

Abstract Interpretation of Annotated Commands

Tobias Nipkow

September 13, 2023

Abstract

This is the Isabelle formalization of the material described in the eponymous ITP paper [1]. It develops a generic abstract interpreter for a while-language, including widening and narrowing. The collecting semantics and the abstract interpreter operate on annotated commands: the program is represented as a syntax tree with the semantic information directly embedded, without auxiliary labels. The aim of the formalization is simplicity, not efficiency or precision. This is motivated by the inclusion of the material in a theorem prover based course on semantics. A similar (but more polished) development is covered in [2].

1 Complete Lattice (indexed)

```
theory Complete-Lattice-ix  
imports Main  
begin
```

A complete lattice is an ordered type where every set of elements has a greatest lower (and thus also a least upper) bound. Sets are the prototypical complete lattice where the greatest lower bound is intersection. Sometimes that set of all elements of a type is not a complete lattice although all elements of the same shape form a complete lattice, for example lists of the same length, where the list elements come from a complete lattice. We will have exactly this situation with annotated commands. This theory introduces a slightly generalised version of complete lattices where elements have an “index” and only the set of elements with the same index form a complete lattice; the type as a whole is a disjoint union of complete lattices. Because sets are not types, this requires a special treatment.

```
locale Complete-Lattice-ix =  
fixes L :: 'i  $\Rightarrow$  'a::order set  
and Glb :: 'i  $\Rightarrow$  'a set  $\Rightarrow$  'a  
assumes Glb-lower:  $A \subseteq L\ i \implies a \in A \implies (Glb\ i\ A) \leq a$   
and Glb-greatest:  $b : L\ i \implies \forall a \in A. b \leq a \implies b \leq (Glb\ i\ A)$   
and Glb-in-L:  $A \subseteq L\ i \implies Glb\ i\ A : L\ i$   
begin
```

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

lemma $index\text{-}lfp$: $lfp\ f\ i : L\ i$
by($auto\ simp$: $lfp\text{-}def$ $intro$: $Glb\text{-}in\text{-}L$)

lemma $lfp\text{-}lowerbound$:
 $\llbracket a : L\ i;\ f\ a \leq a \rrbracket \Longrightarrow lfp\ f\ i \leq a$
by ($auto\ simp\ add$: $lfp\text{-}def$ $intro$: $Glb\text{-}lower$)

lemma $lfp\text{-}greatest$:
 $\llbracket a : L\ i;\ \bigwedge u.\ \llbracket u : L\ i;\ f\ u \leq u \rrbracket \Longrightarrow a \leq u \rrbracket \Longrightarrow a \leq lfp\ f\ i$
by ($auto\ simp\ add$: $lfp\text{-}def$ $intro$: $Glb\text{-}greatest$)

lemma $lfp\text{-}unfold$: **assumes** $\bigwedge x\ i.\ f\ x : L\ i \longleftrightarrow x : L\ i$
and $mono$: $mono\ f$ **shows** $lfp\ f\ i = f\ (lfp\ f\ i)$

proof –

note $assms(1)[simp]$ $index\text{-}lfp[simp]$
have 1 : $f\ (lfp\ f\ i) \leq lfp\ f\ i$
apply($rule\ lfp\text{-}greatest$)
apply $simp$
by ($blast\ intro$: $lfp\text{-}lowerbound\ monoD[OF\ mono]$ $order\text{-}trans$)
have $lfp\ f\ i \leq f\ (lfp\ f\ i)$
by ($fastforce\ intro$: $1\ monoD[OF\ mono]$ $lfp\text{-}lowerbound$)
with 1 **show** $?thesis$ **by**($blast\ intro$: $order\text{-}antisym$)

qed

end

end

2 Annotated Commands

theory $ACom$
imports $HOL\text{-}IMP.Com$
begin

datatype $'a\ acom =$
 $SKIP\ 'a$ $(SKIP\ \{-\}\ 61) |$
 $Assign\ vname\ aexp\ 'a$ $((-\ ::=-\ /\ \{-\})\ [1000,\ 61,\ 0]\ 61) |$
 $Seq\ ('a\ acom)\ ('a\ acom)$ $(-;;/-\ [60,\ 61]\ 60) |$
 $If\ bexp\ ('a\ acom)\ ('a\ acom)\ 'a$
 $((IF\ -/\ THEN\ -/\ ELSE\ -//\{-\})\ [0,\ 0,\ 61,\ 0]\ 61) |$
 $While\ 'a\ bexp\ ('a\ acom)\ 'a$
 $((\{-\}/\ WHILE\ -/\ DO\ (-)\ /\{-\})\ [0,\ 0,\ 61,\ 0]\ 61)$

fun $post :: 'a\ acom \Rightarrow 'a$ **where**
 $post\ (SKIP\ \{P\}) = P |$

$post (x ::= e \{P\}) = P \mid$
 $post (c1;; c2) = post c2 \mid$
 $post (IF b THEN c1 ELSE c2 \{P\}) = P \mid$
 $post (\{Inv\} WHILE b DO c \{P\}) = P$

fun strip :: 'a acom \Rightarrow com where
 $strip (SKIP \{P\}) = com.SKIP \mid$
 $strip (x ::= e \{P\}) = (x ::= e) \mid$
 $strip (c1;;c2) = (strip c1;; strip c2) \mid$
 $strip (IF b THEN c1 ELSE c2 \{P\}) = (IF b THEN strip c1 ELSE strip c2) \mid$
 $strip (\{Inv\} WHILE b DO c \{P\}) = (WHILE b DO strip c)$

fun anno :: 'a \Rightarrow com \Rightarrow 'a acom where
 $anno a com.SKIP = SKIP \{a\} \mid$
 $anno a (x ::= e) = (x ::= e \{a\}) \mid$
 $anno a (c1;;c2) = (anno a c1;; anno a c2) \mid$
 $anno a (IF b THEN c1 ELSE c2) =$
 $(IF b THEN anno a c1 ELSE anno a c2 \{a\}) \mid$
 $anno a (WHILE b DO c) =$
 $(\{a\} WHILE b DO anno a c \{a\})$

fun annos :: 'a acom \Rightarrow 'a list where
 $annos (SKIP \{a\}) = [a] \mid$
 $annos (x::=e \{a\}) = [a] \mid$
 $annos (C1;;C2) = annos C1 @ annos C2 \mid$
 $annos (IF b THEN C1 ELSE C2 \{a\}) = a \# annos C1 @ annos C2 \mid$
 $annos (\{i\} WHILE b DO C \{a\}) = i \# a \# annos C$

fun map-acom :: ('a \Rightarrow 'b) \Rightarrow 'a acom \Rightarrow 'b acom where
 $map-acom f (SKIP \{P\}) = SKIP \{f P\} \mid$
 $map-acom f (x ::= e \{P\}) = (x ::= e \{f P\}) \mid$
 $map-acom f (c1;;c2) = (map-acom f c1;; map-acom f c2) \mid$
 $map-acom f (IF b THEN c1 ELSE c2 \{P\}) =$
 $(IF b THEN map-acom f c1 ELSE map-acom f c2 \{f P\}) \mid$
 $map-acom f (\{Inv\} WHILE b DO c \{P\}) =$
 $(\{f Inv\} WHILE b DO map-acom f c \{f P\})$

lemma post-map-acom[simp]: $post(map-acom f c) = f(post c)$
by (induction c) simp-all

lemma strip-acom[simp]: $strip (map-acom f c) = strip c$
by (induction c) auto

lemma map-acom-SKIP:
 $map-acom f c = SKIP \{S'\} \longleftrightarrow (\exists S. c = SKIP \{S\} \wedge S' = f S)$
by (cases c) auto

lemma map-acom-Assign:

$map\text{-}acom\ f\ c = x ::= e\ \{S'\} \longleftrightarrow (\exists S. c = x ::= e\ \{S\} \wedge S' = f\ S)$
by (*cases c*) *auto*

lemma *map-acom-Seq*:

$map\text{-}acom\ f\ c = c1';;c2' \longleftrightarrow$
 $(\exists c1\ c2. c = c1;;c2 \wedge map\text{-}acom\ f\ c1 = c1' \wedge map\text{-}acom\ f\ c2 = c2')$
by (*cases c*) *auto*

lemma *map-acom-If*:

$map\text{-}acom\ f\ c = IF\ b\ THEN\ c1'\ ELSE\ c2'\ \{S'\} \longleftrightarrow$
 $(\exists S\ c1\ c2. c = IF\ b\ THEN\ c1\ ELSE\ c2\ \{S\} \wedge map\text{-}acom\ f\ c1 = c1' \wedge map\text{-}acom\ f\ c2 = c2' \wedge S' = f\ S)$
by (*cases c*) *auto*

lemma *map-acom-While*:

$map\text{-}acom\ f\ w = \{I'\}\ WHILE\ b\ DO\ c'\ \{P'\} \longleftrightarrow$
 $(\exists I\ P\ c. w = \{I\}\ WHILE\ b\ DO\ c\ \{P\} \wedge map\text{-}acom\ f\ c = c' \wedge I' = f\ I \wedge P' = f\ P)$
by (*cases w*) *auto*

lemma *strip-anno[simp]*: $strip\ (anno\ a\ c) = c$
by(*induct c*) *simp-all*

lemma *strip-eq-SKIP*:

$strip\ c = com.SKIP \longleftrightarrow (\exists P. c = SKIP\ \{P\})$
by (*cases c*) *simp-all*

lemma *strip-eq-Assign*:

$strip\ c = x ::= e \longleftrightarrow (\exists P. c = x ::= e\ \{P\})$
by (*cases c*) *simp-all*

lemma *strip-eq-Seq*:

$strip\ c = c1;;c2 \longleftrightarrow (\exists d1\ d2. c = d1;;d2 \ \&\ strip\ d1 = c1 \ \&\ strip\ d2 = c2)$
by (*cases c*) *simp-all*

lemma *strip-eq-If*:

$strip\ c = IF\ b\ THEN\ c1\ ELSE\ c2 \longleftrightarrow$
 $(\exists d1\ d2\ P. c = IF\ b\ THEN\ d1\ ELSE\ d2\ \{P\} \ \&\ strip\ d1 = c1 \ \&\ strip\ d2 = c2)$
by (*cases c*) *simp-all*

lemma *strip-eq-While*:

$strip\ c = WHILE\ b\ DO\ c1 \longleftrightarrow$
 $(\exists I\ d1\ P. c = \{I\}\ WHILE\ b\ DO\ d1\ \{P\} \ \&\ strip\ d1 = c1)$
by (*cases c*) *simp-all*

lemma *set-annos-anno[simp]*: $set\ (annos\ (anno\ a\ C)) = \{a\}$
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 (auto simp: strip-eq-SKIP strip-eq-Assign strip-eq-Seq strip-eq-If strip-eq-While)
done

```

```

lemmas size-annos-same2 = eqTrueI[OF size-annos-same]

```

```

end

```

3 Collecting Semantics of Commands

```

theory Collecting
imports Complete-Lattice-ix ACom
begin

```

3.1 Annotated commands as a complete lattice

```

instantiation acom :: (order) order
begin

```

```

fun less-eq-acom :: ('a::order)acom  $\Rightarrow$  'a acom  $\Rightarrow$  bool where
(SKIP {S})  $\leq$  (SKIP {S'}) = (S  $\leq$  S') |
(x ::= e {S})  $\leq$  (x' ::= e' {S'}) = (x=x'  $\wedge$  e=e'  $\wedge$  S  $\leq$  S') |
(c1;;c2)  $\leq$  (c1';;c2') = (c1  $\leq$  c1'  $\wedge$  c2  $\leq$  c2') |
(IF b THEN c1 ELSE c2 {S})  $\leq$  (IF b' THEN c1' ELSE c2' {S'}) =
(b=b'  $\wedge$  c1  $\leq$  c1'  $\wedge$  c2  $\leq$  c2'  $\wedge$  S  $\leq$  S') |
({Inv} WHILE b DO c {P})  $\leq$  ({Inv'} WHILE b' DO c' {P'}) =
(b=b'  $\wedge$  c  $\leq$  c'  $\wedge$  Inv  $\leq$  Inv'  $\wedge$  P  $\leq$  P') |
less-eq-acom - - = False

```

```

lemma SKIP-le: SKIP {S}  $\leq$  c  $\longleftrightarrow$  ( $\exists$  S'. c = SKIP {S'}  $\wedge$  S  $\leq$  S')
by (cases c) auto

```

```

lemma Assign-le: x ::= e {S}  $\leq$  c  $\longleftrightarrow$  ( $\exists$  S'. c = x ::= e {S'}  $\wedge$  S  $\leq$  S')
by (cases c) auto

```

```

lemma Seq-le: c1;;c2  $\leq$  c  $\longleftrightarrow$  ( $\exists$  c1' c2'. c = c1';;c2'  $\wedge$  c1  $\leq$  c1'  $\wedge$  c2  $\leq$  c2')
by (cases c) auto

```

```

lemma If-le: IF b THEN c1 ELSE c2 {S}  $\leq$  c  $\longleftrightarrow$ 
( $\exists$  c1' c2' S'. c = IF b THEN c1' ELSE c2' {S'}  $\wedge$  c1  $\leq$  c1'  $\wedge$  c2  $\leq$  c2'  $\wedge$  S  $\leq$ 
S')
by (cases c) auto

```

```

lemma While-le: {Inv} WHILE b DO c {P}  $\leq$  w  $\longleftrightarrow$ 
( $\exists$  Inv' c' P'. w = {Inv'} WHILE b DO c' {P'}  $\wedge$  c  $\leq$  c'  $\wedge$  Inv  $\leq$  Inv'  $\wedge$  P  $\leq$ 
P')

```

P')
by (*cases w*) *auto*

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 *x*) **thus** ?*case* **by** (*induct x*) *auto*

next

case (3 *x y z*) **thus** ?*case*

apply(*induct x y arbitrary: z rule: less-eq-acom.induct*)

apply (*auto intro: le-trans simp: SKIP-le Assign-le Seq-le If-le While-le*)

done

next

case (4 *x y*) **thus** ?*case*

apply(*induct x y rule: less-eq-acom.induct*)

apply (*auto intro: le-antisym*)

done

qed

end

fun *sub*₁ :: 'a *acom* \Rightarrow 'a *acom* **where**

*sub*₁(*c1*;;*c2*) = *c1* |

*sub*₁(*IF b THEN c1 ELSE c2* {*S*}) = *c1* |

*sub*₁({*I*} *WHILE b DO c* {*P*}) = *c*

fun *sub*₂ :: 'a *acom* \Rightarrow 'a *acom* **where**

*sub*₂(*c1*;;*c2*) = *c2* |

*sub*₂(*IF b THEN c1 ELSE c2* {*S*}) = *c2*

fun *invar* :: 'a *acom* \Rightarrow 'a **where**

invar({*I*} *WHILE b DO c* {*P*}) = *I*

fun *lift* :: ('a *set* \Rightarrow 'b) \Rightarrow *com* \Rightarrow 'a *acom* *set* \Rightarrow 'b *acom*

where

lift F com.SKIP M = (*SKIP* {*F* (*post* ' *M*)}) |

lift F (*x ::= a*) *M* = (*x ::= a* {*F* (*post* ' *M*)}) |

lift F (*c1*;;*c2*) *M* =

lift F c1 (*sub*₁ ' *M*);; *lift F c2* (*sub*₂ ' *M*) |

lift F (*IF b THEN c1 ELSE c2*) *M* =

IF b THEN lift F c1 (*sub*₁ ' *M*) *ELSE lift F c2* (*sub*₂ ' *M*)

{*F* (*post* ' *M*)} |

lift F (*WHILE b DO c*) *M* =

{*F* (*invar* ' *M*)}

WHILE b DO lift F c (*sub*₁ ' *M*)

{ F (*post* ‘ M)}

global-interpretation *Complete-Lattice-ix* % c . { c' . *strip* $c' = c$ } *lift Inter*
proof (*standard,goal-cases*)
 case ($1 A - a$)
 have $a:A \implies \text{lift Inter } (\text{strip } a) A \leq a$
 proof(*induction a arbitrary: A*)
 case *Seq* **from** *Seq.prem*s **show** ?*case* **by**(*force intro!*: *Seq.IH*)
 next
 case *If* **from** *If.prem*s **show** ?*case* **by**(*force intro!*: *If.IH*)
 next
 case *While* **from** *While.prem*s **show** ?*case* **by**(*force intro!*: *While.IH*)
 qed *force+*
 with 1 **show** ?*case* **by** *auto*
next
 case ($2 b i A$)
 thus ?*case*
 proof(*induction b arbitrary: i A*)
 case *SKIP* **thus** ?*case* **by** (*force simp:SKIP-le*)
 next
 case *Assign* **thus** ?*case* **by** (*force simp:Assign-le*)
 next
 case *Seq* **from** *Seq.prem*s **show** ?*case*
 by (*force intro!*: *Seq.IH simp:Seq-le*)
 next
 case *If* **from** *If.prem*s **show** ?*case* **by** (*force simp: If-le intro!*: *If.IH*)
 next
 case *While* **from** *While.prem*s **show** ?*case*
 by(*fastforce simp: While-le intro: While.IH*)
 qed
next
 case ($3 A i$)
 have *strip*(*lift Inter* $i A$) = i
 proof(*induction i arbitrary: A*)
 case *Seq* **from** *Seq.prem*s **show** ?*case*
 by (*fastforce simp: strip-eq-Seq subset-iff intro!*: *Seq.IH*)
 next
 case *If* **from** *If.prem*s **show** ?*case*
 by (*fastforce intro!*: *If.IH simp: strip-eq-If*)
 next
 case *While* **from** *While.prem*s **show** ?*case*
 by(*fastforce intro: While.IH simp: strip-eq-While*)
 qed *auto*
 thus ?*case* **by** *auto*
qed

lemma *le-post*: $c \leq d \implies \text{post } c \leq \text{post } d$
by(*induction c d rule: less-eq-acom.induct*) *auto*

3.2 Collecting semantics

```

fun step :: state set  $\Rightarrow$  state set acom  $\Rightarrow$  state set acom where
step S (SKIP {P}) = (SKIP {S}) |
step S (x ::= e {P}) =
  (x ::= e {{s'.  $\exists s \in S. s' = s(x := \text{aval } e \ s)$ }}) |
step S (c1 ;; c2) = step S c1 ;; step (post c1) c2 |
step S (IF b THEN c1 ELSE c2 {P}) =
  IF b THEN step {s:S. bval b s} c1 ELSE step {s:S.  $\neg$  bval b s} c2
  {post c1  $\cup$  post c2} |
step S ({Inv} WHILE b DO c {P}) =
  {S  $\cup$  post c} WHILE b DO (step {s:Inv. bval b s} c) {{s:Inv.  $\neg$  bval b s}}

```

```

definition CS :: com  $\Rightarrow$  state set acom where
CS c = lfp (step UNIV) c

```

```

lemma mono2-step: c1  $\leq$  c2  $\implies$  S1  $\subseteq$  S2  $\implies$  step S1 c1  $\leq$  step S2 c2

```

```

proof(induction c1 c2 arbitrary: S1 S2 rule: less-eq-acom.induct)

```

```

  case 2 thus ?case by fastforce

```

```

next

```

```

  case 3 thus ?case by(simp add: le-post)

```

```

next

```

```

  case 4 thus ?case by(simp add: subset-iff)(metis le-post subsetD)+

```

```

next

```

```

  case 5 thus ?case by(simp add: subset-iff) (metis le-post subsetD)

```

```

qed auto

```

```

lemma mono-step: mono (step S)

```

```

by(blast intro: monoI mono2-step)

```

```

lemma strip-step: strip(step S c) = strip c

```

```

by (induction c arbitrary: S) auto

```

```

lemma lfp-cs-unfold: lfp (step S) c = step S (lfp (step S) c)

```

```

apply(rule lfp-unfold[OF - mono-step])

```

```

apply(simp add: strip-step)

```

```

done

```

```

lemma CS-unfold: CS c = step UNIV (CS c)

```

```

by (metis CS-def lfp-cs-unfold)

```

```

lemma strip-CS[simp]: strip(CS c) = c

```

```

by(simp add: CS-def index-lfp[simplified])

```

```

end

```

4 Abstract Interpretation Abstractly

```

theory Abs-Int0

```



```

imports
  HOL-Library.While-Combinator
  Collecting
begin

```

4.1 Orderings

```

class preord =
fixes le :: 'a ⇒ 'a ⇒ bool (infix ⊆ 50)
assumes le-refl[simp]: x ⊆ x
and le-trans: x ⊆ y ⇒ y ⊆ z ⇒ x ⊆ z
begin

```

definition *mono* **where** $\text{mono } f = (\forall x y. x \subseteq y \longrightarrow f x \subseteq f y)$

lemma *monoD*: $\text{mono } f \Longrightarrow x \subseteq y \Longrightarrow f x \subseteq f y$ **by**(simp add: mono-def)

lemma *mono-comp*: $\text{mono } f \Longrightarrow \text{mono } g \Longrightarrow \text{mono } (g \circ f)$
by(simp add: mono-def)

declare le-trans[trans]

end

Note: no antisymmetry. Allows implementations where some abstract element is implemented by two different values $x \neq y$ such that $x \subseteq y$ and $y \subseteq x$. Antisymmetry is not needed because we never compare elements for equality but only for \subseteq .

```

class SL-top = preord +
fixes join :: 'a ⇒ 'a ⇒ 'a (infixl ⊔ 65)
fixes Top :: 'a (⊤)
assumes join-ge1 [simp]: x ⊆ x ⊔ y
and join-ge2 [simp]: y ⊆ x ⊔ y
and join-least: x ⊆ z ⇒ y ⊆ z ⇒ x ⊔ y ⊆ z
and top[simp]: x ⊆ ⊤
begin

```

lemma *join-le-iff*[simp]: $x \sqcup y \subseteq z \longleftrightarrow x \subseteq z \wedge y \subseteq z$
by (metis join-ge1 join-ge2 join-least le-trans)

lemma *le-join-disj*: $x \subseteq y \vee x \subseteq z \Longrightarrow x \subseteq y \sqcup z$
by (metis join-ge1 join-ge2 le-trans)

end

```

instantiation fun :: (type, SL-top) SL-top
begin

```

definition $f \subseteq g = (\forall x. f x \subseteq g x)$

definition $f \sqcup g = (\lambda x. f x \sqcup g x)$

definition $\top = (\lambda x. \top)$

lemma *join-apply[simp]*: $(f \sqcup g) x = f x \sqcup g x$

by (*simp add: join-fun-def*)

instance

proof (*standard,goal-cases*)

case 2 **thus** ?*case* **by** (*metis le-fun-def preord-class.le-trans*)

qed (*simp-all add: le-fun-def Top-fun-def*)

end

instantiation *acom* :: (*preord*) *preord*

begin

fun *le-acom* :: (*a*::*preord*)*acom* \Rightarrow *a* *acom* \Rightarrow *bool* **where**

le-acom (*SKIP* {*S*}) (*SKIP* {*S'*}) = ($S \sqsubseteq S'$) |

le-acom ($x ::= e$ {*S*}) ($x' ::= e'$ {*S'*}) = ($x=x' \wedge e=e' \wedge S \sqsubseteq S'$) |

le-acom ($c1;;c2$) ($c1';;c2'$) = (*le-acom* *c1* *c1'* \wedge *le-acom* *c2* *c2'*) |

le-acom (*IF* *b* *THEN* *c1* *ELSE* *c2* {*S*}) (*IF* *b'* *THEN* *c1'* *ELSE* *c2'* {*S'*}) =

($b=b' \wedge$ *le-acom* *c1* *c1'* \wedge *le-acom* *c2* *c2'* \wedge $S \sqsubseteq S'$) |

le-acom ({*Inv*} *WHILE* *b* *DO* *c* {*P*}) ({*Inv'*} *WHILE* *b'* *DO* *c'* {*P'*}) =

($b=b' \wedge$ *le-acom* *c* *c'* \wedge $Inv \sqsubseteq Inv' \wedge P \sqsubseteq P'$) |

le-acom - - = *False*

lemma [*simp*]: *SKIP* {*S*} $\sqsubseteq c \longleftrightarrow (\exists S'. c = \text{SKIP } \{S'\} \wedge S \sqsubseteq S')$

by (*cases c*) *auto*

lemma [*simp*]: $x ::= e$ {*S*} $\sqsubseteq c \longleftrightarrow (\exists S'. c = x ::= e$ {*S'*} $\wedge S \sqsubseteq S')$

by (*cases c*) *auto*

lemma [*simp*]: $c1;;c2 \sqsubseteq c \longleftrightarrow (\exists c1' c2'. c = c1';;c2' \wedge c1 \sqsubseteq c1' \wedge c2 \sqsubseteq c2')$

by (*cases c*) *auto*

lemma [*simp*]: *IF* *b* *THEN* *c1* *ELSE* *c2* {*S*} $\sqsubseteq c \longleftrightarrow$

($\exists c1' c2' S'. c = \text{IF } b \text{ THEN } c1' \text{ ELSE } c2' \{S'\} \wedge c1 \sqsubseteq c1' \wedge c2 \sqsubseteq c2' \wedge S \sqsubseteq S'$)

by (*cases c*) *auto*

lemma [*simp*]: {*Inv*} *WHILE* *b* *DO* *c* {*P*} $\sqsubseteq w \longleftrightarrow$

($\exists Inv' c' P'. w = \{Inv'\} \text{ WHILE } b \text{ DO } c' \{P'\} \wedge c \sqsubseteq c' \wedge Inv \sqsubseteq Inv' \wedge P \sqsubseteq P'$)

by (*cases w*) *auto*

instance

proof (*standard,goal-cases*)

case (1 *x*) **thus** ?*case* **by** (*induct x*) *auto*

```

next
  case (2 x y z) thus ?case
  apply(induct x y arbitrary; z rule: le-acom.induct)
  apply (auto intro: le-trans)
  done
qed

end

```

4.1.1 Lifting

```

instantiation option :: (preord)preord
begin

```

```

fun le-option where
  Some x  $\sqsubseteq$  Some y = (x  $\sqsubseteq$  y) |
  None  $\sqsubseteq$  y = True |
  Some -  $\sqsubseteq$  None = False

```

```

lemma [simp]: (x  $\sqsubseteq$  None) = (x = None)
by (cases x) simp-all

```

```

lemma [simp]: (Some x  $\sqsubseteq$  u) = ( $\exists$  y. u = Some y & x  $\sqsubseteq$  y)
by (cases u) auto

```

```

instance
proof (standard,goal-cases)
  case (1 x) show ?case by(cases x, simp-all)
next
  case (2 x y z) thus ?case
  by(cases z, simp, cases y, simp, cases x, auto intro: le-trans)
qed

```

```

end

```

```

instantiation option :: (SL-top)SL-top
begin

```

```

fun join-option where
  Some x  $\sqcup$  Some y = Some(x  $\sqcup$  y) |
  None  $\sqcup$  y = y |
  x  $\sqcup$  None = x

```

```

lemma join-None2[simp]: x  $\sqcup$  None = x
by (cases x) simp-all

```

```

definition  $\top$  = Some  $\top$ 

```

```

instance

```

```

proof (standard,goal-cases)
  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)
next
  case (4 x) thus ?case by(cases x, simp-all add: Top-option-def)
qed

end

```

definition *bot-acom* :: *com* \Rightarrow (*a*::*SL-top*)*option* *acom* (\perp_c) **where**
 $\perp_c = \text{anno } \text{None}$

lemma *strip-bot-acom*[*simp*]: *strip*(\perp_c *c*) = *c*
by(*simp* add: *bot-acom-def*)

lemma *bot-acom*[*rule-format*]: *strip* $c' = c \longrightarrow \perp_c c \sqsubseteq c'$
apply(*induct* *c* *arbitrary*: *c'*)
apply (*simp-all* add: *bot-acom-def*)
apply(*induct-tac* *c'*)
apply *simp-all*
apply(*induct-tac* *c'*)
apply *simp-all*
apply(*induct-tac* *c'*)
apply *simp-all*
apply(*induct-tac* *c'*)
apply *simp-all*
apply(*induct-tac* *c'*)
apply *simp-all*
done

4.1.2 Post-fixed point iteration

definition
pf_p :: (*a*::*preord*) \Rightarrow *a* \Rightarrow *a* \Rightarrow *a* *option* **where**
pf_p *f* = *while-option-stop* ($\lambda x. \neg f x \sqsubseteq x$) *f*

lemma *pf_p-pf_p*: **assumes** *pf_p* *f* *x0* = *Some* *x* **shows** *f* *x* \sqsubseteq *x*
using *while-option-stop*[*OF* *assms*[*simplified pf_p-def*]] **by** *simp*

lemma *pf_p-least*:
assumes *mono*: $\bigwedge x y. x \sqsubseteq y \Longrightarrow f x \sqsubseteq f y$
and *f* *p* \sqsubseteq *p* **and** *x0* \sqsubseteq *p* **and** *pf_p* *f* *x0* = *Some* *x* **shows** *x* \sqsubseteq *p*
proof –
 { **fix** *x* **assume** *x* \sqsubseteq *p*
 hence *f* *x* \sqsubseteq *f* *p* **by**(*rule mono*)
 from *this* $\langle f p \sqsubseteq p \rangle$ **have** *f* *x* \sqsubseteq *p* **by**(*rule le-trans*)

```

}
thus  $x \sqsubseteq p$  using assms(2-) while-option-rule[where  $P = \%x. x \sqsubseteq p$ ]
  unfolding pfpc-def by blast
qed

```

definition

```

lpfpc :: (('a::SL-top)option acom  $\Rightarrow$  'a option acom)  $\Rightarrow$  com  $\Rightarrow$  'a option acom
option where
lpfpc f c = pfpc f ( $\perp_c$  c)

```

lemma *lpfpc-pfp*: *lpfpc* f c0 = Some c \Longrightarrow f c \sqsubseteq c
by(*simp* *add: pfp-pfp lpfpc-def*)

lemma *strip-pfp*:

```

assumes  $\bigwedge x. g(f x) = g x$  and pfpc f x0 = Some x shows g x = g x0
using assms while-option-rule[where  $P = \%x. g x = g x0$  and c = f]
unfolding pfpc-def by metis

```

lemma *strip-lpfpc*: **assumes** $\bigwedge c. \text{strip}(f c) = \text{strip } c$ **and** *lpfpc* f c = Some c'
shows *strip* c' = c
using *assms(1)* *strip-pfp*[OF - *assms(2)*][*simplified lpfpc-def*]]
by(*metis* *strip-bot-acom*)

lemma *lpfpc-least*:

```

assumes mono:  $\bigwedge x y. x \sqsubseteq y \Longrightarrow f x \sqsubseteq f y$ 
and strip p = c0 and f p  $\sqsubseteq$  p and lp: lpfpc f c0 = Some c shows c  $\sqsubseteq$  p
using pfpc-least[OF - - bot-acom[OF  $\langle$ strip p = c0 $\rangle$ ] lp][simplified lpfpc-def]]
  mono  $\langle$ f p  $\sqsubseteq$  p $\rangle$ 
by blast

```

4.2 Abstract Interpretation

definition γ -fun :: ('a \Rightarrow 'b set) \Rightarrow ('c \Rightarrow 'a) \Rightarrow ('c \Rightarrow 'b) set **where**
 γ -fun γ F = {f. $\forall x. f x \in \gamma(F x)$ }

fun γ -option :: ('a \Rightarrow 'b set) \Rightarrow 'a option \Rightarrow 'b set **where**
 γ -option γ None = {} |
 γ -option γ (Some a) = γ a

The interface for abstract values:

locale *Val-abs* =

fixes γ :: 'av::SL-top \Rightarrow val set

assumes *mono-gamma*: a \sqsubseteq b \Longrightarrow γ a \subseteq γ b

and *gamma-Top*[*simp*]: $\gamma \top = UNIV$

fixes *num'* :: val \Rightarrow 'av

and *plus'* :: 'av \Rightarrow 'av \Rightarrow 'av

assumes *gamma-num'*: n : $\gamma(\text{num}' n)$

and *gamma-plus'*:

n1 : γ a1 \Longrightarrow n2 : γ a2 \Longrightarrow n1+n2 : $\gamma(\text{plus}' a1 a2)$

type-synonym $'av\ st = (vname \Rightarrow 'av)$

locale *Abs-Int-Fun* = *Val-abs* γ **for** $\gamma :: 'av::SL\text{-top} \Rightarrow val\ set$
begin

fun *aval'* :: $aexp \Rightarrow 'av\ st \Rightarrow 'av$ **where**
aval' (*N* *n*) *S* = *num'* *n* |
aval' (*V* *x*) *S* = *S* *x* |
aval' (*Plus* *a1* *a2*) *S* = *plus'* (*aval'* *a1* *S*) (*aval'* *a2* *S*)

fun *step'* :: $'av\ st\ option \Rightarrow 'av\ st\ option\ acom \Rightarrow 'av\ st\ option\ acom$
where
step' *S* (*SKIP* {*P*}) = (*SKIP* {*S*}) |
step' *S* (*x* ::= *e* {*P*}) =
 $x ::= e \{case\ S\ of\ None \Rightarrow None \mid Some\ S \Rightarrow Some(S(x ::= aval' e S))\}$ |
step' *S* (*c1* ;; *c2*) = *step'* *S* *c1* ;; *step'* (*post* *c1*) *c2* |
step' *S* (*IF* *b* *THEN* *c1* *ELSE* *c2* {*P*}) =
 $IF\ b\ THEN\ step'\ S\ c1\ ELSE\ step'\ S\ c2\ \{post\ c1\ \sqcup\ post\ c2\}$ |
step' *S* ({*Inv*} *WHILE* *b* *DO* *c* {*P*}) =
 $\{S\ \sqcup\ post\ c\}\ WHILE\ b\ DO\ (step'\ Inv\ c)\ \{Inv\}$

definition *AI* :: $com \Rightarrow 'av\ st\ option\ acom\ option$ **where**
AI = *lfp_c* (*step'* \top)

lemma *strip-step'[simp]*: $strip(step'\ S\ c) = strip\ c$
by(*induct* *c* *arbitrary*: *S*) (*simp-all* *add*: *Let-def*)

abbreviation $\gamma_f :: 'av\ st \Rightarrow state\ set$
where $\gamma_f == \gamma\text{-fun}\ \gamma$

abbreviation $\gamma_o :: 'av\ st\ option \Rightarrow state\ set$
where $\gamma_o == \gamma\text{-option}\ \gamma_f$

abbreviation $\gamma_c :: 'av\ st\ option\ acom \Rightarrow state\ set\ acom$
where $\gamma_c == map\text{-acom}\ \gamma_o$

lemma *gamma-f-Top[simp]*: $\gamma_f\ Top = UNIV$
by(*simp* *add*: *Top-fun-def* $\gamma\text{-fun-def}$)

lemma *gamma-o-Top[simp]*: $\gamma_o\ Top = UNIV$
by (*simp* *add*: *Top-option-def*)

lemma *mono-gamma-f*: $f \sqsubseteq g \Longrightarrow \gamma_f\ f \subseteq \gamma_f\ g$
by(*auto* *simp*: *le-fun-def* $\gamma\text{-fun-def}$ *dest*: *mono-gamma*)

lemma *mono-gamma-o*:
 $sa \sqsubseteq sa' \implies \gamma_o sa \subseteq \gamma_o sa'$
by(*induction sa sa' rule: le-option.induct*)(*simp-all add: mono-gamma-f*)

lemma *mono-gamma-c*: $ca \sqsubseteq ca' \implies \gamma_c ca \leq \gamma_c ca'$
by (*induction ca ca' rule: le-acom.induct*) (*simp-all add: mono-gamma-o*)

Soundness:

lemma *aval'-sound*: $s : \gamma_f S \implies \text{aval } a \ s : \gamma(\text{aval}' \ a \ S)$
by (*induct a*) (*auto simp: gamma-num' gamma-plus' gamma-fun-def*)

lemma *in-gamma-update*:
 $\llbracket s : \gamma_f S; i : \gamma a \rrbracket \implies s(x := i) : \gamma_f(S(x := a))$
by(*simp add: gamma-fun-def*)

lemma *step-preserves-le*:
 $\llbracket S \subseteq \gamma_o S'; c \leq \gamma_c c' \rrbracket \implies \text{step } S \ c \leq \gamma_c (\text{step}' \ S' \ c')$
proof(*induction c arbitrary: c' S S'*)
case *SKIP* **thus** *?case* **by**(*auto simp: SKIP-le map-acom-SKIP*)
next
case *Assign* **thus** *?case*
by (*fastforce simp: Assign-le map-acom-Assign intro: aval'-sound in-gamma-update split: option.splits del: subsetD*)
next
case *Seq* **thus** *?case* **apply** (*auto simp: Seq-le map-acom-Seq*)
by (*metis le-post post-map-acom*)
next
case (*If b c1 c2 P*)
then obtain $c1' \ c2' \ P'$ **where**
 $c' = \text{IF } b \ \text{THEN } c1' \ \text{ELSE } c2' \ \{P'\}$
 $P \subseteq \gamma_o P' \ c1 \leq \gamma_c c1' \ c2 \leq \gamma_c c2'$
by (*fastforce simp: If-le map-acom-If*)
moreover have $\text{post } c1 \subseteq \gamma_o(\text{post } c1' \sqcup \text{post } c2')$
by (*metis (no-types) c1 \le \gamma_c c1' \ join-ge1 le-post mono-gamma-o order-trans post-map-acom*)
moreover have $\text{post } c2 \subseteq \gamma_o(\text{post } c1' \sqcup \text{post } c2')$
by (*metis (no-types) c2 \le \gamma_c c2' \ join-ge2 le-post mono-gamma-o order-trans post-map-acom*)
ultimately show *?case* **using** $\langle S \subseteq \gamma_o S' \rangle$ **by** (*simp add: If.IH subset-iff*)
next
case (*While I b c1 P*)
then obtain $c1' \ I' \ P'$ **where**
 $c' = \{I'\} \ \text{WHILE } b \ \text{DO } c1' \ \{P'\}$
 $I \subseteq \gamma_o I' \ P \subseteq \gamma_o P' \ c1 \leq \gamma_c c1'$
by (*fastforce simp: map-acom-While While-le*)
moreover have $S \cup \text{post } c1 \subseteq \gamma_o(S' \sqcup \text{post } c1')$
using $\langle S \subseteq \gamma_o S' \rangle$ *le-post[OF c1 \le \gamma_c c1', simplified]*
by (*metis (no-types) join-ge1 join-ge2 le-sup-iff mono-gamma-o order-trans*)
ultimately show *?case* **by** (*simp add: While.IH subset-iff*)

qed

lemma *AI-sound*: $AI\ c = Some\ c' \implies CS\ c \leq \gamma_c\ c'$
proof(*simp add: CS-def AI-def*)
 assume *1*: $lpfp_c\ (step'\ \top)\ c = Some\ c'$
 have *2*: $step'\ \top\ c' \sqsubseteq c'$ **by**(*rule lpfp-pfp[OF 1]*)
 have *3*: $strip\ (\gamma_c\ (step'\ \top\ c')) = c$
 by(*simp add: strip-lpfp[OF - 1]*)
 have $lfp\ (step\ UNIV)\ c \leq \gamma_c\ (step'\ \top\ c')$
 proof(*rule lfp-lowerbound[simplified,OF 3]*)
 show $step\ UNIV\ (\gamma_c\ (step'\ \top\ c')) \leq \gamma_c\ (step'\ \top\ c')$
 proof(*rule step-preserves-le[OF - -]*)
 show $UNIV \subseteq \gamma_o\ \top$ **by** *simp*
 show $\gamma_c\ (step'\ \top\ c') \leq \gamma_c\ c'$ **by**(*rule mono-gamma-c[OF 2]*)
 qed
 qed
 with *2* **show** $lfp\ (step\ UNIV)\ c \leq \gamma_c\ c'$
 by (*blast intro: mono-gamma-c order-trans*)
qed
end

4.2.1 Monotonicity

lemma *mono-post*: $c \sqsubseteq c' \implies post\ c \sqsubseteq post\ c'$
by(*induction c c' rule: le-acom.induct*) (*auto*)

locale *Abs-Int-Fun-mono* = *Abs-Int-Fun* +
assumes *mono-plus'*: $a1 \sqsubseteq b1 \implies a2 \sqsubseteq b2 \implies plus'\ a1\ a2 \sqsubseteq plus'\ b1\ b2$
begin

lemma *mono-aval'*: $S \sqsubseteq S' \implies aval'\ e\ S \sqsubseteq aval'\ e\ S'$
by(*induction e*)(*auto simp: le-fun-def mono-plus'*)

lemma *mono-update*: $a \sqsubseteq a' \implies S \sqsubseteq S' \implies S(x := a) \sqsubseteq S'(x := a')$
by(*simp add: le-fun-def*)

lemma *mono-step'*: $S \sqsubseteq S' \implies c \sqsubseteq c' \implies step'\ S\ c \sqsubseteq step'\ S'\ c'$
apply(*induction c c' arbitrary: S S' rule: le-acom.induct*)
apply (*auto simp: Let-def mono-update mono-aval' mono-post le-join-disj split: option.split*)
done
end

Problem: not executable because of the comparison of abstract states, i.e. functions, in the post-fixedpoint computation.

end

5 Abstract State with Computable Ordering

```

theory Abs-State
imports Abs-Int0
        HOL-Library.Char-ord HOL-Library.List-Lexorder

begin

  A concrete type of state with computable  $\sqsubseteq$ :
datatype 'a st = FunDom vname  $\Rightarrow$  'a vname list

fun fun where fun (FunDom f xs) = f
fun dom where dom (FunDom f xs) = xs

definition [simp]: inter-list xs ys = [x $\leftarrow$ xs. x  $\in$  set ys]

definition show-st S = [(x,fun S x). x  $\leftarrow$  sort(dom S)]

definition show-acom = map-acom (map-option show-st)
definition show-acom-opt = map-option show-acom

definition lookup F x = (if x : set(dom F) then fun F x else  $\top$ )

definition update F x y =
  FunDom ((fun F)(x:=y)) (if x  $\in$  set(dom F) then dom F else x  $\#$  dom F)

lemma lookup-update: lookup (update S x y) = (lookup S)(x:=y)
by(rule ext)(auto simp: lookup-def update-def)

definition  $\gamma$ -st  $\gamma$  F = {f.  $\forall x. f x \in \gamma(\text{lookup } F x)$ }

instantiation st :: (SL-top) SL-top
begin

definition le-st F G = ( $\forall x \in \text{set}(\text{dom } G). \text{lookup } F x \sqsubseteq \text{fun } G x$ )

definition
join-st F G =
  FunDom ( $\lambda x. \text{fun } F x \sqcup \text{fun } G x$ ) (inter-list (dom F) (dom G))

definition  $\top$  = FunDom ( $\lambda x. \top$ ) []

instance
proof (standard, goal-cases)
  case 2 thus ?case
    apply(auto simp: le-st-def)
    by (metis lookup-def preord-class.le-trans top)
qed (auto simp: le-st-def lookup-def join-st-def Top-st-def)

end

```

lemma *mono-lookup*: $F \sqsubseteq F' \implies \text{lookup } F \ x \sqsubseteq \text{lookup } F' \ x$
by(*auto simp add: lookup-def le-st-def*)

lemma *mono-update*: $a \sqsubseteq a' \implies S \sqsubseteq S' \implies \text{update } S \ x \ a \sqsubseteq \text{update } S' \ x \ a'$
by(*auto simp add: le-st-def lookup-def update-def*)

locale *Gamma* = *Val-abs* **where** $\gamma = \gamma$ **for** $\gamma :: 'av :: SL\text{-top} \Rightarrow \text{val set}$
begin

abbreviation $\gamma_f :: 'av \text{ st} \Rightarrow \text{state set}$
where $\gamma_f == \gamma\text{-st } \gamma$

abbreviation $\gamma_o :: 'av \text{ st option} \Rightarrow \text{state set}$
where $\gamma_o == \gamma\text{-option } \gamma_f$

abbreviation $\gamma_c :: 'av \text{ st option acom} \Rightarrow \text{state set acom}$
where $\gamma_c == \text{map-acom } \gamma_o$

lemma *gamma-f-Top[simp]*: $\gamma_f \text{ Top} = \text{UNIV}$
by(*auto simp: Top-st-def $\gamma\text{-st-def}$ lookup-def*)

lemma *gamma-o-Top[simp]*: $\gamma_o \text{ Top} = \text{UNIV}$
by (*simp add: Top-option-def*)

lemma *mono-gamma-f*: $f \sqsubseteq g \implies \gamma_f f \subseteq \gamma_f g$
apply(*simp add: $\gamma\text{-st-def}$ subset-iff lookup-def le-st-def split: if-splits*)
by (*metis UNIV-I mono-gamma gamma-Top subsetD*)

lemma *mono-gamma-o*:
 $sa \sqsubseteq sa' \implies \gamma_o sa \subseteq \gamma_o sa'$
by(*induction sa sa' rule: le-option.induct*)(*simp-all add: mono-gamma-f*)

lemma *mono-gamma-c*: $ca \sqsubseteq ca' \implies \gamma_c ca \leq \gamma_c ca'$
by (*induction ca ca' rule: le-acom.induct*) (*simp-all add: mono-gamma-o*)

lemma *in-gamma-option-iff*:
 $x : \gamma\text{-option } r \ u \longleftrightarrow (\exists u'. u = \text{Some } u' \wedge x : r \ u')$
by (*cases u*) *auto*

end

end

6 Computable Abstract Interpretation

theory *Abs-Int1*

imports *Abs-State*
begin

Abstract interpretation over type *st* instead of functions.

context *Gamma*
begin

fun *aval'* :: *aexp* \Rightarrow '*av st* \Rightarrow '*av* **where**
aval' (*N n*) *S* = *num'* *n* |
aval' (*V x*) *S* = *lookup S x* |
aval' (*Plus a1 a2*) *S* = *plus'* (*aval'* *a1 S*) (*aval'* *a2 S*)

lemma *aval'-sound*: *s* : $\gamma_f S \Longrightarrow \text{aval } a \text{ s} : \gamma(\text{aval}' a S)$
by (*induction a*) (*auto simp: gamma-num' gamma-plus' γ -st-def lookup-def*)

end

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* = *Gamma* **where** $\gamma = \gamma$ **for** $\gamma :: 'av :: SL\text{-top} \Rightarrow \text{val set}$
begin

fun *step'* :: '*av st option* \Rightarrow '*av st option acom* \Rightarrow '*av st option acom* **where**
step' *S* (*SKIP* {*P*}) = (*SKIP* {*S*}) |
step' *S* (*x ::= e* {*P*}) =
x ::= e {*case S of None* \Rightarrow *None* | *Some S* \Rightarrow *Some(update S x (aval' e S))*} |
step' *S* (*c1;; c2*) = *step'* *S* *c1;; step'* (*post c1*) *c2* |
step' *S* (*IF b THEN c1 ELSE c2* {*P*}) =
(*let c1' = step'* *S* *c1*; *c2' = step'* *S* *c2*
in *IF b THEN c1' ELSE c2'* {*post c1* \sqcup *post c2*}) |
step' *S* ({*Inv*} *WHILE b DO c* {*P*}) =
{*S* \sqcup *post c*} *WHILE b DO step'* *Inv c* {*Inv*}

definition *AI* :: *com* \Rightarrow '*av st option acom option* **where**
AI = *lfp_c* (*step'* \top)

lemma *strip-step'[simp]*: *strip(step' S c)* = *strip c*
by (*induct c arbitrary: S*) (*simp-all add: Let-def*)

Soundness:

lemma *in-gamma-update*:
 $\llbracket s : \gamma_f S; i : \gamma a \rrbracket \Longrightarrow s(x := i) : \gamma_f(\text{update } S x a)$
by (*simp add: γ -st-def lookup-update*)

The soundness proofs are textually identical to the ones for the step function operating on states as functions.

lemma *step-preserves-le*:
 $\llbracket S \subseteq \gamma_o S'; c \leq \gamma_c c' \rrbracket \Longrightarrow \text{step } S c \leq \gamma_c (\text{step}' S' c')$

```

proof(induction c arbitrary: c' S S')
  case SKIP thus ?case by(auto simp:SKIP-le map-acom-SKIP)
next
  case Assign thus ?case
    by (fastforce simp: Assign-le map-acom-Assign intro: aval'-sound in-gamma-update
      split: option.splits del:subsetD)
next
  case Seq thus ?case apply (auto simp: Seq-le map-acom-Seq)
    by (metis le-post post-map-acom)
next
  case (If b c1 c2 P)
    then obtain c1' c2' P' where
      c' = IF b THEN c1' ELSE c2' {P'}
      P ⊆ γo P' c1 ≤ γc c1' c2 ≤ γc c2'
    by (fastforce simp: If-le map-acom-If)
    moreover have post c1 ⊆ γo(post c1' ⊔ post c2')
      by (metis (no-types) ‹c1 ≤ γc c1'› join-ge1 le-post mono-gamma-o order-trans
        post-map-acom)
    moreover have post c2 ⊆ γo(post c1' ⊔ post c2')
      by (metis (no-types) ‹c2 ≤ γc c2'› join-ge2 le-post mono-gamma-o order-trans
        post-map-acom)
    ultimately show ?case using ‹S ⊆ γo S'› by (simp add: If.IH subset-iff)
next
  case (While I b c1 P)
    then obtain c1' I' P' where
      c' = {I'} WHILE b DO c1' {P'}
      I ⊆ γo I' P ⊆ γo P' c1 ≤ γc c1'
    by (fastforce simp: map-acom-While While-le)
    moreover have S ∪ post c1 ⊆ γo (S' ⊔ post c1')
      using ‹S ⊆ γo S'› le-post[OF ‹c1 ≤ γc c1'›, simplified]
    by (metis (no-types) join-ge1 join-ge2 le-sup-iff mono-gamma-o order-trans)
    ultimately show ?case by (simp add: While.IH subset-iff)
qed

```

```

lemma AI-sound: AI c = Some c' ⇒ CS c ≤ γc c'
proof(simp add: CS-def AI-def)
  assume 1: lfpc (step' ⊔) c = Some c'
  have 2: step' ⊔ c' ⊆ c' by(rule lfpc-pfp[OF 1])
  have 3: strip (γc (step' ⊔ c')) = c
    by(simp add: strip-lfpc[OF - 1])
  have lfp (step UNIV) c ≤ γc (step' ⊔ c')
proof(rule lfp-lowerbound[simplified,OF 3])
  show step UNIV (γc (step' ⊔ c')) ≤ γc (step' ⊔ c')
proof(rule step-preserves-le[OF - -])
  show UNIV ⊆ γo ⊔ by simp
  show γc (step' ⊔ c') ≤ γc c' by(rule mono-gamma-c[OF 2])
qed
qed
from this 2 show lfp (step UNIV) c ≤ γc c'

```

by (*blast intro: mono-gamma-c order-trans*)
qed

end

6.1 Monotonicity

locale *Abs-Int-mono* = *Abs-Int* +
assumes *mono-plus'*: $a1 \sqsubseteq b1 \implies a2 \sqsubseteq b2 \implies plus' a1 a2 \sqsubseteq plus' b1 b2$
begin

lemma *mono-aval'*: $S \sqsubseteq S' \implies aval' e S \sqsubseteq aval' e S'$
by(*induction e*) (*auto simp: le-st-def lookup-def mono-plus'*)

lemma *mono-update*: $a \sqsubseteq a' \implies S \sqsubseteq S' \implies update S x a \sqsubseteq update S' x a'$
by(*auto simp add: le-st-def lookup-def update-def*)

lemma *mono-step'*: $S \sqsubseteq S' \implies c \sqsubseteq c' \implies step' S c \sqsubseteq step' S' c'$
apply(*induction c c' arbitrary: S S' rule: le-acom.induct*)
apply (*auto simp: Let-def mono-update mono-aval' mono-post le-join-disj split: option.split*)

done

end

6.2 Ascending Chain Condition

abbreviation *strict* $r == r \cap -(r^{\wedge-1})$
abbreviation *acc* $r == wf((strict r)^{\wedge-1})$

lemma *strict-inv-image*: $strict(inv-image r f) = inv-image (strict r) f$
by(*auto simp: inv-image-def*)

lemma *acc-inv-image*:
 $acc r \implies acc (inv-image r f)$
by (*metis converse-inv-image strict-inv-image wf-inv-image*)

ACC for option type:

lemma *acc-option*: assumes $acc \{(x,y::'a::preord). x \sqsubseteq y\}$
shows $acc \{(x,y::'a::preord option). x \sqsubseteq y\}$
proof(*auto simp: wf-eq-minimal*)
fix $xo :: 'a option$ and Qo assume $xo : Qo$
let $?Q = \{x. Some x \in Qo\}$
show $\exists yo \in Qo. \forall zo. yo \sqsubseteq zo \wedge \sim zo \sqsubseteq yo \longrightarrow zo \notin Qo$ (*is* $\exists zo \in Qo. ?P zo$)
proof *cases*
assume $?Q = \{\}$
hence $?P None$ by *auto*
moreover have $None \in Qo$ using $\langle ?Q = \{\} \rangle \langle xo : Qo \rangle$
by *auto (metis not-Some-eq)*
ultimately show *?thesis* by *blast*

```

next
  assume ?Q ≠ {}
  with assms show ?thesis
    apply(auto simp: wf-eq-minimal)
    apply(erule-tac x=?Q in allE)
    apply auto
    apply(rule-tac x = Some z in beexI)
    by auto
qed
qed

```

ACC for abstract states, via measure functions.

```

lemma measure-st: assumes (strict{(x,y::'a::SL-top). x ⊆ y})-1 <= measure
  m
and ∀ x y::'a::SL-top. x ⊆ y ∧ y ⊆ x ⟶ m x = m y
shows (strict{(S,S'::'a::SL-top st). S ⊆ S'})-1 ⊆
  measure(%fd. ∑ x | x ∈ set(dom fd) ∧ ~ ⊤ ⊆ fun fd x. m(fun fd x)+1)
proof -
  { fix S S' :: 'a st assume S ⊆ S' ~ S' ⊆ S
    let ?X = set(dom S) let ?Y = set(dom S')
    let ?f = fun S let ?g = fun S'
    let ?X' = {x: ?X. ~ ⊤ ⊆ ?f x} let ?Y' = {y: ?Y. ~ ⊤ ⊆ ?g y}
    from ⟨S ⊆ S'⟩ have ∀ y ∈ ?Y' ∩ ?X. ?f y ⊆ ?g y
      by(auto simp: le-st-def lookup-def)
    hence 1: ∀ y ∈ ?Y' ∩ ?X. m(?g y)+1 ≤ m(?f y)+1
      using assms(1,2) by(fastforce)
    from ⟨~ S' ⊆ S⟩ obtain u where u: u : ?X ~ lookup S' u ⊆ ?f u
      by(auto simp: le-st-def)
    hence u : ?X' by simp (metis preord-class.le-trans top)
    have ?Y' - ?X = {} using ⟨S ⊆ S'⟩ by(fastforce simp: le-st-def lookup-def)
    have ?Y' ∩ ?X <= ?X' apply auto
      apply (metis ⟨S ⊆ S'⟩ le-st-def lookup-def preord-class.le-trans)
    done
    have (∑ y ∈ ?Y'. m(?g y)+1) = (∑ y ∈ (?Y' - ?X) ∪ (?Y' ∩ ?X). m(?g y)+1)
      by (metis Un-Diff-Int)
    also have ... = (∑ y ∈ ?Y' ∩ ?X. m(?g y)+1)
      using ⟨?Y' - ?X = {}⟩ by (metis Un-empty-left)
    also have ... < (∑ x ∈ ?X'. m(?f x)+1)
  }
proof cases
  assume u ∈ ?Y'
  hence m(?g u) < m(?f u) using assms(1) ⟨S ⊆ S'⟩ u
    by (fastforce simp: le-st-def lookup-def)
  have (∑ y ∈ ?Y' ∩ ?X. m(?g y)+1) < (∑ y ∈ ?Y' ∩ ?X. m(?f y)+1)
    using ⟨u: ?X⟩ ⟨u: ?Y'⟩ ⟨m(?g u) < m(?f u)⟩
    by(fastforce intro!: sum-strict-mono-ex1[OF - 1])
  also have ... ≤ (∑ y ∈ ?X'. m(?f y)+1)
    by(simp add: sum-mono2[OF - ⟨?Y' ∩ ?X <= ?X'⟩])
  finally show ?thesis .
next

```

```

assume  $u \notin ?Y'$ 
with  $\langle ?Y' \cap ?X \leq ?X' \rangle$  have  $?Y' \cap ?X - \{u\} \leq ?X' - \{u\}$  by blast
have  $(\sum_{y \in ?Y' \cap ?X} m(?g\ y) + 1) = (\sum_{y \in ?Y' \cap ?X - \{u\}} m(?g\ y) + 1)$ 
proof–
  have  $?Y' \cap ?X = ?Y' \cap ?X - \{u\}$  using  $\langle u \notin ?Y' \rangle$  by auto
  thus ?thesis by metis
qed
also have  $\dots < (\sum_{y \in ?Y' \cap ?X - \{u\}} m(?g\ y) + 1) + (\sum_{y \in \{u\}} m(?f\ y) + 1)$ 
by simp
  also have  $(\sum_{y \in ?Y' \cap ?X - \{u\}} m(?g\ y) + 1) \leq (\sum_{y \in ?Y' \cap ?X - \{u\}} m(?f\ y) + 1)$ 
  using 1 by(blast intro: sum-mono)
  also have  $\dots \leq (\sum_{y \in ?X' - \{u\}} m(?f\ y) + 1)$ 
  by(simp add: sum-mono2[OF -  $\langle ?Y' \cap ?X - \{u\} \leq ?X' - \{u\} \rangle$ ])
  also have  $\dots + (\sum_{y \in \{u\}} m(?f\ y) + 1) = (\sum_{y \in (?X' - \{u\}) \cup \{u\}} m(?f\ y) + 1)$ 
  using  $\langle u : ?X' \rangle$  by(subst sum.union-disjoint[symmetric]) auto
  also have  $\dots = (\sum_{x \in ?X'} m(?f\ x) + 1)$ 
  using  $\langle u : ?X' \rangle$  by(simp add: insert-absorb)
  finally show ?thesis by (blast intro: add-right-mono)
qed
finally have  $(\sum_{y \in ?Y'} m(?g\ y) + 1) < (\sum_{x \in ?X'} m(?f\ x) + 1)$  .
} thus ?thesis by(auto simp add: measure-def inv-image-def)
qed

```

ACC for acom. First the ordering on acom is related to an ordering on lists of annotations.

lemma *listrel-Cons-iff*:

```

 $(x \# xs, y \# ys) : listrel\ r \longleftrightarrow (x, y) \in r \wedge (xs, ys) \in listrel\ r$ 
by (blast intro: listrel.Cons)

```

lemma *listrel-app*: $(xs1, ys1) : listrel\ r \implies (xs2, ys2) : listrel\ r$

```

 $\implies (xs1 @ xs2, ys1 @ ys2) : listrel\ r$ 
by(auto simp add: listrel-iff-zip)

```

lemma *listrel-app-same-size*: $size\ xs1 = size\ ys1 \implies size\ xs2 = size\ ys2 \implies$

```

 $(xs1 @ xs2, ys1 @ ys2) : listrel\ r \longleftrightarrow$ 
 $(xs1, ys1) : listrel\ r \wedge (xs2, ys2) : listrel\ r$ 
by(auto simp add: listrel-iff-zip)

```

lemma *listrel-converse*: $listrel(r^{-1}) = (listrel\ r)^{-1}$

proof–

```

{ fix  $xs\ ys$ 
  have  $(xs, ys) : listrel(r^{-1}) \longleftrightarrow (ys, xs) : listrel\ r$ 
  apply(induct xs arbitrary: ys)
  apply (fastforce simp: listrel.Nil)
  apply (fastforce simp: listrel-Cons-iff)
  done
} thus ?thesis by auto

```

qed

lemma *acc-listrel*: fixes $r :: ('a * 'a) \text{set}$ **assumes** *refl* r **and** *trans* r
and *acc* r **shows** *acc* (*listrel* $r - \{(\[], \[])\}$)

proof –

have *refl*: $\forall x. (x, x) : r$ **using** $\langle \text{refl } r \rangle$ **unfolding** *refl-on-def* **by** *blast*
have *trans*: $\forall x y z. (x, y) : r \implies (y, z) : r \implies (x, z) : r$
using $\langle \text{trans } r \rangle$ **unfolding** *trans-def* **by** *blast*
from *assms*(β) **obtain** $mx :: 'a \text{ set} \implies 'a$ **where**
 $mx: \forall S x. x : S \implies mx S : S \wedge (\forall y. (mx S, y) : \text{strict } r \longrightarrow y \notin S)$
by(*simp add: wf-eq-minimal*) *metis*
let $?R = \text{listrel } r - \{(\[], \[])\}$
{ fix Q **and** $xs :: 'a \text{ list}$
have $xs \in Q \implies \exists ys. ys \in Q \wedge (\forall zs. (ys, zs) \in \text{strict } ?R \longrightarrow zs \notin Q)$
 $(\text{is } - \implies \exists ys. ?P Q ys)$
proof(*induction xs arbitrary: Q rule: length-induct*)
case ($1\ xs$)
{ have $\forall ys Q. \text{size } ys < \text{size } xs \implies ys : Q \implies \exists ms. ?P Q ms$
using $1.IH$ **by** *blast*
} **note** $IH = \text{this}$
show $?case$
proof(*cases xs*)
case Nil **with** $\langle xs : Q \rangle$ **have** $?P Q []$ **by** *auto*
thus $?thesis$ **by** *blast*
next
case ($Cons\ x\ ys$)
let $?Q1 = \{a. \exists bs. \text{size } bs = \text{size } ys \wedge a \# bs : Q\}$
have $x : ?Q1$ **using** $\langle xs : Q \rangle$ *Cons* **by** *auto*
from $mx[OF\ \text{this}]$ **obtain** $m1$ **where**
 $1: m1 \in ?Q1 \wedge (\forall y. (m1, y) \in \text{strict } r \longrightarrow y \notin ?Q1)$ **by** *blast*
then **obtain** $ms1$ **where** $\text{size } ms1 = \text{size } ys\ m1 \# ms1 : Q$ **by** *blast+*
hence $\text{size } ms1 < \text{size } xs$ **using** *Cons* **by** *auto*
let $?Q2 = \{bs. \exists m1'. (m1', m1) : r \wedge (m1, m1') : r \wedge m1' \# bs : Q \wedge \text{size } bs = \text{size } ms1\}$
have $ms1 : ?Q2$ **using** $\langle m1 \# ms1 : Q \rangle$ **by**(*blast intro: refl*)
from $IH[OF\ \langle \text{size } ms1 < \text{size } xs \rangle\ \text{this}]$
obtain ms **where** $2: ?P\ ?Q2\ ms$ **by** *auto*
then **obtain** $m1'$ **where** $m1': (m1', m1) : r \wedge (m1, m1') : r \wedge m1' \# ms : Q$
by *blast*
hence $\forall ab. (m1' \# ms, ab) : \text{strict } ?R \longrightarrow ab \notin Q$ **using** $1\ 2$
apply (*auto simp: listrel-Cons-iff*)
apply (*metis* $\langle \text{length } ms1 = \text{length } ys \rangle$ *listrel-eq-len trans*)
by (*metis* $\langle \text{length } ms1 = \text{length } ys \rangle$ *listrel-eq-len trans*)
with $m1'$ **show** $?thesis$ **by** *blast*
qed
qed
}
thus $?thesis$ **unfolding** *wf-eq-minimal* **by** (*metis converse-iff*)

qed

lemma *le-iff-le-annos*: $c1 \sqsubseteq c2 \iff$
(*annos* $c1$, *annos* $c2$) : $listrel\{(x,y). x \sqsubseteq y\} \wedge strip\ c1 = strip\ c2$
apply(*induct* $c1\ c2$ *rule*: *le-acom.induct*)
apply (*auto simp*: *listrel.Nil listrel-Cons-iff listrel-app size-annos-same2*)
apply (*metis listrel-app-same-size size-annos-same*)
done

lemma *le-acom-subset-same-annos*:
(*strict*{ $(c,c'::'a::preord\ acom). c \sqsubseteq c'$ } $^{\wedge-1} \sqsubseteq$
(*strict*(*inv-image* (*listrel*{ $(a,a'::'a). a \sqsubseteq a'$ } - $\{([],[])\}$) *annos*)) $^{\wedge-1}$
by(*auto simp*: *le-iff-le-annos*)

lemma *acc-acom*: $acc\ \{(a,a'::'a::preord). a \sqsubseteq a'\} \implies$
 $acc\ \{(c,c'::'a\ acom). c \sqsubseteq c'\}$
apply(*rule wf-subset*[*OF* - *le-acom-subset-same-annos*])
apply(*rule acc-inv-image*[*OF* *acc-listrel*])
apply(*auto simp*: *refl-on-def trans-def intro: le-trans*)
done

Termination of the fixed-point finders, assuming monotone functions:

lemma *pfpc-termination*:
fixes $x0 :: 'a::preord$
assumes *mono*: $\bigwedge x\ y. x \sqsubseteq y \implies f\ x \sqsubseteq f\ y$ **and** $acc\ \{(x::'a,y). x \sqsubseteq y\}$
and $x0 \sqsubseteq f\ x0$ **shows** $\exists x. pfpc\ f\ x0 = Some\ x$
proof(*simp add*: *pfpc-def*, *rule wf-while-option-Some*[**where** $P = \%x. x \sqsubseteq f\ x$])
 show *wf* $\{(x, s). (s \sqsubseteq f\ s \wedge \neg f\ s \sqsubseteq s) \wedge x = f\ s\}$
 by(*rule wf-subset*[*OF* *assms*(2)]) *auto*
next
 show $x0 \sqsubseteq f\ x0$ **by**(*rule assms*)
next
 fix x **assume** $x \sqsubseteq f\ x$ **thus** $f\ x \sqsubseteq f\ (f\ x)$ **by**(*rule mono*)
qed

lemma *lpfpc-termination*:
fixes $f :: (('a::SL-top)\ option\ acom \Rightarrow 'a\ option\ acom)$
assumes $acc\ \{(x::'a,y). x \sqsubseteq y\}$ **and** $\bigwedge x\ y. x \sqsubseteq y \implies f\ x \sqsubseteq f\ y$
and $\bigwedge c. strip\ (f\ c) = strip\ c$
shows $\exists c'. lpfpc\ f\ c = Some\ c'$
unfolding *lpfpc-def*
apply(*rule pfpc-termination*)
 apply(*erule assms*(2))
 apply(*rule acc-acom*[*OF* *acc-option*[*OF* *assms*(1)])])
apply(*simp add*: *bot-acom assms*(3))
done

context *Abs-Int-mono*

```

begin

lemma AI-Some-measure:
  assumes  $(\text{strict}\{(x,y:'a). x \sqsubseteq y\})^{\wedge-1} \leq \text{measure } m$ 
  and  $\forall x y::'a. x \sqsubseteq y \wedge y \sqsubseteq x \longrightarrow m\ x = m\ y$ 
  shows  $\exists c'. \text{AI } c = \text{Some } c'$ 
  unfolding AI-def
  apply(rule lpfp-termination)
  apply(rule wf-subset[OF wf-measure measure-st[OF assms]])
  apply(erule mono-step'[OF le-refl])
  apply(rule strip-step')
done

end

end

```

7 Backward Analysis of Expressions

```

theory Abs-Int2
imports Abs-Int1 HOL-IMP.Vars
begin

instantiation prod :: (preord,preord) preord
begin

definition le-prod  $p1\ p2 = (\text{fst } p1 \sqsubseteq \text{fst } p2 \wedge \text{snd } p1 \sqsubseteq \text{snd } p2)$ 

instance
proof (standard, goal-cases)
  case 1 show ?case by(simp add: le-prod-def)
next
  case 2 thus ?case unfolding le-prod-def by(metis le-trans)
qed

end

hide-const bot

class L-top-bot = SL-top +
fixes meet ::  $'a \Rightarrow 'a \Rightarrow 'a$  (infixl  $\sqcap$  65)
and bot ::  $'a$  ( $\perp$ )
assumes meet-le1 [simp]:  $x \sqcap y \sqsubseteq x$ 
and meet-le2 [simp]:  $x \sqcap y \sqsubseteq y$ 
and meet-greatest:  $x \sqsubseteq y \Longrightarrow x \sqsubseteq z \Longrightarrow x \sqsubseteq y \sqcap z$ 
assumes bot[simp]:  $\perp \sqsubseteq x$ 
begin

lemma mono-meet:  $x \sqsubseteq x' \Longrightarrow y \sqsubseteq y' \Longrightarrow x \sqcap y \sqsubseteq x' \sqcap y'$ 

```

by (metis meet-greatest meet-le1 meet-le2 le-trans)

end

locale Val-abs1-gamma =

Gamma where $\gamma = \gamma$ for $\gamma :: 'av::L-top-bot \Rightarrow val set +$
assumes inter-gamma-subset-gamma-meet:

$\gamma a1 \cap \gamma a2 \subseteq \gamma(a1 \sqcap a2)$

and gamma-Bot[simp]: $\gamma \perp = \{\}$

begin

lemma in-gamma-meet: $x : \gamma a1 \Longrightarrow x : \gamma a2 \Longrightarrow x : \gamma(a1 \sqcap a2)$

by (metis IntI inter-gamma-subset-gamma-meet subsetD)

lemma gamma-meet[simp]: $\gamma(a1 \sqcap a2) = \gamma a1 \cap \gamma a2$

by (metis equalityI inter-gamma-subset-gamma-meet le-inf-iff mono-gamma meet-le1 meet-le2)

end

locale Val-abs1 =

Val-abs1-gamma where $\gamma = \gamma$

for $\gamma :: 'av::L-top-bot \Rightarrow val set +$

fixes test-num' :: $val \Rightarrow 'av \Rightarrow bool$

and filter-plus' :: $'av \Rightarrow 'av \Rightarrow 'av \Rightarrow 'av * 'av$

and filter-less' :: $bool \Rightarrow 'av \Rightarrow 'av \Rightarrow 'av * 'av$

assumes test-num': $test-num' n a = (n : \gamma a)$

and filter-plus': $filter-plus' a a1 a2 = (b1, b2) \Longrightarrow$

$n1 : \gamma a1 \Longrightarrow n2 : \gamma a2 \Longrightarrow n1+n2 : \gamma a \Longrightarrow n1 : \gamma b1 \wedge n2 : \gamma b2$

and filter-less': $filter-less' (n1 < n2) a1 a2 = (b1, b2) \Longrightarrow$

$n1 : \gamma a1 \Longrightarrow n2 : \gamma a2 \Longrightarrow n1 : \gamma b1 \wedge n2 : \gamma b2$

locale Abs-Int1 =

Val-abs1 where $\gamma = \gamma$ for $\gamma :: 'av::L-top-bot \Rightarrow val set$

begin

lemma in-gamma-join-UpI: $s : \gamma_o S1 \vee s : \gamma_o S2 \Longrightarrow s : \gamma_o(S1 \sqcup S2)$

by (metis (no-types) join-ge1 join-ge2 mono-gamma-o rev-subsetD)

fun aval'' :: $aexp \Rightarrow 'av st option \Rightarrow 'av$ where

$aval'' e None = \perp$ |

$aval'' e (Some sa) = aval' e sa$

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

by(cases S)(simp add: aval'-sound)+

7.1 Backward analysis

```

fun afilter :: aexp  $\Rightarrow$  'av  $\Rightarrow$  'av st option  $\Rightarrow$  'av st option where
  afilter (N n) a S = (if test-num' n a then S else None) |
  afilter (V x) a S = (case S of None  $\Rightarrow$  None | Some S  $\Rightarrow$ 
    let a' = lookup S x  $\sqcap$  a in
    if a'  $\sqsubseteq$   $\perp$  then None else Some(update S x a')) |
  afilter (Plus e1 e2) a S =
    (let (a1,a2) = filter-plus' a (aval'' e1 S) (aval'' e2 S)
     in afilter e1 a1 (afilter e2 a2 S))

```

The test for \perp in the V -case is important: \perp 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- \perp values. Otherwise the (pointwise) join of two abstract states, one of which contains \perp values, may produce too large a result, thus making the analysis less precise.

```

fun bfilter :: bexp  $\Rightarrow$  bool  $\Rightarrow$  'av st option  $\Rightarrow$  'av st option where
  bfilter (Bc v) res S = (if v=res then S else None) |
  bfilter (Not b) res S = bfilter b ( $\neg$  res) S |
  bfilter (And b1 b2) res S =
    (if res then bfilter b1 True (bfilter b2 True S)
     else bfilter b1 False S  $\sqcup$  bfilter b2 False S) |
  bfilter (Less e1 e2) res S =
    (let (res1,res2) = filter-less' res (aval'' e1 S) (aval'' e2 S)
     in afilter e1 res1 (afilter e2 res2 S))

```

lemma afilter-sound: $s : \gamma_o S \Longrightarrow \text{aval } e \ s : \gamma \ a \Longrightarrow s : \gamma_o (\text{afilter } e \ a \ S)$

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** $s : \gamma_f S'$ **using** $\langle s : \gamma_o S \rangle$

by(auto simp: in-gamma-option-iff)

moreover **hence** $s \ x : \gamma (\text{lookup } S' \ x)$ **by**(simp add: γ -st-def)

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

ultimately **show** ?case **using** V(1)

by(simp add: lookup-update Let-def γ -st-def)

 (metis mono-gamma emptyE in-gamma-meet gamma-Bot subset-empty)

next

case (Plus e1 e2) **thus** ?case

using filter-plus'[OF - aval''-sound[OF Plus(3)]] aval''-sound[OF Plus(3)]]

by (auto split: prod.split)

qed

lemma bfilter-sound: $s : \gamma_o S \Longrightarrow \text{bv} = \text{bval } b \ s \Longrightarrow s : \gamma_o (\text{bfilter } b \ \text{bv } S)$

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
    apply hypsubst-thin
    apply (fastforce simp: in-gamma-join-UpI)
    done
next
  case (Less e1 e2) thus ?case
    apply hypsubst-thin
    apply (auto split: prod.split)
    apply (metis afilter-sound filter-less' aval''-sound Less(1))
    done
qed

```

```

fun step' :: 'av st option  $\Rightarrow$  'av st option acom  $\Rightarrow$  'av st option acom
  where
    step' S (SKIP {P}) = (SKIP {S}) |
    step' S (x ::= e {P}) =
      x ::= e {case S of None  $\Rightarrow$  None | Some S  $\Rightarrow$  Some(update S x (aval' e S))} |
    step' S (c1;; c2) = step' S c1;; step' (post c1) c2 |
    step' S (IF b THEN c1 ELSE c2 {P}) =
      (let c1' = step' (bfilter b True S) c1; c2' = step' (bfilter b False S) c2
       in IF b THEN c1' ELSE c2' {post c1  $\sqcup$  post c2}) |
    step' S ({Inv} WHILE b DO c {P}) =
      {S  $\sqcup$  post c}
      WHILE b DO step' (bfilter b True Inv) c
      {bfilter b False Inv}

```

definition AI :: com \Rightarrow 'av st option acom option **where**
 AI = lfp_c (step' \top)

lemma strip-step'[simp]: strip(step' S c) = strip c
by(induct c arbitrary: S) (simp-all add: Let-def)

7.2 Soundness

lemma in-gamma-update:

$\llbracket s : \gamma_f S; i : \gamma a \rrbracket \Longrightarrow s(x := i) : \gamma_f(\text{update } S x a)$
by(simp add: γ -st-def lookup-update)

lemma step-preserves-le:

$\llbracket S \subseteq \gamma_o S'; cs \leq \gamma_c ca \rrbracket \Longrightarrow \text{step } S cs \leq \gamma_c (\text{step}' S' ca)$

proof(induction cs arbitrary: ca S S')

case SKIP thus ?case **by**(auto simp:SKIP-le map-acom-SKIP)

next

case Assign thus ?case

by (fastforce simp: Assign-le map-acom-Assing intro: aval'-sound in-gamma-update)

split: option.splits del:subsetD
next
case *Seq* **thus** *?case* **apply** (*auto simp: Seq-le map-acom-Seq*)
by (*metis le-post post-map-acom*)
next
case (*If b cs1 cs2 P*)
then obtain *ca1 ca2 Pa* **where**
ca = IF b THEN ca1 ELSE ca2 {Pa}
P ⊆ γ_o Pa cs1 ≤ γ_c ca1 cs2 ≤ γ_c ca2
by (*fastforce simp: If-le map-acom-If*)
moreover have *post cs1 ⊆ γ_o(post ca1 ⊔ post ca2)*
by (*metis (no-types) ‹cs1 ≤ γ_c ca1› join-ge1 le-post mono-gamma-o order-trans post-map-acom*)
moreover have *post cs2 ⊆ γ_o(post ca1 ⊔ post ca2)*
by (*metis (no-types) ‹cs2 ≤ γ_c ca2› join-ge2 le-post mono-gamma-o order-trans post-map-acom*)
ultimately show *?case* **using** *‹S ⊆ γ_o S'›*
by (*simp add: If.IH subset-iff bfilter-sound*)
next
case (*While I b cs1 P*)
then obtain *ca1 Ia Pa* **where**
ca = {Ia} WHILE b DO ca1 {Pa}
I ⊆ γ_o Ia P ⊆ γ_o Pa cs1 ≤ γ_c ca1
by (*fastforce simp: map-acom-While While-le*)
moreover have *S ∪ post cs1 ⊆ γ_o (S' ⊔ post ca1)*
using *‹S ⊆ γ_o S'› le-post[OF ‹cs1 ≤ γ_c ca1›, simplified]*
by (*metis (no-types) join-ge1 join-ge2 le-sup-iff mono-gamma-o order-trans*)
ultimately show *?case* **by** (*simp add: While.IH subset-iff bfilter-sound*)
qed

lemma *AI-sound: AI c = Some c' ⇒ CS c ≤ γ_c c'*
proof(*simp add: CS-def AI-def*)
assume *1: lpfpc (step' ⊔) c = Some c'*
have *2: step' ⊔ c' ⊆ c'* **by**(*rule lpfpc-pfp[OF 1]*)
have *3: strip (γ_c (step' ⊔ c')) = c*
by(*simp add: strip-lpfpc[OF - 1]*)
have *lfp (step UNIV) c ≤ γ_c (step' ⊔ c')*
proof(*rule lfp-lowerbound[simplified,OF 3]*)
show *step UNIV (γ_c (step' ⊔ c')) ≤ γ_c (step' ⊔ c')*
proof(*rule step-preserves-le[OF - -]*)
show *UNIV ⊆ γ_o ⊔* **by** *simp*
show *γ_c (step' ⊔ c') ≤ γ_c c'* **by**(*rule mono-gamma-c[OF 2]*)
qed
qed
from *this 2* **show** *lfp (step UNIV) c ≤ γ_c c'*
by (*blast intro: mono-gamma-c order-trans*)
qed

7.3 Commands over a set of variables

Key invariant: the domains of all abstract states are subsets of the set of variables of the program.

definition $\text{domo } S = (\text{case } S \text{ of } \text{None} \Rightarrow \{\} \mid \text{Some } S' \Rightarrow \text{set}(\text{dom } S'))$

definition $\text{Com} :: \text{vname set} \Rightarrow 'a \text{ st option acom set}$ **where**
 $\text{Com } X = \{c. \forall S \in \text{set}(\text{annos } c). \text{domo } S \subseteq X\}$

lemma $\text{domo-Top}[\text{simp}]$: $\text{domo } \top = \{\}$
by($\text{simp add: domo-def Top-st-def Top-option-def}$)

lemma $\text{bot-acom-Com}[\text{simp}]$: $\perp_c c \in \text{Com } X$
by($\text{simp add: bot-acom-def Com-def domo-def}$)

lemma post-in-annos : $\text{post } c : \text{set}(\text{annos } c)$
by($\text{induction } c$) simp-all

lemma domo-join : $\text{domo } (S \sqcup T) \subseteq \text{domo } S \cup \text{domo } T$
by($\text{auto simp: domo-def join-st-def split: option.split}$)

lemma domo-afilter : $\text{vars } a \subseteq X \Longrightarrow \text{domo } S \subseteq X \Longrightarrow \text{domo}(\text{afilter } a \ i \ S) \subseteq X$
apply($\text{induction } a \text{ arbitrary: } i \ S$)
apply($\text{simp add: domo-def}$)
apply($\text{simp add: domo-def Let-def update-def lookup-def split: option.splits}$)
apply blast
apply($\text{simp split: prod.split}$)
done

lemma domo-bfilter : $\text{vars } b \subseteq X \Longrightarrow \text{domo } S \subseteq X \Longrightarrow \text{domo}(\text{bfilter } b \ \text{bv } S) \subseteq X$
apply($\text{induction } b \text{ arbitrary: } \text{bv } S$)
apply($\text{simp add: domo-def}$)
apply(simp)
apply(simp)
apply rule
apply ($\text{metis le-sup-iff subset-trans[OF domo-join]}$)
apply($\text{simp split: prod.split}$)
by ($\text{metis domo-afilter}$)

lemma step'-Com :
 $\text{domo } S \subseteq X \Longrightarrow \text{vars}(\text{strip } c) \subseteq X \Longrightarrow c : \text{Com } X \Longrightarrow \text{step}' S \ c : \text{Com } X$
apply($\text{induction } c \text{ arbitrary: } S$)
apply(simp add: Com-def)
apply($\text{simp add: Com-def domo-def update-def split: option.splits}$)
apply($\text{simp (no-asm-use) add: Com-def ball-Un}$)
apply ($\text{metis post-in-annos}$)
apply($\text{simp (no-asm-use) add: Com-def ball-Un}$)
apply rule
apply ($\text{metis Un-assoc domo-join order-trans post-in-annos subset-Un-eq}$)

```

apply (metis domo-bfilter)
apply(simp (no-asm-use) add: Com-def)
apply rule
apply (metis domo-join le-sup-iff post-in-annos subset-trans)
apply rule
apply (metis domo-bfilter)
by (metis domo-bfilter)

end

```

7.4 Monotonicity

```

locale Abs-Int1-mono = Abs-Int1 +
assumes mono-plus':  $a1 \sqsubseteq b1 \implies a2 \sqsubseteq b2 \implies plus' a1 a2 \sqsubseteq plus' b1 b2$ 
and mono-filter-plus':  $a1 \sqsubseteq b1 \implies a2 \sqsubseteq b2 \implies r \sqsubseteq r' \implies$ 
    $filter-plus' r a1 a2 \sqsubseteq filter-plus' r' b1 b2$ 
and mono-filter-less':  $a1 \sqsubseteq b1 \implies a2 \sqsubseteq b2 \implies$ 
    $filter-less' bv a1 a2 \sqsubseteq filter-less' bv b1 b2$ 
begin

```

```

lemma mono-aval':  $S \sqsubseteq S' \implies aval' e S \sqsubseteq aval' e S'$ 
by(induction e) (auto simp: le-st-def lookup-def mono-plus')

```

```

lemma mono-aval'':  $S \sqsubseteq S' \implies aval'' e S \sqsubseteq aval'' e S'$ 
apply(cases S)
  apply simp
apply(cases S')
  apply simp
by (simp add: mono-aval')

```

```

lemma mono-afilter:  $r \sqsubseteq r' \implies S \sqsubseteq S' \implies afilter e r S \sqsubseteq afilter e r' S'$ 
apply(induction e arbitrary: r r' S S')
apply(auto simp: test-num' Let-def split: option.splits prod.splits)
apply (metis mono-gamma subsetD)
apply(rename-tac list a b c d, drule-tac x = list in mono-lookup)
apply (metis mono-meet le-trans)
apply (metis mono-meet mono-lookup mono-update)
apply(metis mono-aval'' mono-filter-plus'[simplified le-prod-def] fst-conv snd-conv)
done

```

```

lemma mono-bfilter:  $S \sqsubseteq S' \implies bfilter b r S \sqsubseteq bfilter b r S'$ 
apply(induction b arbitrary: r S S')
apply(auto simp: le-trans[OF - join-ge1] le-trans[OF - join-ge2] split: prod.splits)
apply(metis mono-aval'' mono-afilter mono-filter-less'[simplified le-prod-def] fst-conv)
done

```

```

lemma mono-step':  $S \sqsubseteq S' \implies c \sqsubseteq c' \implies step' S c \sqsubseteq step' S' c'$ 
apply(induction c c' arbitrary: S S' rule: le-acom.induct)

```


apply (*auto simp: mono-post mono-bfilter mono-update mono-aval' Let-def le-join-disj*
split: option.split)

done

lemma *mono-step'2: mono (step' S)*
by(*simp add: mono-def mono-step'[OF le-refl]*)

end

end

8 Interval Analysis

theory *Abs-Int2-ivl*
imports *Abs-Int2 HOL-IMP.Abs-Int-Tests*
begin

datatype *ivl = I int option int option*

definition γ -*ivl* $i =$ (*case i of*
I (Some l) (Some h) \Rightarrow {l..h} |
I (Some l) None \Rightarrow {l..} |
I None (Some h) \Rightarrow {..h} |
I None None \Rightarrow UNIV)

abbreviation *I-Some-Some* $::$ *int \Rightarrow int \Rightarrow ivl* (*{-...}*) **where**
{lo...hi} == I (Some lo) (Some hi)

abbreviation *I-Some-None* $::$ *int \Rightarrow ivl* (*{-...}*) **where**
{lo...} == I (Some lo) None

abbreviation *I-None-Some* $::$ *int \Rightarrow ivl* (*{...}*) **where**
{..hi} == I None (Some hi)

abbreviation *I-None-None* $::$ *ivl* (*{...}*) **where**
{...} == I None None

definition *num-ivl* $n =$ *{n...n}*

fun *in-ivl* $::$ *int \Rightarrow ivl \Rightarrow bool* **where**
in-ivl k (I (Some l) (Some h)) \longleftrightarrow $l \leq k \wedge k \leq h$ |
in-ivl k (I (Some l) None) \longleftrightarrow $l \leq k$ |
in-ivl k (I None (Some h)) \longleftrightarrow $k \leq h$ |
in-ivl k (I None None) \longleftrightarrow True

instantiation *option* $::$ (*plus*)*plus*
begin

fun *plus-option* **where**
Some x + Some y = Some(x+y) |
- + - = None

instance ..

end

definition *empty* **where** *empty* = $\{1 \dots 0\}$

fun *is-empty* **where**
is-empty $\{l \dots h\}$ = $(h < l)$ |
is-empty - = *False*

lemma [*simp*]: *is-empty* $(I\ l\ h)$ =
 $(\text{case } l \text{ of } \text{Some } l \Rightarrow (\text{case } h \text{ of } \text{Some } h \Rightarrow h < l \mid \text{None} \Rightarrow \text{False}) \mid \text{None} \Rightarrow \text{False})$
by(*auto split:option.split*)

lemma [*simp*]: *is-empty* $i \Rightarrow \gamma\text{-ivl } i = \{\}$
by(*auto simp add: \(\gamma\text{-ivl-def split: ivl.split option.split}\)*)

definition *plus-ivl* $i1\ i2$ = $(\text{if } \text{is-empty } i1 \mid \text{is-empty } i2 \text{ then } \text{empty} \text{ else}$
 $\text{case } (i1, i2) \text{ of } (I\ l1\ h1, I\ l2\ h2) \Rightarrow I\ (l1+l2)\ (h1+h2))$

instantiation *ivl* :: *SL-top*
begin

definition *le-option* :: $\text{bool} \Rightarrow \text{int option} \Rightarrow \text{int option} \Rightarrow \text{bool}$ **where**
le-option $\text{pos } x\ y$ =
 $(\text{case } x \text{ of } (\text{Some } i) \Rightarrow (\text{case } y \text{ of } \text{Some } j \Rightarrow i \leq j \mid \text{None} \Rightarrow \text{pos})$
 $\mid \text{None} \Rightarrow (\text{case } y \text{ of } \text{Some } j \Rightarrow \neg \text{pos} \mid \text{None} \Rightarrow \text{True}))$

fun *le-aux* **where**
le-aux $(I\ l1\ h1)\ (I\ l2\ h2)$ = $(\text{le-option } \text{False } l2\ l1 \ \& \ \text{le-option } \text{True } h1\ h2)$

definition *le-ivl* **where**
i1 \sqsubseteq *i2* =
 $(\text{if } \text{is-empty } i1 \text{ then } \text{True} \text{ else}$
 $\text{if } \text{is-empty } i2 \text{ then } \text{False} \text{ else } \text{le-aux } i1\ i2)$

definition *min-option* :: $\text{bool} \Rightarrow \text{int option} \Rightarrow \text{int option} \Rightarrow \text{int option}$ **where**
min-option $\text{pos } o1\ o2$ = $(\text{if } \text{le-option } \text{pos } o1\ o2 \text{ then } o1 \text{ else } o2)$

definition *max-option* :: $\text{bool} \Rightarrow \text{int option} \Rightarrow \text{int option} \Rightarrow \text{int option}$ **where**
max-option $\text{pos } o1\ o2$ = $(\text{if } \text{le-option } \text{pos } o1\ o2 \text{ then } o2 \text{ else } o1)$

definition $i1 \sqcup i2$ =
 $(\text{if } \text{is-empty } i1 \text{ then } i2 \text{ else if } \text{is-empty } i2 \text{ then } i1$
 $\text{else case } (i1, i2) \text{ of } (I\ l1\ h1, I\ l2\ h2) \Rightarrow$
 $I\ (\text{min-option } \text{False } l1\ l2)\ (\text{max-option } \text{True } h1\ h2))$

definition \top = $\{\dots\}$

```

instance
proof (standard, goal-cases)
  case (1 x) thus ?case
    by(cases x, simp add: le-ivl-def le-option-def split: option.split)
next
  case (2 x y z) thus ?case
    by(cases x, cases y, cases z, auto simp: le-ivl-def le-option-def split: option.splits if-splits)
next
  case (3 x y) thus ?case
    by(cases x, cases y, simp add: le-ivl-def join-ivl-def le-option-def min-option-def max-option-def split: option.splits)
next
  case (4 x y) thus ?case
    by(cases x, cases y, simp add: le-ivl-def join-ivl-def le-option-def min-option-def max-option-def split: option.splits)
next
  case (5 x y z) thus ?case
    by(cases x, cases y, cases z, auto simp add: le-ivl-def join-ivl-def le-option-def min-option-def max-option-def split: option.splits if-splits)
next
  case (6 x) thus ?case
    by(cases x, simp add: Top-ivl-def le-ivl-def le-option-def split: option.split)
qed

end

```

```

instantiation ivl :: L-top-bot
begin

```

```

definition i1  $\sqcap$  i2 = (if is-empty i1  $\vee$  is-empty i2 then empty else
  case (i1, i2) of (I l1 h1, I l2 h2)  $\Rightarrow$ 
    I (max-option False l1 l2) (min-option True h1 h2))

```

```

definition  $\perp$  = empty

```

```

instance
proof (standard, goal-cases)
  case 1 thus ?case
    by (simp add: meet-ivl-def empty-def le-ivl-def le-option-def max-option-def min-option-def split: ivl.splits option.splits)
next
  case 2 thus ?case
    by (simp add: empty-def meet-ivl-def le-ivl-def le-option-def max-option-def min-option-def split: ivl.splits option.splits)
next
  case (3 x y z) thus ?case
    by (cases x, cases y, cases z, auto simp add: le-ivl-def meet-ivl-def empty-def)

```

```

le-option-def max-option-def min-option-def split: option.splits if-splits)
next
  case (4 x) show ?case by(cases x, simp add: bot-ivl-def empty-def le-ivl-def)
qed

end

instantiation option :: (minus)minus
begin

fun minus-option where
  Some x - Some y = Some(x-y) |
  - - = None

instance ..

end

definition minus-ivl i1 i2 = (if is-empty i1 | is-empty i2 then empty else
  case (i1,i2) of (I l1 h1, I l2 h2) ⇒ I (l1-h2) (h1-l2))

lemma gamma-minus-ivl:
  n1 : γ-ivl i1 ⇒ n2 : γ-ivl i2 ⇒ n1-n2 : γ-ivl(minus-ivl i1 i2)
by(auto simp add: minus-ivl-def γ-ivl-def split: ivl.splits option.splits)

definition filter-plus-ivl i i1 i2 = (if is-empty i1 | is-empty i2 then empty else
  i1 ⊔ minus-ivl i i2, i2 ⊔ minus-ivl i i1)

fun filter-less-ivl :: bool ⇒ ivl ⇒ ivl ⇒ ivl * ivl where
  filter-less-ivl res (I l1 h1) (I l2 h2) =
    (if is-empty(I l1 h1) ∨ is-empty(I l2 h2) then (empty, empty) else
     if res
     then (I l1 (min-option True h1 (h2 - Some 1)),
           I (max-option False (l1 + Some 1) l2) h2)
     else (I (max-option False l1 l2) h1, I l2 (min-option True h1 h2)))

global-interpretation Val-abs
where γ = γ-ivl and num' = num-ivl and plus' = plus-ivl
proof (standard, goal-cases)
  case 1 thus ?case
    by(auto simp: γ-ivl-def le-ivl-def le-option-def split: ivl.split option.split if-splits)
next
  case 2 show ?case by(simp add: γ-ivl-def Top-ivl-def)
next
  case 3 thus ?case by(simp add: γ-ivl-def num-ivl-def)
next
  case 4 thus ?case
    by(auto simp add: γ-ivl-def plus-ivl-def split: ivl.split option.splits)
qed

```

```

global-interpretation Val-abs1-gamma
where  $\gamma = \gamma\text{-ivl}$  and  $num' = num\text{-ivl}$  and  $plus' = plus\text{-ivl}$ 
defines  $aval\text{-ivl} = aval'$ 
proof (standard, goal-cases)
  case 1 thus ?case
    by(auto simp add:  $\gamma\text{-ivl-def}$  meet-ivl-def empty-def min-option-def max-option-def
      split: ivl.split option.split)
  next
    case 2 show ?case by(auto simp add: bot-ivl-def  $\gamma\text{-ivl-def}$  empty-def)
qed

```

```

lemma mono-minus-ivl:
   $i1 \sqsubseteq i1' \implies i2 \sqsubseteq i2' \implies minus\text{-ivl } i1 \ i2 \sqsubseteq minus\text{-ivl } i1' \ i2'$ 
apply(auto simp add: minus-ivl-def empty-def le-ivl-def le-option-def split: ivl.splits)
  apply(simp split: option.splits)
  apply(simp split: option.splits)
  apply(simp split: option.splits)
done

```

```

global-interpretation Val-abs1
where  $\gamma = \gamma\text{-ivl}$  and  $num' = num\text{-ivl}$  and  $plus' = plus\text{-ivl}$ 
and  $test\text{-num}' = in\text{-ivl}$ 
and  $filter\text{-plus}' = filter\text{-plus-ivl}$  and  $filter\text{-less}' = filter\text{-less-ivl}$ 
proof (standard, goal-cases)
  case 1 thus ?case
    by (simp add:  $\gamma\text{-ivl-def}$  split: ivl.split option.split)
  next
    case 2 thus ?case
      by(auto simp add: filter-plus-ivl-def)
      (metis gamma-minus-ivl add-diff-cancel add.commute)+
  next
    case (3 - - a1 a2) thus ?case
      by(cases a1, cases a2,
        auto simp:  $\gamma\text{-ivl-def}$  min-option-def max-option-def le-option-def split: if-splits
        option.splits)
qed

```

```

global-interpretation Abs-Int1
where  $\gamma = \gamma\text{-ivl}$  and  $num' = num\text{-ivl}$  and  $plus' = plus\text{-ivl}$ 
and  $test\text{-num}' = in\text{-ivl}$ 
and  $filter\text{-plus}' = filter\text{-plus-ivl}$  and  $filter\text{-less}' = filter\text{-less-ivl}$ 
defines  $afilter\text{-ivl} = afilter$ 
and  $bfilter\text{-ivl} = bfilter$ 
and  $step\text{-ivl} = step'$ 
and  $AI\text{-ivl} = AI$ 
and  $aval\text{-ivl}' = aval''$ 
..

```

Monotonicity:

```
global-interpretation Abs-Int1-mono
where  $\gamma = \gamma\text{-ivl}$  and  $\text{num}' = \text{num-ivl}$  and  $\text{plus}' = \text{plus-ivl}$ 
and  $\text{test-num}' = \text{in-ivl}$ 
and  $\text{filter-plus}' = \text{filter-plus-ivl}$  and  $\text{filter-less}' = \text{filter-less-ivl}$ 
proof (standard, goal-cases)
  case 1 thus ?case
    by(auto simp: plus-ivl-def le-ivl-def le-option-def empty-def split: if-splits ivl.splits option.splits)
  next
    case 2 thus ?case
      by(auto simp: filter-plus-ivl-def le-prod-def mono-meet mono-minus-ivl)
  next
    case (3 a1 b1 a2 b2) thus ?case
      apply(cases a1, cases b1, cases a2, cases b2, auto simp: le-prod-def)
      by(auto simp add: empty-def le-ivl-def le-option-def min-option-def max-option-def split: option.splits)
qed
```

8.1 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)
```

```
value show-acom-opt (AI-ivl test2-ivl)
value show-acom (((step-ivl  $\top$ )  $\sim$ 0) ( $\perp_c$  test2-ivl))
value show-acom (((step-ivl  $\top$ )  $\sim$ 1) ( $\perp_c$  test2-ivl))
value show-acom (((step-ivl  $\top$ )  $\sim$ 2) ( $\perp_c$  test2-ivl))
```

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 (((step-ivl  $\top$ )  $\sim$ 0) ( $\perp_c$  test3-ivl))
value show-acom (((step-ivl  $\top$ )  $\sim$ 1) ( $\perp_c$  test3-ivl))
value show-acom (((step-ivl  $\top$ )  $\sim$ 2) ( $\perp_c$  test3-ivl))
value show-acom (((step-ivl  $\top$ )  $\sim$ 3) ( $\perp_c$  test3-ivl))
value show-acom (((step-ivl  $\top$ )  $\sim$ 4) ( $\perp_c$  test3-ivl))
```

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 (((step-ivl  $\top$ )  $\sim$ 50) ( $\perp_c$  test4-ivl))
```

Relationships between variables are NOT captured:

```
value show-acom-opt (AI-ivl test5-ivl)
```


$FunDom (\lambda x. fun F1 x \nabla fun F2 x) (inter-list (dom F1) (dom F2))$

definition *narrow-st* $F1 F2 =$

$FunDom (\lambda x. fun F1 x \triangle fun F2 x) (inter-list (dom F1) (dom F2))$

instance

proof (*standard, goal-cases*)

case 1 thus *?case*

by(*simp add: widen-st-def le-st-def lookup-def widen1*)

next

case 2 thus *?case*

by(*simp add: widen-st-def le-st-def lookup-def widen2*)

next

case 3 thus *?case*

by(*auto simp: narrow-st-def le-st-def lookup-def narrow1*)

next

case 4 thus *?case*

by(*auto simp: narrow-st-def le-st-def lookup-def narrow2*)

qed

end

9.3 Option

instantiation *option* :: (WN) WN

begin

fun *widen-option* **where**

$None \nabla x = x \mid$

$x \nabla None = x \mid$

$(Some x) \nabla (Some y) = Some(x \nabla y)$

fun *narrow-option* **where**

$None \triangle x = None \mid$

$x \triangle None = None \mid$

$(Some x) \triangle (Some y) = Some(x \triangle y)$

instance

proof (*standard, goal-cases*)

case (1 $x y$) **show** *?case*

by(*induct x y rule: widen-option.induct*) (*simp-all add: widen1*)

next

case (2 $x y$) **show** *?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

9.4 Annotated commands

fun *map2-acom* :: ('a ⇒ 'a ⇒ 'a) ⇒ 'a acom ⇒ 'a acom ⇒ 'a acom **where**
map2-acom f (*SKIP* {a1}) (*SKIP* {a2}) = (*SKIP* {f a1 a2}) |
map2-acom f (x ::= e {a1}) (x' ::= e' {a2}) = (x ::= e {f a1 a2}) |
map2-acom f (c1;;c2) (c1';;c2') = (*map2-acom* f c1 c1';; *map2-acom* f c2 c2') |
map2-acom f (*IF* b *THEN* c1 *ELSE* c2 {a1}) (*IF* b' *THEN* c1' *ELSE* c2' {a2})
 =
 (*IF* b *THEN* *map2-acom* f c1 c1' *ELSE* *map2-acom* f c2 c2' {f a1 a2}) |
map2-acom f ({a1} *WHILE* b *DO* c {a2}) ({a3} *WHILE* b' *DO* c' {a4}) =
 ({f a1 a3} *WHILE* b *DO* *map2-acom* f c c' {f a2 a4})

abbreviation *widen-acom* :: ('a::WN)acom ⇒ 'a acom ⇒ 'a acom (**infix** ∇_c 65)
where *widen-acom* == *map2-acom* (∇)

abbreviation *narrow-acom* :: ('a::WN)acom ⇒ 'a acom ⇒ 'a acom (**infix** Δ_c 65)
where *narrow-acom* == *map2-acom* (Δ)

lemma *widen1-acom*: *strip* c = *strip* c' ⇒ c ⊆ c ∇_c c'
by(*induct c c' rule: le-acom.induct*)(*simp-all add: widen1*)

lemma *widen2-acom*: *strip* c = *strip* c' ⇒ c' ⊆ c ∇_c c'
by(*induct c c' rule: le-acom.induct*)(*simp-all add: widen2*)

lemma *narrow1-acom*: y ⊆ x ⇒ y ⊆ x Δ_c y
by(*induct y x rule: le-acom.induct*) (*simp-all add: narrow1*)

lemma *narrow2-acom*: y ⊆ x ⇒ x Δ_c y ⊆ x
by(*induct y x rule: le-acom.induct*) (*simp-all add: narrow2*)

9.5 Post-fixed point computation

definition *iter-widen* :: ('a acom ⇒ 'a acom) ⇒ 'a acom ⇒ ('a::WN)acom option
where *iter-widen* f = *while-option* (λc. ¬ f c ⊆ c) (λc. c ∇_c f c)

definition *iter-narrow* :: ('a acom ⇒ 'a acom) ⇒ 'a acom ⇒ 'a::WN acom option
where *iter-narrow* f = *while-option* (λc. ¬ c ⊆ c Δ_c f c) (λc. c Δ_c f c)

definition *pfp-wn* ::
 (('a::WN)option acom ⇒ 'a option acom) ⇒ com ⇒ 'a option acom option
where *pfp-wn* f c = (*case* *iter-widen* f (⊥_c c) of *None* ⇒ *None*
 | *Some* c' ⇒ *iter-narrow* f c')

lemma *strip-map2-acom*:

strip c1 = strip c2 \implies *strip(map2-acom f c1 c2) = strip c1*
by(*induct f c1 c2 rule: map2-acom.induct simp-all*)

lemma *iter-widen-pfp*: *iter-widen f c = Some c' \implies f c' \sqsubseteq c'*
by(*auto simp add: iter-widen-def dest: while-option-stop*)

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::WN 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 ∇_c f c) = strip c* **by** (*metis assms(1) strip-map2-acom*)
from *strip-while[OF this] assms(2)* **show** *?thesis* **by**(*simp add: iter-widen-def*)
qed

lemma *iter-narrow-pfp*: **assumes** *mono f* **and** *f c0 \sqsubseteq c0*
and *iter-narrow f c0 = Some c*
shows *f c \sqsubseteq c \wedge c \sqsubseteq c0* (**is** *?P c*)
proof –
{ **fix** *c* **assume** *?P c*
note *1 = conjunct1[OF this]* **and** *2 = conjunct2[OF this]*
let *?c' = c Δ_c f c*
have *?P ?c'*
proof
have *f ?c' \sqsubseteq f c* **by**(*rule monoD[OF <mono f> narrow2-acom[OF 1]]*)
also have $\dots \sqsubseteq ?c'$ **by**(*rule narrow1-acom[OF 1]*)
finally show *f ?c' \sqsubseteq ?c'*.
have *?c' \sqsubseteq c* **by** (*rule narrow2-acom[OF 1]*)
also have *c \sqsubseteq c0* **by**(*rule 2*)
finally show *?c' \sqsubseteq c0*.
qed
}
with *while-option-rule[where P = ?P, OF - assms(3)][simplified iter-narrow-def]*
assms(2) le-refl
show *?thesis* **by** *blast*
qed

lemma *pfp-wn-pfp*:
 $\llbracket mono f; pfp-wn f c = Some c' \rrbracket \implies f c' \sqsubseteq c'$
unfolding *pfp-wn-def*
by (*auto dest: iter-widen-pfp iter-narrow-pfp split: option.splits*)

lemma *strip-pfp-wn*:
 $\llbracket \forall c. strip(f c) = strip c; pfp-wn f c = Some c' \rrbracket \implies strip c' = c$

apply(*auto simp add: pfp-wn-def iter-narrow-def split: option.splits*)
by (*metis (mono-tags) strip-map2-acom strip-while strip-bot-acom strip-iter-widen*)

locale *Abs-Int2* = *Abs-Int1-mono*
where $\gamma = \gamma$ **for** $\gamma :: 'av :: \{WN, L\text{-top-bot}\} \Rightarrow \text{val set}$
begin

definition *AI-wn* :: *com* \Rightarrow *'av st option acom option* **where**
AI-wn = *pfp-wn (step' \top)*

lemma *AI-wn-sound*: *AI-wn* *c* = *Some c'* \Longrightarrow *CS* *c* \leq γ_c *c'*

proof(*simp add: CS-def AI-wn-def*)

assume *1*: *pfp-wn (step' \top) c* = *Some c'*

from *pfp-wn-pfp[OF mono-step'2 1]*

have *2*: *step' \top c' \sqsubseteq c'*.

have *3*: *strip (γ_c (step' \top c')) = c* **by**(*simp add: strip-pfp-wn[OF - 1]*)

have *lfp (step UNIV) c \leq γ_c (step' \top c')*

proof(*rule lfp-lowerbound[simplified, OF 3]*)

show *step UNIV (γ_c (step' \top c')) \leq γ_c (step' \top c')*

proof(*rule step-preserves-le[OF - -]*)

show *UNIV \subseteq $\gamma_o \top$* **by** *simp*

show γ_c (step' \top c') \leq γ_c c' **by**(*rule mono-gamma-c[OF 2]*)

qed

qed

from *this 2* **show** *lfp (step UNIV) c \leq γ_c c'*

by (*blast intro: mono-gamma-c order-trans*)

qed

end

global-interpretation *Abs-Int2*

where $\gamma = \gamma\text{-ivl}$ **and** *num'* = *num-ivl* **and** *plus'* = *plus-ivl*

and *test-num'* = *in-ivl*

and *filter-plus'* = *filter-plus-ivl* **and** *filter-less'* = *filter-less-ivl*

defines *AI-ivl'* = *AI-wn*

..

9.6 Tests

definition *step-up-ivl* *n* = $((\lambda c. c \nabla_c \text{step-ivl } \top c) \widehat{\widehat{n}})$

definition *step-down-ivl* *n* = $((\lambda c. c \Delta_c \text{step-ivl } \top c) \widehat{\widehat{n}})$

For *test3-ivl*, *AI-ivl* needed as many iterations as the loop took to execute. In contrast, *AI-ivl'* converges in a constant number of steps:

value *show-acom (step-up-ivl 1 (\perp_c test3-ivl))*

value *show-acom (step-up-ivl 2 (\perp_c test3-ivl))*

value *show-acom (step-up-ivl 3 (\perp_c test3-ivl))*

value *show-acom (step-up-ivl 4 (\perp_c test3-ivl))*

value *show-acom (step-up-ivl 5 (\perp_c test3-ivl))*

value *show-acom* (*step-down-ivl* 1 (*step-up-ivl* 5 (\perp_c *test3-ivl*)))
value *show-acom* (*step-down-ivl* 2 (*step-up-ivl* 5 (\perp_c *test3-ivl*)))
value *show-acom* (*step-down-ivl* 3 (*step-up-ivl* 5 (\perp_c *test3-ivl*)))

Now all the analyses terminate:

value *show-acom-opt* (*AI-ivl'* *test4-ivl*)
value *show-acom-opt* (*AI-ivl'* *test5-ivl*)
value *show-acom-opt* (*AI-ivl'* *test6-ivl*)

9.7 Termination: Intervals

definition *m-ivl* :: *ivl* \Rightarrow *nat* **where**

m-ivl *ivl* = (*case* *ivl* of *l* | *h* \Rightarrow
(*case* *l* of *None* \Rightarrow 0 | *Some* - \Rightarrow 1) + (*case* *h* of *None* \Rightarrow 0 | *Some* - \Rightarrow 1))

lemma *m-ivl-height*: *m-ivl* *ivl* \leq 2

by(*simp* *add*: *m-ivl-def* *split*: *ivl.split* *option.split*)

lemma *m-ivl-anti-mono*: (*y*::*ivl*) \sqsubseteq *x* \Longrightarrow *m-ivl* *x* \leq *m-ivl* *y*

by(*auto* *simp*: *m-ivl-def* *le-option-def* *le-ivl-def*
split: *ivl.split* *option.split* *if-splits*)

lemma *m-ivl-widen*:

\sim *y* \sqsubseteq *x* \Longrightarrow *m-ivl*(*x* ∇ *y*) $<$ *m-ivl* *x*

by(*auto* *simp*: *m-ivl-def* *widen-ivl-def* *le-option-def* *le-ivl-def*
split: *ivl.splits* *option.splits* *if-splits*)

lemma *Top-less-ivl*: \top \sqsubseteq *x* \Longrightarrow *m-ivl* *x* = 0

by(*auto* *simp*: *m-ivl-def* *le-option-def* *le-ivl-def* *empty-def* *Top-ivl-def*
split: *ivl.split* *option.split* *if-splits*)

definition *n-ivl* :: *ivl* \Rightarrow *nat* **where**

n-ivl *ivl* = 2 - *m-ivl* *ivl*

lemma *n-ivl-mono*: (*x*::*ivl*) \sqsubseteq *y* \Longrightarrow *n-ivl* *x* \leq *n-ivl* *y*

unfolding *n-ivl-def* **by** (*metis* *diff-le-mono2* *m-ivl-anti-mono*)

lemma *n-ivl-narrow*:

\sim *x* \sqsubseteq *x* Δ *y* \Longrightarrow *n-ivl*(*x* Δ *y*) $<$ *n-ivl* *x*

by(*auto* *simp*: *n-ivl-def* *m-ivl-def* *narrow-ivl-def* *le-option-def* *le-ivl-def*
split: *ivl.splits* *option.splits* *if-splits*)

9.8 Termination: Abstract State

definition *m-st* *m* *st* = (\sum *x* \in *set*(*dom* *st*). *m*(*fun* *st* *x*))

lemma *m-st-height*: **assumes** *finite* *X* **and** *set* (*dom* *S*) \subseteq *X*

shows *m-st* *m-ivl* *S* \leq 2 * *card* *X*

proof(*auto* *simp*: *m-st-def*)

have $(\sum_{x \in \text{set}(\text{dom } S)}. m\text{-ivl } (\text{fun } S \ x)) \leq (\sum_{x \in \text{set}(\text{dom } S)}. 2)$ (**is** $?L \leq -$)
by(*rule sum-mono*)(*simp add:m-ivl-height*)
also have $\dots \leq (\sum_{x \in X}. 2)$
by(*rule sum-mono2[OF assms]*) *simp*
also have $\dots = 2 * \text{card } X$ **by**(*simp*)
finally show $?L \leq \dots$.
qed

lemma *m-st-anti-mono*:

$S1 \sqsubseteq S2 \implies m\text{-st } m\text{-ivl } S2 \leq m\text{-st } m\text{-ivl } S1$
proof(*auto simp: m-st-def le-st-def lookup-def split: if-splits*)
let $?X = \text{set}(\text{dom } S1)$ **let** $?Y = \text{set}(\text{dom } S2)$
let $?f = \text{fun } S1$ **let** $?g = \text{fun } S2$
assume *asm*: $\forall x \in ?Y. (x \in ?X \longrightarrow ?f \ x \sqsubseteq ?g \ x) \wedge (x \in ?X \vee \top \sqsubseteq ?g \ x)$
hence $1: \forall y \in ?Y \cap ?X. m\text{-ivl}(?g \ y) \leq m\text{-ivl}(?f \ y)$ **by**(*simp add: m-ivl-anti-mono*)
have $0: \forall x \in ?Y - ?X. m\text{-ivl}(?g \ x) = 0$ **using** *asm* **by** (*auto simp: Top-less-ivl*)
have $(\sum_{y \in ?Y}. m\text{-ivl}(?g \ y)) = (\sum_{y \in (?Y - ?X) \cup (?Y \cap ?X)}. m\text{-ivl}(?g \ y))$
by (*metis Un-Diff-Int*)
also have $\dots = (\sum_{y \in ?Y - ?X}. m\text{-ivl}(?g \ y)) + (\sum_{y \in ?Y \cap ?X}. m\text{-ivl}(?g \ y))$
by(*subst sum.union-disjoint*) *auto*
also have $(\sum_{y \in ?Y - ?X}. m\text{-ivl}(?g \ y)) = 0$ **using** 0 **by** *simp*
also have $0 + (\sum_{y \in ?Y \cap ?X}. m\text{-ivl}(?g \ y)) = (\sum_{y \in ?Y \cap ?X}. m\text{-ivl}(?g \ y))$ **by**
simp
also have $\dots \leq (\sum_{y \in ?Y \cap ?X}. m\text{-ivl}(?f \ y))$
by(*rule sum-mono*)(*simp add: 1*)
also have $\dots \leq (\sum_{y \in ?X}. m\text{-ivl}(?f \ y))$
by(*simp add: sum-mono2[of ?X ?Y Int ?X, OF - Int-lower2]*)
finally show $(\sum_{y \in ?Y}. m\text{-ivl}(?g \ y)) \leq (\sum_{x \in ?X}. m\text{-ivl}(?f \ x))$.
qed

lemma *m-st-widen*:

assumes $\neg S2 \sqsubseteq S1$ **shows** $m\text{-st } m\text{-ivl } (S1 \nabla S2) < m\text{-st } m\text{-ivl } S1$
proof–
{ let $?X = \text{set}(\text{dom } S1)$ **let** $?Y = \text{set}(\text{dom } S2)$
let $?f = \text{fun } S1$ **let** $?g = \text{fun } S2$
fix x **assume** $x \in ?X \neg \text{lookup } S2 \ x \sqsubseteq ?f \ x$
have $(\sum_{x \in ?X \cap ?Y}. m\text{-ivl}(?f \ x \nabla ?g \ x)) < (\sum_{x \in ?X}. m\text{-ivl}(?f \ x))$ (**is** $?L <$
 $?R$)
proof *cases*
assume $x : ?Y$
have $?L < (\sum_{x \in ?X \cap ?Y}. m\text{-ivl}(?f \ x))$
proof(*rule sum-strict-mono-ex1, simp*)
show $\forall x \in ?X \cap ?Y. m\text{-ivl}(?f \ x \nabla ?g \ x) \leq m\text{-ivl}(?f \ x)$
by (*metis m-ivl-anti-mono widen1*)
next
show $\exists x \in ?X \cap ?Y. m\text{-ivl}(?f \ x \nabla ?g \ x) < m\text{-ivl}(?f \ x)$
using $\langle x : ?X \rangle \langle x : ?Y \rangle \langle \neg \text{lookup } S2 \ x \sqsubseteq ?f \ x \rangle$
by (*metis IntI m-ivl-widen lookup-def*)
qed

```

also have ...  $\leq$  ?R by(simp add: sum-mono2[OF - Int-lower1])
finally show ?thesis .
next
assume  $x \sim: ?Y$ 
have ?L  $\leq$  ( $\sum x \in ?X \cap ?Y. m\text{-ivl}(?f x)$ )
proof(rule sum-mono, simp)
  fix  $x$  assume  $x: ?X \wedge x: ?Y$  show  $m\text{-ivl}(?f x \nabla ?g x) \leq m\text{-ivl}(?f x)$ 
  by (metis m-ivl-anti-mono widen1)
qed
also have ...  $<$   $m\text{-ivl}(?f x) + \dots$ 
  using m-ivl-widen[OF  $\langle \neg \text{lookup } S2 \ x \sqsubseteq ?f x \rangle$ ]
  by (metis Nat.le-refl add-strict-increasing gr0I not-less0)
also have ...  $=$  ( $\sum y \in \text{insert } x \ (?X \cap ?Y). m\text{-ivl}(?f y)$ )
  using  $\langle x \sim: ?Y \rangle$  by simp
also have ...  $\leq$  ( $\sum x \in ?X. m\text{-ivl}(?f x)$ )
  by(rule sum-mono2)(insert  $\langle x: ?X \rangle$ , auto)
finally show ?thesis .
qed
} with assms show ?thesis
by(auto simp: le-st-def widen-st-def m-st-def Int-def)
qed

```

definition $n\text{-st } m \ X \ st = (\sum x \in X. m(\text{lookup } st \ x))$

lemma $n\text{-st-mono}$: **assumes** $\text{set}(\text{dom } S1) \subseteq X \ \text{set}(\text{dom } S2) \subseteq X \ S1 \sqsubseteq S2$
shows $n\text{-st } n\text{-ivl } X \ S1 \leq n\text{-st } n\text{-ivl } X \ S2$

proof–
have ($\sum x \in X. n\text{-ivl}(\text{lookup } S1 \ x) \leq \sum x \in X. n\text{-ivl}(\text{lookup } S2 \ x)$)
apply(rule sum-mono) **using** assms
by(auto simp: le-st-def lookup-def n-ivl-mono split: if-splits)
thus ?thesis **by**(simp add: n-st-def)

qed

lemma $n\text{-st-narrow}$:

assumes $\text{finite } X$ **and** $\text{set}(\text{dom } S1) \subseteq X \ \text{set}(\text{dom } S2) \subseteq X$
and $S2 \sqsubseteq S1 \neg S1 \sqsubseteq S1 \triangle S2$

shows $n\text{-st } n\text{-ivl } X \ (S1 \triangle S2) < n\text{-st } n\text{-ivl } X \ S1$

proof–

have 1: $\forall x \in X. n\text{-ivl}(\text{lookup } (S1 \triangle S2) \ x) \leq n\text{-ivl}(\text{lookup } S1 \ x)$
using assms(2–4)
by(auto simp: le-st-def narrow-st-def lookup-def n-ivl-mono narrow2
split: if-splits)

have 2: $\exists x \in X. n\text{-ivl}(\text{lookup } (S1 \triangle S2) \ x) < n\text{-ivl}(\text{lookup } S1 \ x)$
using assms(2–5)

by(auto simp: le-st-def narrow-st-def lookup-def intro: n-ivl-narrow
split: if-splits)

have ($\sum x \in X. n\text{-ivl}(\text{lookup } (S1 \triangle S2) \ x) < \sum x \in X. n\text{-ivl}(\text{lookup } S1 \ x)$)
apply(rule sum-strict-mono-ex1[OF $\langle \text{finite } X \rangle$]) **using** 1 2 **by** blast+
thus ?thesis **by**(simp add: n-st-def)

qed

9.9 Termination: Option

definition $m\text{-}o\ m\ n\ opt = (case\ opt\ of\ None \Rightarrow n+1 \mid Some\ x \Rightarrow m\ x)$

lemma $m\text{-}o\text{-}anti\text{-}mono$: $finite\ X \Longrightarrow domo\ S2 \subseteq X \Longrightarrow S1 \sqsubseteq S2 \Longrightarrow$
 $m\text{-}o\ (m\text{-}st\ m\text{-}ivl)\ (2 * card\ X)\ S2 \leq m\text{-}o\ (m\text{-}st\ m\text{-}ivl)\ (2 * card\ X)\ S1$
apply($induction\ S1\ S2\ rule: le\text{-}option.induct$)
apply($auto\ simp: domo\text{-}def\ m\text{-}o\text{-}def\ m\text{-}st\text{-}anti\text{-}mono\ le\text{-}SucI\ m\text{-}st\text{-}height$
 $split: option.splits$)
done

lemma $m\text{-}o\text{-}widen$: $\llbracket finite\ X; domo\ S2 \subseteq X; \neg S2 \sqsubseteq S1 \rrbracket \Longrightarrow$
 $m\text{-}o\ (m\text{-}st\ m\text{-}ivl)\ (2 * card\ X)\ (S1 \nabla S2) < m\text{-}o\ (m\text{-}st\ m\text{-}ivl)\ (2 * card\ X)\ S1$
by($auto\ simp: m\text{-}o\text{-}def\ domo\text{-}def\ m\text{-}st\text{-}height\ less\text{-}Suc\text{-}eq\text{-}le\ m\text{-}st\text{-}widen$
 $split: option.split$)

definition $n\text{-}o\ n\ opt = (case\ opt\ of\ None \Rightarrow 0 \mid Some\ x \Rightarrow n\ x + 1)$

lemma $n\text{-}o\text{-}mono$: $domo\ S1 \subseteq X \Longrightarrow domo\ S2 \subseteq X \Longrightarrow S1 \sqsubseteq S2 \Longrightarrow$
 $n\text{-}o\ (n\text{-}st\ n\text{-}ivl\ X)\ S1 \leq n\text{-}o\ (n\text{-}st\ n\text{-}ivl\ X)\ S2$
apply($induction\ S1\ S2\ rule: le\text{-}option.induct$)
apply($auto\ simp: domo\text{-}def\ n\text{-}o\text{-}def\ n\text{-}st\text{-}mono$
 $split: option.splits$)
done

lemma $n\text{-}o\text{-}narrow$:
 $\llbracket finite\ X; domo\ S1 \subseteq X; domo\ S2 \subseteq X; S2 \sqsubseteq S1; \neg S1 \sqsubseteq S1 \triangle S2 \rrbracket$
 $\Longrightarrow n\text{-}o\ (n\text{-}st\ n\text{-}ivl\ X)\ (S1 \triangle S2) < n\text{-}o\ (n\text{-}st\ n\text{-}ivl\ X)\ S1$
apply($induction\ S1\ S2\ rule: narrow\text{-}option.induct$)
apply($auto\ simp: n\text{-}o\text{-}def\ domo\text{-}def\ n\text{-}st\text{-}narrow$)
done

lemma $domo\text{-}widen\text{-}subset$: $domo\ (S1 \nabla S2) \subseteq domo\ S1 \cup domo\ S2$
apply($induction\ S1\ S2\ rule: widen\text{-}option.induct$)
apply ($auto\ simp: domo\text{-}def\ widen\text{-}st\text{-}def$)
done

lemma $domo\text{-}narrow\text{-}subset$: $domo\ (S1 \triangle S2) \subseteq domo\ S1 \cup domo\ S2$
apply($induction\ S1\ S2\ rule: narrow\text{-}option.induct$)
apply ($auto\ simp: domo\text{-}def\ narrow\text{-}st\text{-}def$)
done

9.10 Termination: Commands

lemma $strip\text{-}widen\text{-}acom$ [$simp$]:
 $strip\ c' = strip\ (c::'a::WN\ acom) \Longrightarrow strip\ (c \nabla_c\ c') = strip\ c$
by($induction\ widen::'a \Rightarrow 'a \Rightarrow 'a\ c\ c'\ rule: map2\text{-}acom.induct$) $simp\text{-}all$

lemma *strip-narrow-acom*[simp]:

$strip\ c' = strip\ (c::'a::WN\ acom) \implies strip\ (c\ \Delta_c\ c') = strip\ c$
by(*induction narrow::'a=>'a=>'a c c' rule: map2-acom.induct*) *simp-all*

lemma *annos-widen-acom*[simp]: $strip\ c1 = strip\ (c2::'a::WN\ acom) \implies$

$annos(c1\ \nabla_c\ c2) = map\ (\% (x,y).x\nabla y)\ (zip\ (annos\ c1)\ (annos(c2::'a::WN\ acom)))$
by(*induction widen::'a=>'a=>'a c1 c2 rule: map2-acom.induct*)
(simp-all add: size-annos-same2)

lemma *annos-narrow-acom*[simp]: $strip\ c1 = strip\ (c2::'a::WN\ acom) \implies$

$annos(c1\ \Delta_c\ c2) = map\ (\% (x,y).x\Delta y)\ (zip\ (annos\ c1)\ (annos(c2::'a::WN\ acom)))$
by(*induction narrow::'a=>'a=>'a c1 c2 rule: map2-acom.induct*)
(simp-all add: size-annos-same2)

lemma *widen-acom-Com*[simp]: $strip\ c2 = strip\ c1 \implies$

$c1 : Com\ X \implies c2 : Com\ X \implies (c1\ \nabla_c\ c2) : Com\ X$
apply(*auto simp add: Com-def*)
apply(*rename-tac S S' x*)
apply(*erule in-set-zipE*)
apply(*auto simp: domo-def split: option.splits*)
apply(*case-tac S*)
apply(*case-tac S'*)
apply *simp*
apply *fastforce*
apply(*case-tac S'*)
apply *fastforce*
apply (*fastforce simp: widen-st-def*)
done

lemma *narrow-acom-Com*[simp]: $strip\ c2 = strip\ c1 \implies$

$c1 : Com\ X \implies c2 : Com\ X \implies (c1\ \Delta_c\ c2) : Com\ X$
apply(*auto simp add: Com-def*)
apply(*rename-tac S S' x*)
apply(*erule in-set-zipE*)
apply(*auto simp: domo-def split: option.splits*)
apply(*case-tac S*)
apply(*case-tac S'*)
apply *simp*
apply *fastforce*
apply(*case-tac S'*)
apply *fastforce*
apply (*fastforce simp: narrow-st-def*)
done

definition *m-c m c* = (*let as = annos c in $\sum_{i=0..<size\ as} m(as!i)$*)

lemma *measure-m-c*: $finite\ X \implies \{(c, c\ \nabla_c\ c') \mid c\ c'::ivl\ st\ option\ acom.\}$

$strip\ c' = strip\ c \wedge c : Com\ X \wedge c' : Com\ X \wedge \neg c' \sqsubseteq c\}^{-1}$
 $\sqsubseteq measure(m-c(m-o\ (m-st\ m-ivl)\ (2*card(X))))$


```

apply(auto simp: m-c-def Let-def Com-def)
apply(subgoal-tac length(annos c') = length(annos c))
prefer 2 apply (simp add: size-annos-same2)
apply (auto)
apply(rule sum-strict-mono-ex1)
apply simp
apply (clarsimp)
apply(erule m-o-anti-mono)
apply(rule subset-trans[OF domo-widen-subset])
apply fastforce
apply(rule widen1)
apply(auto simp: le-iff-le-annos listrel-iff-nth)
apply(rule-tac x=n in be1)
prefer 2 apply simp
apply(erule m-o-widen)
apply (simp)+
done

```

```

lemma measure-n-c: finite X  $\implies \{(c, c \Delta_c c') \mid c c'\}^{-1}$ 
  strip c = strip c'  $\wedge c \in \text{Com } X \wedge c' \in \text{Com } X \wedge c' \sqsubseteq c \wedge \neg c \sqsubseteq c \Delta_c c'\}$ -1
   $\subseteq \text{measure}(m-c(n-o (n-st n-ivl X)))$ 
apply(auto simp: m-c-def Let-def Com-def)
apply(subgoal-tac length(annos c') = length(annos c))
prefer 2 apply (simp add: size-annos-same2)
apply (auto)
apply(rule sum-strict-mono-ex1)
apply simp
apply (clarsimp)
apply(rule n-o-mono)
using domo-narrow-subset apply fastforce
apply fastforce
apply(rule narrow2)
apply(fastforce simp: le-iff-le-annos listrel-iff-nth)
apply(auto simp: le-iff-le-annos listrel-iff-nth)
apply(rule-tac x=n in be1)
prefer 2 apply simp
apply(erule n-o-narrow)
apply (simp)+
done

```

9.11 Termination: Post-Fixed Point Iterations

```

lemma iter-widen-termination:
fixes c0 :: 'a::WN acom
assumes P-f:  $\bigwedge c. P c \implies P(f c)$ 
assumes P-widen:  $\bigwedge c c'. P c \implies P c' \implies P(c \nabla_c c')$ 
and wf( $\{(c::'a acom, c \nabla_c c') \mid c c'. P c \wedge P c' \wedge \sim c' \sqsubseteq c\}^{-1}$ )
and P c0 and c0  $\sqsubseteq f c0$  shows  $\exists c. \text{iter-widen } f c0 = \text{Some } c$ 
proof(simp add: iter-widen-def, rule wf-while-option-Some[where P = P])

```

```

  show wf  $\{(cc', c). (P\ c \wedge \neg f\ c \sqsubseteq c) \wedge cc' = c \nabla_c f\ c\}$ 
    apply(rule wf-subset[OF assms(3)]) by(blast intro: P-f)
next
  show  $P\ c0$  by(rule  $\langle P\ c0 \rangle$ )
next
  fix  $c$  assume  $P\ c$  thus  $P\ (c \nabla_c f\ c)$  by(simp add: P-f P-widen)
qed

```

```

lemma iter-narrow-termination:
  assumes P-f:  $\bigwedge c. P\ c \implies P(c \Delta_c f\ c)$ 
  and wf:  $wf(\{(c, c \Delta_c f\ c) \mid c\ c'. P\ c \wedge \sim c \sqsubseteq c \Delta_c f\ c\}^{\wedge-1})$ 
  and  $P\ c0$  shows  $\exists c. \text{iter-narrow } f\ c0 = \text{Some } c$ 
proof(simp add: iter-narrow-def, rule wf-while-option-Some[where  $P = P$ ])
  show wf  $\{(c', c). (P\ c \wedge \neg c \sqsubseteq c \Delta_c f\ c) \wedge c' = c \Delta_c f\ c\}$ 
    apply(rule wf-subset[OF wf]) by(blast intro: P-f)
next
  show  $P\ c0$  by(rule  $\langle P\ c0 \rangle$ )
next
  fix  $c$  assume  $P\ c$  thus  $P\ (c \Delta_c f\ c)$  by(simp add: P-f)
qed

```

```

lemma iter-widen-step-ivl-termination:
   $\exists c. \text{iter-widen } (\text{step-ivl } \top) (\perp_c\ c0) = \text{Some } c$ 
  apply(rule iter-widen-termination[where
     $P = \%c. \text{strip } c = c0 \wedge c : \text{Com}(\text{vars } c0)$ ])
  apply(simp-all add: step'-Com bot-acom)
  apply(rule wf-subset)
  apply(rule wf-measure)
  apply(rule subset-trans)
  prefer 2
  apply(rule measure-m-c[where  $X = \text{vars } c0$ , OF finite-cvars])
  apply blast
done

```

```

lemma iter-narrow-step-ivl-termination:
   $c0 \in \text{Com } (\text{vars}(\text{strip } c0)) \implies \text{step-ivl } \top\ c0 \sqsubseteq c0 \implies$ 
   $\exists c. \text{iter-narrow } (\text{step-ivl } \top)\ c0 = \text{Some } c$ 
  apply(rule iter-narrow-termination[where
     $P = \%c. \text{strip } c = \text{strip } c0 \wedge c : \text{Com}(\text{vars}(\text{strip } c0)) \wedge \text{step-ivl } \top\ c \sqsubseteq c$ ])
  apply(simp-all add: step'-Com)
  apply(clarify)
  apply(frule narrow2-acom, drule mono-step'[OF le-refl], erule le-trans[OF - narrow1-acom])
  apply assumption
  apply(rule wf-subset)
  apply(rule wf-measure)
  apply(rule subset-trans)
  prefer 2
  apply(rule measure-n-c[where  $X = \text{vars}(\text{strip } c0)$ , OF finite-cvars])

```

apply *auto*
by (*metis bot-least domo-Top order-refl step'-Com strip-step'*)

lemma *while-Com*:
fixes $c :: 'a \text{ st option acom}$
assumes *while-option* $P f c = \text{Some } c'$
and $!!c. \text{strip}(f c) = \text{strip } c$
and $\forall c::'a \text{ st option acom}. c : \text{Com}(X) \longrightarrow \text{vars}(\text{strip } c) \subseteq X \longrightarrow f c : \text{Com}(X)$
and $c : \text{Com}(X)$ **and** $\text{vars}(\text{strip } c) \subseteq X$ **shows** $c' : \text{Com}(X)$
using *while-option-rule*[**where** $P = \lambda c'. c' : \text{Com}(X) \wedge \text{vars}(\text{strip } c') \subseteq X$, *OF - assms(1)*]
by(*simp add: assms(2-)*)

lemma *iter-widen-Com*: **fixes** $f :: 'a::\text{WN st option acom} \Rightarrow 'a \text{ st option acom}$
assumes *iter-widen* $f c = \text{Some } c'$
and $\forall c. c : \text{Com}(X) \longrightarrow \text{vars}(\text{strip } c) \subseteq X \longrightarrow f c : \text{Com}(X)$
and $!!c. \text{strip}(f c) = \text{strip } c$
and $c : \text{Com}(X)$ **and** $\text{vars}(\text{strip } c) \subseteq X$ **shows** $c' : \text{Com}(X)$
proof -
have $\forall c. c : \text{Com}(X) \longrightarrow \text{vars}(\text{strip } c) \subseteq X \longrightarrow c \nabla_c f c : \text{Com}(X)$
by (*metis (full-types) widen-acom-Com assms(2,3)*)
from *while-Com*[*OF assms(1)*][*simplified iter-widen-def*] - *this assms(4,5)*
show *?thesis using assms(3)* **by**(*simp*)
qed

context *Abs-Int2*
begin

lemma *iter-widen-step'-Com*:
 $\text{iter-widen } (\text{step}' \top) c = \text{Some } c' \Longrightarrow \text{vars}(\text{strip } c) \subseteq X \Longrightarrow c : \text{Com}(X)$
 $\Longrightarrow c' : \text{Com}(X)$
apply(*subgoal-tac strip c'= strip c*)
prefer 2 **apply** (*metis strip-iter-widen strip-step'*)
apply(*drule iter-widen-Com*)
prefer 3 **apply** *assumption*
prefer 3 **apply** *assumption*
apply (*auto simp: step'-Com*)
done

end

theorem *AI-ivl'-termination*:
 $\exists c'. \text{AI-ivl}' c = \text{Some } c'$
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 (*metis bot-acom-Com iter-widen-step'-Com*[*OF - subset-refl*] *strip-iter-widen*)

```
strip-step')
apply(erule iter-widen-pfp)
done

end
```

References

- [1] T. Nipkow. Abstract interpretation of annotated commands. In Beringer and Felty, editors, *Interactive Theorem Proving (ITP 2012)*, volume 7406 of *LNCS*, pages 116–132. Springer, 2012.
- [2] T. Nipkow and G. Klein. *Concrete Semantics with Isabelle/HOL*. Springer, 2014. 298 pp. <http://concrete-semantics.org>.