp_all*

lemma *vc_mono*:
 $\forall s. \ P \ s \ \longrightarrow \ P' \ s \ \Longrightarrow \ vc \ C \ P \ \Longrightarrow \ vc \ C \ P'$
proof(*induction C arbitrary: P P'*)
case *Aseq* **thus** *?case* **by** *simp (metis pre_mono)*
qed *simp_all*

lemma *vc_complete*:
 $\vdash \{P\} c \{Q\} \Longrightarrow \exists C. \ strip \ C = c \ \wedge \ vc \ C \ Q \ \wedge \ (\forall s. \ P \ s \ \longrightarrow \ pre \ C \ Q \ s)$
(is $_ \Longrightarrow \exists C. \ ?G \ P \ c \ Q \ C$)
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

```

13.4 Hoare Logic for Total Correctness

```

theory Hoare_Total
imports Hoare_Examples
begin

```

13.4.1 Hoare Logic for Total Correctness — Separate Termination Relation

Note that this definition of total validity \models_t only works if execution is deterministic (which it is in our case).

definition hoare_tvalid :: $assn \Rightarrow com \Rightarrow assn \Rightarrow bool$

($\models_t \{(1-)\} / (-) / \{(1-)\} 50$) **where**
 $\models_t \{P\}c\{Q\} \iff (\forall s. P s \longrightarrow (\exists t. (c,s) \Rightarrow t \wedge Q t))$

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: $\vdash_t \{P\} \text{ SKIP } \{P\} \mid$

Assign: $\vdash_t \{\lambda s. P(s[a/x])\} x ::= a \{P\} \mid$

Seq: $\llbracket \vdash_t \{P_1\} c_1 \{P_2\}; \vdash_t \{P_2\} c_2 \{P_3\} \rrbracket \Longrightarrow \vdash_t \{P_1\} c_1;;c_2 \{P_3\} \mid$

If: $\llbracket \vdash_t \{\lambda s. P s \wedge \text{bval } b s\} c_1 \{Q\}; \vdash_t \{\lambda s. P s \wedge \neg \text{bval } b s\} c_2 \{Q\} \rrbracket$
 $\Longrightarrow \vdash_t \{P\} \text{ IF } b \text{ THEN } c_1 \text{ ELSE } c_2 \{Q\} \mid$

While:

($\wedge n::\text{nat}$.

$\vdash_t \{\lambda s. P s \wedge \text{bval } b s \wedge T s n\} c \{\lambda s. P s \wedge (\exists n' < n. T s n')\}$

$\Longrightarrow \vdash_t \{\lambda s. P s \wedge (\exists n. T s n)\} \text{ WHILE } b \text{ DO } c \{\lambda s. P s \wedge \neg \text{bval } b s\} \mid$

conseq: $\llbracket \forall s. P' s \longrightarrow P s; \vdash_t \{P\} c \{Q\}; \forall s. Q s \longrightarrow Q' s \rrbracket \Longrightarrow$
 $\vdash_t \{P'\} c \{Q'\}$

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 $T :: \text{state} \Rightarrow \text{nat} \Rightarrow \text{bool}$ decreases. The following functional version is more intuitive:

lemma *While_fun*:

$\llbracket \wedge n::\text{nat}. \vdash_t \{\lambda s. P s \wedge \text{bval } b s \wedge n = f s\} c \{\lambda s. P s \wedge f s < n\} \rrbracket$

$\Longrightarrow \vdash_t \{P\} \text{ WHILE } b \text{ DO } c \{\lambda s. P s \wedge \neg \text{bval } b s\}$

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'*: $\forall s. P s \longrightarrow Q(s[a/x]) \Longrightarrow \vdash_t \{P\} x ::= a \{Q\}$

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*: $\vdash_t \{P\} c \{Q\} \Longrightarrow \models_t \{P\} c \{Q\}$
proof (*unfold hoare_tvalid_def, induction rule: hoaret.induct*)
case (*While P b T c*)
{
fix $s\ n$
have $\llbracket P\ s; T\ s\ n \rrbracket \Longrightarrow \exists t. (WHILE\ b\ DO\ c, s) \Rightarrow t \wedge P\ t \wedge \neg bval\ b\ t$
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**
 $wpt\ c\ Q = (\lambda s. \exists t. (c, s) \Rightarrow t \wedge Q\ t)$

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  $\forall s. wp_t ?w Q s \longrightarrow wp_t ?w Q s \wedge (\exists n. Its b c s n)$ 
  unfolding wpt_def by (metis WHILE_Its)
  moreover
  { fix n
    let ?R =  $\lambda s'. wp_t ?w Q s' \wedge (\exists n' < n. ?T s' n')$ 
    { fix s t assume bval b s and ?T s n and  $(?w, s) \Rightarrow t$  and Q t
      from  $\langle bval b s \rangle$  and  $\langle (?w, s) \Rightarrow t \rangle$  obtain s' where
         $(c, s) \Rightarrow s' \langle ?w, s' \rangle \Rightarrow t$  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  $\langle (c, s) \Rightarrow s' \rangle$  have ?T s (Suc n') by (rule Its_Suc)
      with  $\langle ?T s n \rangle$  have  $n = Suc n'$  by (rule Its_fun)
      with  $\langle (c, s) \Rightarrow s' \rangle$  and  $\langle (?w, s') \Rightarrow t \rangle$  and  $\langle Q t \rangle$  and  $\langle ?T s' n' \rangle$ 
      have  $wp_t c ?R s$  by (auto simp: wpt_def)
    }
  }
  hence  $\forall s. wp_t ?w Q s \wedge bval b s \wedge ?T s n \longrightarrow wp_t c ?R s$ 
  unfolding wpt_def by auto

  note strengthen_pre[OF this While.IH]
  } note hoaret.While[OF this]
  moreover have  $\forall s. wp_t ?w Q s \wedge \neg bval b s \longrightarrow Q s$ 
  by (auto simp add:wpt_def)
  ultimately show ?case by (rule 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:  $\models_t \{P\}c\{Q\} \Longrightarrow \vdash_t \{P\}c\{Q\}$ 
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

theory *Complete_Lattice*
imports *Main*
begin

locale *Complete_Lattice* =
fixes $L :: 'a::order\ set$ **and** $Glb :: 'a\ set \Rightarrow 'a$
assumes $Glb_lower: A \subseteq L \Longrightarrow a \in A \Longrightarrow Glb\ A \leq a$
and $Glb_greatest: b : L \Longrightarrow \forall a \in A. b \leq a \Longrightarrow b \leq Glb\ A$
and $Glb_in_L: A \subseteq L \Longrightarrow Glb\ A : L$
begin

definition $lfp :: ('a \Rightarrow 'a) \Rightarrow 'a$ **where**
 $lfp\ f = Glb\ \{a : L. f\ a \leq a\}$

lemma *index_lfp*: $lfp\ f : L$
by (*auto simp: lfp_def intro: Glb_in_L*)

lemma *lfp_lowerbound*:
 $\llbracket a : L; f\ a \leq a \rrbracket \Longrightarrow lfp\ f \leq a$
by (*auto simp add: lfp_def intro: Glb_lower*)

lemma *lfp_greatest*:
 $\llbracket a : L; \bigwedge u. \llbracket u : L; f\ u \leq u \rrbracket \Longrightarrow a \leq u \rrbracket \Longrightarrow a \leq lfp\ f$
by (*auto simp add: lfp_def intro: Glb_greatest*)

lemma *lfp_unfold*: **assumes** $\bigwedge x. f\ x : L \longleftrightarrow x : L$
and *mono*: *mono f* **shows** $lfp\ f = f\ (lfp\ f)$
proof–

note *assms(1)[simp] index_lfp[simp]*
have $1: f\ (lfp\ f) \leq lfp\ f$
apply (*rule lfp_greatest*)
apply *simp*
by (*blast intro: lfp_lowerbound monoD[OF mono] order_trans*)
have $lfp\ f \leq f\ (lfp\ f)$

```

    by (fastforce intro: 1 monoD[OF mono] lfp_lowerbound)
  with 1 show ?thesis by (blast intro: order_antisym)
qed

end

end

```

```

theory ACom
imports Com
begin

```

14.1 Annotated Commands

```

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 \Rightarrow 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 \Rightarrow 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 \Rightarrow 'a) \Rightarrow nat \Rightarrow nat \Rightarrow 'a **where**

```

shift f n = ( $\lambda p. f(p+n)$ )

```

fun *annotate* :: (nat \Rightarrow 'a) \Rightarrow com \Rightarrow 'a acom **where**

$annotate\ f\ SKIP = SKIP\ \{f\ 0\} \mid$
 $annotate\ f\ (x ::= e) = x ::= e\ \{f\ 0\} \mid$
 $annotate\ f\ (c_1;;c_2) = annotate\ f\ c_1;;\ annotate\ (shift\ f\ (asize\ c_1))\ c_2 \mid$
 $annotate\ f\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) =$
 $\quad IF\ b\ THEN\ \{f\ 0\}\ annotate\ (shift\ f\ 1)\ c_1$
 $\quad ELSE\ \{f\ (asize\ c_1 + 1)\}\ annotate\ (shift\ f\ (asize\ c_1 + 2))\ c_2$
 $\quad \{f\ (asize\ c_1 + asize\ c_2 + 2)\} \mid$
 $annotate\ f\ (WHILE\ b\ DO\ c) =$
 $\quad \{f\ 0\}\ WHILE\ b\ DO\ \{f\ 1\}\ annotate\ (shift\ f\ 2)\ c\ \{f\ (asize\ c + 2)\}$

fun $annos :: 'a\ acom \Rightarrow 'a\ list$ **where**
 $annos\ (SKIP\ \{P\}) = [P] \mid$
 $annos\ (x ::= e\ \{P\}) = [P] \mid$
 $annos\ (C_1;;C_2) = annos\ C_1\ @\ annos\ C_2 \mid$
 $annos\ (IF\ b\ THEN\ \{P_1\}\ C_1\ ELSE\ \{P_2\}\ C_2\ \{Q\}) =$
 $\quad P_1\ \#\ annos\ C_1\ @\ P_2\ \#\ annos\ C_2\ @\ [Q] \mid$
 $annos\ (\{I\}\ WHILE\ b\ DO\ \{P\}\ C\ \{Q\}) = I\ \#\ P\ \#\ annos\ C\ @\ [Q]$
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**
 $map_acom\ f\ (SKIP\ \{P\}) = SKIP\ \{f\ P\} \mid$
 $map_acom\ f\ (x ::= e\ \{P\}) = x ::= e\ \{f\ P\} \mid$
 $map_acom\ f\ (C_1;;C_2) = map_acom\ f\ C_1;;\ map_acom\ f\ C_2 \mid$
 $map_acom\ f\ (IF\ b\ THEN\ \{P_1\}\ C_1\ ELSE\ \{P_2\}\ C_2\ \{Q\}) =$
 $\quad IF\ b\ THEN\ \{f\ P_1\}\ map_acom\ f\ C_1\ ELSE\ \{f\ P_2\}\ map_acom\ f\ C_2$
 $\quad \{f\ Q\} \mid$
 $map_acom\ f\ (\{I\}\ WHILE\ b\ DO\ \{P\}\ C\ \{Q\}) =$
 $\quad \{f\ I\}\ WHILE\ b\ DO\ \{f\ P\}\ map_acom\ f\ C\ \{f\ Q\}$

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 $C = \text{SKIP} \longleftrightarrow (\text{EX } P. C = \text{SKIP } \{P\})$
by (*cases* C) *simp_all*

lemma *strip_eq_Assign*:
strip $C = x ::= e \longleftrightarrow (\text{EX } P. C = x ::= e \{P\})$
by (*cases* C) *simp_all*

lemma *strip_eq_Seq*:
strip $C = c1 ;; c2 \longleftrightarrow (\text{EX } C1 C2. C = C1 ;; C2 \ \& \ \text{strip } C1 = c1 \ \& \ \text{strip } C2 = c2)$
by (*cases* C) *simp_all*

lemma *strip_eq_If*:
strip $C = \text{IF } b \ \text{THEN } c1 \ \text{ELSE } c2 \longleftrightarrow$
 $(\text{EX } P1 P2 C1 C2 Q. C = \text{IF } b \ \text{THEN } \{P1\} C1 \ \text{ELSE } \{P2\} C2 \ \{Q\} \ \& \ \text{strip } C1 = c1 \ \& \ \text{strip } C2 = c2)$
by (*cases* C) *simp_all*

lemma *strip_eq_While*:
strip $C = \text{WHILE } b \ \text{DO } c1 \longleftrightarrow$
 $(\text{EX } I P C1 Q. C = \{I\} \ \text{WHILE } b \ \text{DO } \{P\} C1 \ \{Q\} \ \& \ \text{strip } C1 = c1)$
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* $(\text{annos } (\text{annotate } (\lambda p. a) c)) = \{a\}$
by(*induction* c) *simp_all*

lemma *post_in_annos*: *post* $C \in \text{set}(\text{annos } C)$
by(*auto simp: post_def annos_ne*)

lemma *post_anno_asize*: *post* $C = \text{anno } C (\text{size}(\text{annos } C) - 1)$
by(*simp add: post_def last_conv_nth[OF annos_ne] anno_def*)

end
theory *Collecting*
imports *Complete_Lattice Big_Step ACom*
begin

14.2 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 Collecting Semantics of Commands

14.3.1 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:
   $C1 \leq C2 \iff \text{strip } C1 = \text{strip } C2 \wedge \text{list\_all2 } (op \leq) (\text{annos } C1) (\text{annos } C2)$ 
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

14.3.2 Collecting semantics

definition *step* = *Step* ($\lambda x e S. \{s(x := \text{aval } e s) \mid s. s : S\}$) ($\lambda b S. \{s:S. \text{bval } b s\}$)

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 $!!x e S1 S2. S1 \leq S2 \Longrightarrow f x e S1 \leq f x e S2$

$!!b S1 S2. S1 \leq S2 \Longrightarrow g b S1 \leq g b S2$

shows $C1 \leq C2 \Longrightarrow S1 \leq S2 \Longrightarrow \text{Step } f g S1 C1 \leq \text{Step } f g S2 C2$

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*: $\text{strip}(\text{step } S \ C) = \text{strip } C$
by (*induction C arbitrary: S*) (*auto simp: step_def*)

lemma *lfp_cs_unfold*: $\text{lfp } c \ (\text{step } S) = \text{step } S \ (\text{lfp } c \ (\text{step } S))$
apply(rule *lfp_unfold[OF _ mono_step]*)
apply(*simp add: strip_step*)
done

lemma *CS_unfold*: $CS \ c = \text{step } UNIV \ (CS \ c)$
by (*metis CS_def lfp_cs_unfold*)

lemma *strip_CS[simp]*: $\text{strip}(CS \ c) = c$
by(*simp add: CS_def index_lfp[simplified]*)

14.3.3 Relation to big-step semantics

lemma *asize_nz*: $\text{asize}(c::\text{com}) \neq 0$
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*: $\text{post}(\text{lfp } c \ f) = (\bigcap \{\text{post } C \mid C. \text{strip } C = c \wedge f \ C \leq C\})$
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.prem1 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.prem3 ⟨C = ⋂⟩
  have step P C' ≤ C' {s ∈ I. bval b s} ≤ P S ≤ I step (post C') C ≤
  C
  by (auto simp: step_def post_def)
  have step {s ∈ I. bval b s} C' ≤ C'
  by (rule order_trans[OF mono2_step[OF order_refl {s ∈ I. bval b s} ≤
  P] ⟨step P C' ≤ C'⟩])
  have s1: {s:I. bval b s} using ⟨s1 ∈ S⟩ ⟨S ⊆ I⟩ ⟨bval b s1⟩ by auto
  note s2_in_post_C' = WhileTrue.IH(1)[OF ⟨strip C' = c'⟩ this ⟨step {s
  ∈ I. bval b s} C' ≤ C'⟩]
  from WhileTrue.IH(2)[OF WhileTrue.prem1 s2_in_post_C' ⟨step (post
  C') C ≤ C'⟩]
  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*: $(c,s) \Rightarrow t \Longrightarrow t : \text{post}(CS \ c)$
by(simp add: CS_def big_step_lfp)

end
theory *Collecting_Examples*
imports *Collecting_Vars*
begin

14.4 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 ⇒ '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 ⇒ (vname*val)set set acom where
show_acom C =
  annotate (λp. (λs. (λx. (x, s x)) ‘ (vars_acom C)) ‘ anno C p) (strip C)
```

14.5 Examples

```
definition c0 = WHILE Less (V "x") (N 3)
              DO "x" ::= Plus (V "x") (N 2)
```

```
definition C0 :: state set acom where C0 = annotate (%p. {}) c0
```

Collecting semantics:

```
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
theory Abs_Int_Tests
imports Com
begin

```

14.6 Test Programs

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

theory Abs_Int0

imports Abs_Int_init

begin

14.7 Orderings

The basic type classes *order*, *semilattice_sup* and *order_top* are defined in *Main*, more precisely in theories *Orderings* and *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 ⊔ Some y = Some(x ⊔ y) |
```


$None \sqcup y = y \mid$
 $x \sqcup None = x$

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** $\top = \text{Some } \top$

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* ($\lambda p.$ *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*)

14.7.1 Pre-fixpoint iteration

definition *pf* :: (('a::order) \Rightarrow 'a) \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*(γ)[*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 $f C' \leq C'$ *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_inv*:

pf *f* *x* = *Some* *y* \implies ($\wedge x. P x \implies P(f x)$) \implies $P x \implies P y$

unfolding *pf_def* **by** (*blast* *intro: while_option_rule*)

lemma *strip_pfp*:

assumes $\wedge x. g(f x) = g x$ **and** *pf* *f* *x0* = *Some* *x* **shows** $g x = g x0$

using *pf_inv*[*OF* *assms*(2), **where** $P = \%x. g x = g x0$] *assms*(1) **by**

simp

14.8 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*: $a \leq b \Longrightarrow \gamma\ a \leq \gamma\ b$
 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'*: $i1 \in \gamma\ a1 \Longrightarrow i2 \in \gamma\ a2 \Longrightarrow i1+i2 \in \gamma(plus'\ a1\ a2)$

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 $mono_gamma_s: f1 \leq f2 \Longrightarrow \gamma_s\ f1 \subseteq \gamma_s\ f2$
by ($auto\ simp: le_fun_def\ \gamma_fun_def\ dest: mono_gamma$)

lemma $mono_gamma_o:$
 $S1 \leq S2 \Longrightarrow \gamma_o\ S1 \subseteq \gamma_o\ S2$
by ($induction\ S1\ S2\ rule: less_eq_option.induct$) ($simp_all\ add: mono_gamma_s$)

lemma $mono_gamma_c: C1 \leq C2 \Longrightarrow \gamma_c\ C1 \leq \gamma_c\ C2$
by ($simp\ add: less_eq_acom_def\ mono_gamma_o\ size_annos\ anno_map_acom\ size_annos_same$ [of $C1\ C2$])

Correctness:

lemma $aval'_correct: s : \gamma_s\ S \Longrightarrow aval\ a\ s : \gamma(aval'\ a\ S)$
by ($induct\ a$) ($auto\ simp: gamma_num'\ gamma_plus'\ \gamma_fun_def$)

lemma $in_gamma_update: \llbracket s : \gamma_s\ S; i : \gamma\ a \rrbracket \Longrightarrow s(x := i) : \gamma_s(S(x := a))$
by ($simp\ add: \gamma_fun_def$)

lemma $gamma_Step_subcomm:$
assumes $!!x\ e\ S. f1\ x\ e\ (\gamma_o\ S) \subseteq \gamma_o\ (f2\ x\ e\ S)\ !!b\ S. g1\ b\ (\gamma_o\ S) \subseteq \gamma_o\ (g2\ b\ S)$
shows $Step\ f1\ g1\ (\gamma_o\ S)\ (\gamma_c\ C) \leq \gamma_c\ (Step\ f2\ g2\ S\ C)$
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*: $pf\ (step'\ \top)\ (bot\ c) = Some\ C$

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

14.8.1 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*: $C \leq C' \implies step'\ \top\ C \leq step'\ \top\ C'$

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

```

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)

```

14.8.2 Termination

```

lemma pfp_termination:
fixes x0 :: 'a::order and m :: 'a ⇒ 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 ≤ 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 ≤ f x])
  show wf {(y,x). ((I x ∧ x ≤ f x) ∧ ¬ f x ≤ x) ∧ y = f x}
    by(rule wf_subset[OF wf_measure[of m]]) (auto simp: m I)
next
  show I x0 ∧ x0 ≤ f x0 using ⟨I x0⟩ ⟨x0 ≤ f x0⟩ by blast
next
  fix x assume I x ∧ x ≤ f x thus I(f x) ∧ f x ≤ f(f x)
    by (blast intro: I mono)
qed

```

```

lemma le_iff_le_annos: C1 ≤ C2  $\iff$ 
  strip C1 = strip C2 ∧ ( $\forall i < \text{size}(\text{annos } C1). \text{annos } C1 ! i \leq \text{annos } C2 ! i$ )
by(simp add: less_eq_acom_def anno_def)

```

```

locale Measure1_fun =
fixes m :: 'av::top ⇒ nat

```

```

fixes  $h :: \text{nat}$ 
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  $m\_s.h: finite\ X \Longrightarrow m\_s\ S\ X \leq h * card\ X$ 
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
 $m\_o\ (Some\ S)\ X = m\_s\ S\ X \mid$ 
 $m\_o\ None\ X = h * card\ X + 1$ 

lemma  $m\_o.h: finite\ X \Longrightarrow m\_o\ opt\ X \leq (h * card\ X + 1)$ 
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  $m :: 'av :: semilattice\_sup\_top \Rightarrow nat +$ 
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 vname set \Rightarrow bool (top'_on_s) **where**
top_on_st S X = ($\forall x \in X. S x = \top$)

fun *top_on_opt* :: 'av 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* :: 'av st option acom \Rightarrow vname set \Rightarrow bool (top'_on_c)
where
top_on_acom C X = ($\forall a \in \text{set}(\text{annos } C). \text{top_on_opt } a X$)

lemma *top_on_top*: *top_on_opt* \top X
by(*auto simp: top_option_def*)

lemma *top_on_bot*: *top_on_acom* (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 \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 e S. [*top_on_opt* S X; x \notin X; vars e \subseteq -X] \Longrightarrow *top_on_opt*
(f x e S) X
!!b S. *top_on_opt* S X \Longrightarrow vars b \subseteq -X \Longrightarrow *top_on_opt* (g b S) X

shows $\llbracket \text{vars } C \subseteq -X; \text{top_on_opt } S X; \text{top_on_acom } C X \rrbracket \Longrightarrow \text{top_on_acom}$
(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)

lemma *m1: $x \leq y \Longrightarrow m\ x \geq m\ y$*
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 EX x: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) $\Longrightarrow S1 = S2$ on -X $\Longrightarrow S1 < S2 \Longrightarrow m_s\ S1\ X > m_s\ S2\ X$*
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 $\Longrightarrow \text{top_on_opt } o1\ (-X) \Longrightarrow \text{top_on_opt } o2\ (-X) \Longrightarrow$*
 $o1 < o2 \Longrightarrow m_o\ o1\ X > m_o\ o2\ X$
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 $\Longrightarrow \text{top_on_opt } o1\ (-X) \Longrightarrow \text{top_on_opt } o2\ (-X) \Longrightarrow$*
 $o1 \leq o2 \Longrightarrow m_o\ o1\ X \geq m_o\ o2\ X$
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 < size(annos C2)  $\neg$  annos C2 ! i  $\leq$  annos C1 ! i
  have topo1: top_on_opt (annos C1 ! i) (- ?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_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': top_on_acom C (-vars C)  $\implies$  top_on_acom (step'  $\top$ 
C) (-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:  $\exists C. AI\ c = Some\ C$ 
unfolding AI_def
apply(rule pfp_termination[where  $I = \lambda C. top\_on\_acom\ C\ (-\ vars\ C)$ 
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

```

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 = ( $\%x. case\ map\_of\ ps\ x\ of\ None \Rightarrow \top \mid Some\ a \Rightarrow a$ )
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 [[*typedef_overloaded*]] — allow quotient types to depend on classes

```

quotient_type  $'a\ st = ('a::top)\ st\_rep / eq\_st$ 
morphisms rep_st St
by (metis eq_st_def equivpI reflpI sympI transpI)

```

```

lift_definition update :: ( $'a::top$ )  $st \Rightarrow vname \Rightarrow 'a \Rightarrow 'a\ st$ 
  is  $\lambda ps\ x\ a. (x,a)\#ps$ 
by(auto simp: eq_st_def)

```

lift_definition $fun :: ('a::top) st \Rightarrow vname \Rightarrow 'a$ **is** fun_rep
by($simp$ $add: eq_st_def$)

definition $show_st :: vname\ set \Rightarrow ('a::top) st \Rightarrow (vname * 'a) set$ **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 $\gamma_st :: (('a::top) \Rightarrow 'b\ set) \Rightarrow 'a\ st \Rightarrow (vname \Rightarrow 'b) set$ **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 (op  $\sqcup$ )
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 $\not\Leftarrow$ **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*]:
 $a1 \leq a2 \implies S1 \leq S2 \implies update\ S1\ x\ a1 \leq update\ S2\ x\ a2$
by *transfer (auto simp add: less_eq_st_rep_def)*

lemma *mono_fun*: $S1 \leq S2 \implies fun\ S1\ x \leq fun\ S2\ x$
by *transfer (simp add: less_eq_st_rep_iff)*

locale *Gamma_semilattice* = *Val_semilattice* **where** $\gamma = \gamma$
for $\gamma :: 'av :: semilattice_sup_top \Rightarrow val\ set$
begin

abbreviation $\gamma_s :: 'av\ st \Rightarrow state\ set$
where $\gamma_s == \gamma_st\ \gamma$

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 (*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:
   $x : \gamma\_option\ r\ u \iff (\exists u'. u = Some\ u' \wedge x : r\ u')$ 
by (cases u) auto

end

end

```

```

theory Abs_Int1
imports Abs_State
begin

```

14.9 Computable Abstract Interpretation

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:  $s : \gamma_s\ S \implies aval\ a\ s : \gamma(aval'\ a\ S)$ 
by (induction a) (auto simp: gamma_num' gamma_plus' \gamma_st_def)

```

```

lemma gamma_Step_subcomm: fixes C1 C2 :: 'a::semilattice_sup acom
  assumes  $!!x\ e\ S. f1\ x\ e\ (\gamma_o\ S) \subseteq \gamma_o\ (f2\ x\ e\ S)$ 
   $!!b\ S. g1\ b\ (\gamma_o\ S) \subseteq \gamma_o\ (g2\ b\ S)$ 
  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:  $\llbracket s : \gamma_s\ S; i : \gamma\ a \rrbracket \implies s(x := i) : \gamma_s(update\ S\ x\ a)$ 
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 $AI :: com \Rightarrow 'av\ st\ option\ acom\ option\ \mathbf{where}$

$AI\ c = pfp\ (step'\ \top)\ (bot\ c)$

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 $AI_correct: AI\ c = Some\ C \implies CS\ c \leq \gamma_c\ C$

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

14.9.1 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'*: $S1 \leq S2 \implies C1 \leq C2 \implies \text{step}' S1 C1 \leq \text{step}' S2 C2$

unfolding *step'_def*

by(*rule mono2_Step*) (*auto simp: mono_aval' split: option.split*)

lemma *mono_step'_top*: $C \leq C' \implies \text{step}' \top C \leq \text{step}' \top C'$

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

14.9.2 Termination

locale *Measure1* =

fixes $m :: 'av::\text{order_top} \Rightarrow \text{nat}$

fixes $h :: \text{nat}$

assumes $h: m\ x \leq h$

begin

definition $m_s :: 'av\ st \Rightarrow \text{vname set} \Rightarrow \text{nat } (m_s)$ **where**

$m_s\ S\ X = (\sum x \in X. m(\text{fun } S\ x))$

lemma $m_s.h: \text{finite } X \implies m_s\ S\ X \leq h * \text{card } X$

by(*simp add: m_s_def*) (*metis mult.commute of_nat_id sum_bounded_above[OF h]*)

definition $m_o :: 'av\ st\ \text{option} \Rightarrow \text{vname set} \Rightarrow \text{nat } (m_o)$ **where**

$m_o\ \text{opt } X = (\text{case opt of None} \Rightarrow h * \text{card } X + 1 \mid \text{Some } S \Rightarrow m_s\ S\ X)$

lemma $m_o.h: \text{finite } X \implies m_o\ \text{opt } X \leq (h * \text{card } X + 1)$

by(*auto simp add: m_o_def m_s_h le_SucI split: option.split dest:m_s.h*)

definition $m_c :: 'av\ st\ \text{option acom} \Rightarrow \text{nat } (m_c)$ **where**

$m_c\ C = \text{sum_list } (\text{map } (\lambda a. m_o\ a\ (\text{vars } C))\ (\text{annos } C))$

Upper complexity bound:

lemma $m_c.h: m_c\ C \leq \text{size}(\text{annos } C) * (h * \text{card}(\text{vars } C) + 1)$

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 ...  $\leq$  ( $\sum i < ?a. h * ?n + 1$ )
  apply(rule sum_mono) using m_o_h[OF finite_Cvars] by simp
also have ... = ?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 = ( $\forall x \in X. \text{fun } S \ x = \top$ )
```

```
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 = ( $\forall a \in \text{set}(annos C). \text{top\_on\_opt } a \ X$ )
```

```
lemma top_on_top: top_on_opt ( $\top :: \_ \text{ st option}$ ) 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** $EX\ x:X. S1\ x < S2\ x$
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 : $finite(X) \implies fun\ S1 = fun\ S2\ on\ -X$
 $\implies S1 < S2 \implies m_s\ S1\ X > m_s\ S2\ X$
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 (-X)  $\implies$  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_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 (-X)  $\implies$  top_on_opt o2 (-X)
 $\implies$ 
  o1 ≤ o2  $\implies$  m_o o1 X ≥ 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 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) 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  $m_o (\text{annos } C1 ! i) ?X > m_o (\text{annos } C2 ! i) ?X$  (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_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':  $\llbracket \text{top\_on\_acom } C (-\text{vars } C) \rrbracket \Longrightarrow \text{top\_on\_acom } (step'$ 
 $\top C) (-\text{vars } C)$ 
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 fpf_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

```

theory Abs_Int1_const
imports Abs_Int1
begin

```

14.10 Constant Propagation

```

datatype const = Const val | Any

```

```

fun  $\gamma\_const$  where
 $\gamma\_const (\text{Const } i) = \{i\}$  |

```

$\gamma_const (Any) = UNIV$

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) \Rightarrow Const(*i+j*) | _ \Rightarrow Any)*
by(*auto split: prod.split const.split*)

instantiation *const* :: *semilattice_sup_top*
begin

fun *less_eq_const* **where** $x \leq y = (y = Any \mid x=y)$

definition $x < (y::const) = (x \leq y \ \& \ \neg y \leq x)$

fun *sup_const* **where** $x \sqcup y = (if\ x=y\ then\ x\ else\ Any)$

definition $\top = Any$

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  $aval'\_const = aval'$ 
..

```

14.10.1 Tests

```

definition  $steps\ c\ i = (step\_const \top \wedge \wedge i)$  (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

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 Backward Analysis of Expressions

```

subclass (in bounded_lattice) semilattice_sup_top ..

```

```

locale Val_lattice_gamma = Gamma_semilattice where  $\gamma = \gamma$ 
  for  $\gamma :: 'av::bounded\_lattice \Rightarrow val\ set +$ 
assumes inter_gamma_subset_gamma_inf:
   $\gamma\ a1 \cap \gamma\ a2 \subseteq \gamma(a1 \sqcap a2)$ 
and gamma_bot[simp]:  $\gamma \perp = \{\}$ 
begin

lemma in_gamma_inf:  $x : \gamma\ a1 \Longrightarrow x : \gamma\ a2 \Longrightarrow x : \gamma(a1 \sqcap a2)$ 
by (metis IntI inter_gamma_subset_gamma_inf set_mp)

lemma gamma_inf:  $\gamma(a1 \sqcap a2) = \gamma\ a1 \cap \gamma\ a2$ 
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':  $test\_num' i a = (i : \gamma a)$ 
and inv_plus':  $inv\_plus' a a1 a2 = (a1', a2') \Longrightarrow$ 
   $i1 : \gamma a1 \Longrightarrow i2 : \gamma a2 \Longrightarrow i1+i2 : \gamma a \Longrightarrow i1 : \gamma a1' \wedge i2 : \gamma a2'$ 
and inv_less':  $inv\_less' (i1 < i2) a1 a2 = (a1', a2') \Longrightarrow$ 
   $i1 : \gamma a1 \Longrightarrow i2 : \gamma a2 \Longrightarrow i1 : \gamma a1' \wedge i2 : \gamma a2'$ 

```

```

locale Abs_Int_inv = Val_inv where  $\gamma = \gamma$ 
  for  $\gamma :: 'av::bounded\_lattice \Rightarrow val\ set$ 
begin

```

```

lemma in_gamma_sup_UpI:

```

```

   $s : \gamma_o S1 \vee s : \gamma_o S2 \Longrightarrow s : \gamma_o(S1 \sqcup S2)$ 

```

```

by (metis (hide_lams, no_types) sup_ge1 sup_ge2 mono_gamma_o subsetD)

```

```

fun aval'' ::  $aexp \Rightarrow 'av\ st\ option \Rightarrow 'av$  where
   $aval'' e\ None = \perp$  |
   $aval'' e\ (Some\ S) = aval'\ e\ S$ 

```

```

lemma aval''_correct:  $s : \gamma_o S \Longrightarrow aval\ a\ s : \gamma(aval'' a S)$ 

```

```

by(cases S)(auto simp add: aval'_correct split: option.splits)

```

14.11.1 Backward analysis

```

fun inv_aval' ::  $aexp \Rightarrow 'av \Rightarrow 'av\ st\ option \Rightarrow 'av\ st\ option$  where
   $inv\_aval' (N\ n) a\ S = (if\ test\_num'\ n\ a\ then\ S\ else\ None) |$ 
   $inv\_aval' (V\ x) a\ S = (case\ S\ of\ None \Rightarrow None | Some\ S \Rightarrow$ 
     $let\ a' = fun\ S\ x\ \square\ a\ in$ 
     $if\ a' = \perp\ then\ None\ else\ Some(update\ S\ x\ a')) |$ 
   $inv\_aval' (Plus\ e1\ e2) a\ S =$ 
   $(let\ (a1, a2) = inv\_plus'\ a\ (aval''\ e1\ S)\ (aval''\ e2\ S)$ 
   $in\ inv\_aval'\ e1\ a1\ (inv\_aval'\ e2\ a2\ S))$ 

```

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: $s : \gamma_o S \Longrightarrow \text{aval } e \text{ } s : \gamma a \Longrightarrow s : \gamma_o (\text{inv_aval}' e a S)$

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** $s : \gamma_s S'$ **using** (s : $\gamma_o S$)

by(auto simp: in_gamma_option_iff)

moreover hence $s x : \gamma (\text{fun } S' x)$

by(simp add: γ _st_def)

moreover have $s x : \gamma a$ **using** V(2) **by** simp

ultimately show ?case

by(simp add: Let_def γ _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: $s : \gamma_o S \Longrightarrow bv = \text{bval } b \text{ } s \Longrightarrow s : \gamma_o (\text{inv_bval}' b bv S)$

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($rule\ top_on_Step$)
 $(auto\ simp\ add: top_on_top\ top_on_inv_bval'\ split: option.split)$

14.11.2 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 $pfpr': step'\ \top\ C \leq C$ **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

```

14.11.3 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

```

```

theory Abs_Int2_ivl

```

```

imports Abs_Int2

```

```

begin

```

14.12 Interval Analysis

```

type_synonym eint = int extended

```

```

type_synonym eint2 = eint * eint

```

```

definition  $\gamma\_rep :: eint2 \Rightarrow int\ set$  where

```

```

 $\gamma\_rep\ p = (let\ (l,h) = p\ in\ \{i.\ l \leq Fin\ i \wedge Fin\ i \leq h\})$ 

```

```

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 equivI)(auto simp: reflp_def symp_def transp_def eq_ivl_def)

```

abbreviation $ivl_abbr :: eint \Rightarrow eint \Rightarrow ivl$ ($[-, -]$) **where**
 $[l,h] == abs_ivl(l,h)$

lift_definition $\gamma_ivl :: ivl \Rightarrow int$ set **is** γ_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: γ_rep_def)

lift_definition $num_ivl :: int \Rightarrow ivl$ **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 $is_empty_rep :: eint2 \Rightarrow bool$ **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 $is_empty_ivl :: ivl \Rightarrow bool$ **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 $empty_rep :: eint2$ **where** $empty_rep = (Pinf, Minf)$

lift_definition $empty_ivl :: ivl$ **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: γ_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 γ_rep_def split: if_splits prod.splits*)[1]

apply(*auto simp: is_empty_rep_def γ_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 ( $\neg x < Pinf$ ) = ( $x = Pinf$ )

```

```

by(simp add: not_less)

```

```

lemma [simp]: fixes x :: 'a::linorder extended shows ( $\neg Minf < x$ ) = ( $x = Minf$ )

```

```

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 γ_{inf_rep} : $\gamma_{rep}(inf_rep\ p1\ p2) = \gamma_{rep}\ p1 \cap \gamma_{rep}\ p2$
by(*auto simp: inf_rep_def* γ_{rep_cases} *split: prod.splits extended.splits*)

lift_definition *inf_ivl* :: *ivl* \Rightarrow *ivl* \Rightarrow *ivl* **is** *inf_rep*
by(*auto simp: γ_{inf_rep} eq_ivl_def*)

lemma γ_{inf} : $\gamma_{ivl}(iv1 \sqcap iv2) = \gamma_{ivl}\ iv1 \cap \gamma_{ivl}\ iv2$
by *transfer* (*rule γ_{inf_rep}*)

definition $\perp = empty_ivl$

instance

proof (*standard, goal_cases*)

case 1 **thus** ?*case* **by** (*simp add: γ_{inf} le_ivl_iff_subset*)
next

case 2 **thus** ?*case* **by** (*simp add: γ_{inf} le_ivl_iff_subset*)
next

case 3 **thus** ?*case* **by** (*simp add: γ_{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*: $[l1, h1] \leq [l2, h2] \iff$
 (*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_Minf*[*simp*]: $-x = \text{Minf} \iff x = \text{Pinf}$
by(*cases x*) *auto*

lemma *uminus_eq_Pinf*[*simp*]: $-x = \text{Pinf} \iff x = \text{Minf}$
by(*cases x*) *auto*

lemma *uminus_le_Fin_iff*: $-x \leq \text{Fin}(-y) \iff \text{Fin } y \leq (x::'a::\text{ordered_ab_group_add}$
extended)

by(*cases x*) *auto*

lemma *Fin_uminus_le_iff*: $\text{Fin}(-y) \leq -x \iff x \leq ((\text{Fin } y)::'a::\text{ordered_ab_group_add}$
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 γ_uminus_rep : $i : \gamma_rep\ p \implies -i \in \gamma_rep(uminus_rep\ p)$
by (*auto simp: uminus_rep_def γ_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 γ_rep_cases*)
(auto simp: Icc_eq_Icc split: extended.splits)

instance ..
end

lemma γ_uminus : $i : \gamma_ivl\ iv \implies -i \in \gamma_ivl(-\ iv)$
by *transfer (rule γ_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 γ_aboveI : $i \in \gamma_ivl\ iv \implies i \leq j \implies j \in \gamma_ivl(above\ iv)$
by *transfer*

(*auto simp add: above_rep_def γ _rep_cases is_empty_rep_def
split: extended.splits*)

lemma γ _belowI: $i : \gamma$ _ivl $iv \implies j \leq i \implies j : \gamma$ _ivl(*below* iv)
by *transfer*

(*auto simp add: below_rep_def γ _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 ($iv1 \sqcap (\text{below } iv2 - [1,1])$,
 $iv2 \sqcap (\text{above } iv1 + [1,1])$)
 else ($iv1 \sqcap \text{above } iv2$, $iv2 \sqcap \text{below } iv1$))

lemma *above_nice*: $\text{above}[l,h] = (\text{if } [l,h] = \perp \text{ then } \perp \text{ else } [l,\infty])$
unfolding *bot_ivl_def* **by** *transfer* (*simp add: above_rep_def eq_ivl_empty*)

lemma *below_nice*: $\text{below}[l,h] = (\text{if } [l,h] = \perp \text{ then } \perp \text{ else } [-\infty,h])$
unfolding *bot_ivl_def* **by** *transfer* (*simp add: below_rep_def eq_ivl_empty*)

lemma *add_mono_le_Fin*:

$\llbracket x1 \leq \text{Fin } y1; x2 \leq \text{Fin } y2 \rrbracket \implies x1 + x2 \leq \text{Fin } (y1 + (y2::'a::\text{ordered_ab_group_add}))$
by(*drule* (1) *add_mono*) *simp*

lemma *add_mono_Fin_le*:

$\llbracket \text{Fin } y1 \leq x1; \text{Fin } y2 \leq x2 \rrbracket \implies \text{Fin}(y1 + y2::'a::\text{ordered_ab_group_add})$
 $\leq x1 + x2$
by(*drule* (1) *add_mono*) *simp*

global interpretation *Val_semilattice*

where $\gamma = \gamma$ _ivl **and** $\text{num}' = \text{num_ivl}$ **and** $\text{plus}' = \text{op } +$

proof (*standard*, *goal_cases*)

case 1 **thus** ?*case* **by** *transfer* (*simp add: le_iff_subset*)

next

case 2 **show** ?*case* **by** *transfer* (*simp add: γ _rep_def*)

next

case 3 **show** ?*case* **by** *transfer* (*simp add: γ _rep_def*)

next

case 4 **thus** ?*case*

apply *transfer*

apply(*auto simp: γ _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  $plus' = op +$ 
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  $plus' = op +$ 
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  $plus' = op +$ 
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  $plus' = op +$   
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  $\top$   $\hat{\hat{}}$  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
```

```
theory Abs_Int3
```

```
imports Abs_Int2_ivl
```

```
begin
```

14.13 Widening and Narrowing

```
class widen =
```

```
fixes widen :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a (infix  $\nabla$  65)
```

```
class narrow =
```

```
fixes narrow :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a (infix  $\triangle$  65)
```



```

class wn = widen + narrow + order +
assumes widen1:  $x \leq x \nabla y$ 
assumes widen2:  $y \leq x \nabla y$ 
assumes narrow1:  $y \leq x \implies y \leq x \triangle y$ 
assumes narrow2:  $y \leq x \implies x \triangle y \leq x$ 
begin

lemma narrowid[simp]:  $x \triangle x = x$ 
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* (op ∇)
by(*auto simp: eq_st_def*)

lift_definition *narrow_st* :: 'a st \Rightarrow 'a st \Rightarrow 'a st **is** *map2_st_rep* (op Δ)
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 ∇ *x* = *x* |

x ∇ *None* = *x* |

(*Some* *x*) ∇ (*Some* *y*) = *Some*(*x* ∇ *y*)

fun *narrow_option* **where**

None Δ *x* = *None* |

x Δ *None* = *None* |

(*Some* *x*) Δ (*Some* *y*) = *Some*(*x* Δ *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 (4 y x) thus ?case
    by(induct x y rule: narrow_option.induct) (simp_all add: narrow2)
qed

end

definition map2_acom :: ('a ⇒ 'a ⇒ 'a) ⇒ 'a acom ⇒ 'a acom ⇒ 'a acom
where
map2_acom f C1 C2 = annotate (λp. f (anno C1 p) (anno C2 p)) (strip C1)

instantiation acom :: (widen)widen
begin
definition widen_acom = map2_acom (op ∇)
instance ..
end

instantiation acom :: (narrow)narrow
begin
definition narrow_acom = map2_acom (op Δ)
instance ..
end

lemma strip_map2_acom[simp]:
  strip C1 = strip C2 ⇒ strip(map2_acom f C1 C2) = strip C1
by(simp add: map2_acom_def)

lemma strip_widen_acom[simp]:
  strip C1 = strip C2 ⇒ strip(C1 ∇ C2) = strip C1
by(simp add: widen_acom_def)

lemma strip_narrow_acom[simp]:
  strip C1 = strip C2 ⇒ strip(C1 Δ C2) = strip C1
by(simp add: narrow_acom_def)

lemma narrow1_acom: C2 ≤ C1 ⇒ C2 ≤ C1 Δ (C2::'a::wn acom)
by(simp add: narrow_acom_def narrow1 map2_acom_def less_eq_acom_def size_annos)

lemma narrow2_acom: C2 ≤ C1 ⇒ C1 Δ (C2::'a::wn acom) ≤ 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 ∧ f p ≤ p ∧ p ≤ p0
  { fix p assume ?Q p
    note P = conjunct1[OF this] and 12 = conjunct2[OF this]
    note 1 = conjunct1[OF 12] and 2 = conjunct2[OF 12]
    let ?p' = p Δ f p
    have ?Q ?p'
    proof auto
      show P ?p' by (blast intro: P Pinv)
      have f ?p' ≤ f p by (rule mono[OF ⟨P (p Δ f p)⟩ P narrow2_acom[OF
1]])
      also have ... ≤ ?p' by (rule narrow1_acom[OF 1])
      finally show f ?p' ≤ ?p' .
      have ?p' ≤ p by (rule narrow2_acom[OF 1])
      also have p ≤ p0 by (rule 2)
      finally show ?p' ≤ 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 pfp_wn_pfp:
assumes mono: !!x1 x2:::wn acom. P x1 ⇒ P x2 ⇒ x1 ≤ x2 ⇒ f x1
≤ f x2
and Pinv: P x !!x. P x ⇒ P(f x)
  !!x1 x2. P x1 ⇒ P x2 ⇒ P(x1 ∇ x2)
  !!x1 x2. P x1 ⇒ P x2 ⇒ P(x1 Δ x2)
and pfp_wn: pfp_wn f x = Some p shows P p ∧ f p ≤ p
proof-
  from pfp_wn obtain p0
  where its: iter_widen f x = Some p0 iter_narrow f p0 = Some p
  by (auto simp: pfp_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_pfp[where P=P] mono Pinv(2,4) iter_widen_pfp
its)+
qed

```

```

lemma strip_pfp_wn:
  [ [ ∇ C. strip(f C) = strip C; pfp_wn f C = Some C' ] ⇒ strip C' = strip
C

```

by(*auto simp add: pfp_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' = op +*

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* = $((\lambda C. C \nabla step_ivl \top C) \hat{\wedge} n)$

definition *step_down_ivl n* = $((\lambda C. C \Delta step_ivl \top C) \hat{\wedge} n)$

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*:

$\llbracket top_on_acom\ C1\ X; strip\ C1 = strip\ C2; top_on_acom\ C2\ X \rrbracket$

$\implies \text{top_on_acom } (C1 \nabla C2 :: _ \text{ st option acom}) X$
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* $\neg X$ $\neg S2 x \leq S1 x$

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: [finite X; top_on_opt S1 (-X); top_on_opt S2 (-X);
 $\neg S2 \leq S1$ ]  $\implies$ 
   $m\_o (S1 \nabla S2) X < m\_o S1 X$ 
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 \Longrightarrow fun S1 = fun S2 on -X \Longrightarrow S2 \leq S1
 \Longrightarrow S1 Δ S2 < S1

$\Longrightarrow n_s(S1 \ \Delta \ S2) \ X < n_s \ S1 \ X$

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) \Longrightarrow top_on_opt S2 (-X) \Longrightarrow finite X

\Longrightarrow S2 \leq S1 \Longrightarrow S1 Δ S2 < S1 \Longrightarrow $n_o(S1 \ \Delta \ S2) \ X < n_o \ S1 \ X$

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 Δ C2 < C1 \implies n_c (C1 Δ 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** EX 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 ⟨P C⟩)
next
fix C **assume** P C \neg f C \leq C **thus** P (C ∇ f C) \wedge m (C ∇ 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** $EX C'. \text{iter_narrow } f C = \text{Some } C'$
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**
 $m_rep p = (\text{if } \text{is_empty_rep } p \text{ then } 3 \text{ else}$
let $(l, h) = p$ **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 $m_ivl_anti_mono$: $y \leq x \implies m_ivl x \leq m_ivl y$

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 γ_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*:
 $x \triangle y < x \implies n_ivl(x \triangle y) < n_ivl x$
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*
 γ_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** $plus' = op +$
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>.