

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-lfp*: $lfp\ f\ i : L\ i$
 $\langle proof \rangle$

lemma *lfp-lowerbound*:
 $\llbracket a : L\ i;\ f\ a \leq a \rrbracket \Longrightarrow lfp\ f\ i \leq a$
 $\langle proof \rangle$

lemma *lfp-greatest*:
 $\llbracket a : L\ i;\ \bigwedge u.\ \llbracket u : L\ i;\ f\ u \leq u \rrbracket \Longrightarrow a \leq u \rrbracket \Longrightarrow a \leq lfp\ f\ i$
 $\langle proof \rangle$

lemma *lfp-unfold*: **assumes** $\bigwedge x\ i.\ f\ x : L\ i \longleftrightarrow x : L\ i$
and *mono*: *mono* f **shows** $lfp\ f\ i = f\ (lfp\ f\ i)$
 $\langle proof \rangle$

end

end

2 Annotated Commands

theory *ACom*
imports *HOL-IMP.Com*
begin

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

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

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} |
anno a (x ::= e) = (x ::= e {a}) |
anno a (c1;;c2) = (anno a c1;; anno a c2) |
anno a (IF b THEN c1 ELSE c2) =
 (IF b THEN anno a c1 ELSE anno a c2 {a}) |
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] |
annos (x ::= e {a}) = [a] |
annos (C1;;C2) = annos C1 @ annos C2 |
annos (IF b THEN C1 ELSE C2 {a}) = a # annos C1 @ annos C2 |
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} |
map-acom f (x ::= e {P}) = (x ::= e {f P}) |
map-acom f (c1;;c2) = (map-acom f c1;; map-acom f c2) |
map-acom f (IF b THEN c1 ELSE c2 {P}) =
 (IF b THEN map-acom f c1 ELSE map-acom f c2 {f P}) |
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]*: $post(map-acom\ f\ c) = f(post\ c)$
 <proof>

lemma *strip-acom[simp]*: $strip\ (map-acom\ f\ c) = strip\ c$
 <proof>

lemma *map-acom-SKIP*:
 $map-acom\ f\ c = SKIP\ \{S'\} \longleftrightarrow (\exists S. c = SKIP\ \{S\} \wedge S' = f\ S)$
 <proof>

lemma *map-acom-Assign*:
 $map-acom\ f\ c = x ::= e\ \{S'\} \longleftrightarrow (\exists S. c = x ::= e\ \{S\} \wedge S' = f\ S)$
 <proof>

lemma *map-acom-Seq*:
 $map-acom\ f\ c = c1';;c2' \longleftrightarrow$
 $(\exists c1\ c2. c = c1;;c2 \wedge map-acom\ f\ c1 = c1' \wedge map-acom\ f\ c2 = c2')$
 <proof>

lemma *map-acom-If*:

$map\text{-}acom\ f\ c = IF\ b\ THEN\ c1'\ ELSE\ c2'\ \{S'\} \longleftrightarrow$
 $(\exists S\ c1\ c2. c = IF\ b\ THEN\ c1\ ELSE\ c2\ \{S\} \wedge map\text{-}acom\ f\ c1 = c1' \wedge map\text{-}acom$
 $f\ c2 = c2' \wedge S' = f\ S)$
 <proof>

lemma *map-acom-While*:

$map\text{-}acom\ f\ w = \{I'\}\ WHILE\ b\ DO\ c'\ \{P'\} \longleftrightarrow$
 $(\exists I\ P\ c. w = \{I\}\ WHILE\ b\ DO\ c\ \{P\} \wedge map\text{-}acom\ f\ c = c' \wedge I' = f\ I \wedge P' = f$
 $P)$
 <proof>

lemma *strip-anno[simp]*: $strip\ (anno\ a\ c) = c$
 <proof>

lemma *strip-eq-SKIP*:

$strip\ c = com.SKIP \longleftrightarrow (\exists P. c = SKIP\ \{P\})$
 <proof>

lemma *strip-eq-Assign*:

$strip\ c = x::=e \longleftrightarrow (\exists P. c = x::=e\ \{P\})$
 <proof>

lemma *strip-eq-Seq*:

$strip\ c = c1;;c2 \longleftrightarrow (\exists d1\ d2. c = d1;;d2 \ \&\ strip\ d1 = c1 \ \&\ strip\ d2 = c2)$
 <proof>

lemma *strip-eq-If*:

$strip\ c = IF\ b\ THEN\ c1\ ELSE\ c2 \longleftrightarrow$
 $(\exists d1\ d2\ P. c = IF\ b\ THEN\ d1\ ELSE\ d2\ \{P\} \ \&\ strip\ d1 = c1 \ \&\ strip\ d2 = c2)$
 <proof>

lemma *strip-eq-While*:

$strip\ c = WHILE\ b\ DO\ c1 \longleftrightarrow$
 $(\exists I\ d1\ P. c = \{I\}\ WHILE\ b\ DO\ d1\ \{P\} \ \&\ strip\ d1 = c1)$
 <proof>

lemma *set-annos-anno[simp]*: $set\ (annos\ (anno\ a\ C)) = \{a\}$
 <proof>

lemma *size-annos-same*: $strip\ C1 = strip\ C2 \implies size(annos\ C1) = size(annos$
 $C2)$
 <proof>

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')
<proof>
```

```
lemma Assign-le: x ::= e {S}  $\leq$  c  $\longleftrightarrow$  ( $\exists$  S'. c = x ::= e {S'}  $\wedge$  S  $\leq$  S')
<proof>
```

```
lemma Seq-le: c1;;c2  $\leq$  c  $\longleftrightarrow$  ( $\exists$  c1' c2'. c = c1';;c2'  $\wedge$  c1  $\leq$  c1'  $\wedge$  c2  $\leq$  c2')
<proof>
```

```
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')
<proof>
```

```
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')
<proof>
```

```
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>
```

```
end
```

```
fun sub1 :: 'a acom  $\Rightarrow$  'a acom where
sub1(c1;;c2) = c1 |
```

$sub_1(IF\ b\ THEN\ c1\ ELSE\ c2\ \{S\}) = c1 \mid$
 $sub_1(\{I\}\ WHILE\ b\ DO\ c\ \{P\}) = c$

fun $sub_2 :: 'a\ acom \Rightarrow 'a\ acom$ **where**
 $sub_2(c1;;c2) = c2 \mid$
 $sub_2(IF\ b\ THEN\ c1\ ELSE\ c2\ \{S\}) = c2$

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

$lift\ F\ com.SKIP\ M = (SKIP\ \{F\ (post\ 'M)\}) \mid$
 $lift\ F\ (x\ ::=\ a)\ M = (x\ ::=\ a\ \{F\ (post\ 'M)\}) \mid$
 $lift\ F\ (c1;;c2)\ M =$
 $\quad lift\ F\ c1\ (sub_1\ 'M);; lift\ F\ c2\ (sub_2\ 'M) \mid$
 $lift\ F\ (IF\ b\ THEN\ c1\ ELSE\ c2)\ M =$
 $\quad IF\ b\ THEN\ lift\ F\ c1\ (sub_1\ 'M)\ ELSE\ lift\ F\ c2\ (sub_2\ 'M)$
 $\quad \{F\ (post\ 'M)\} \mid$
 $lift\ F\ (WHILE\ b\ DO\ c)\ M =$
 $\quad \{F\ (invar\ 'M)\}$
 $\quad WHILE\ b\ DO\ lift\ F\ c\ (sub_1\ 'M)$
 $\quad \{F\ (post\ 'M)\}$

global-interpretation $Complete-Lattice-ix\ \%c.\ \{c'.\ strip\ c' = c\}\ lift\ Inter$
 $\langle proof \rangle$

lemma $le-post: c \leq d \Longrightarrow post\ c \leq post\ d$
 $\langle proof \rangle$

3.2 Collecting semantics

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

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

lemma $mono2-step: c1 \leq c2 \Longrightarrow S1 \subseteq S2 \Longrightarrow step\ S1\ c1 \leq step\ S2\ c2$
 $\langle proof \rangle$

```

lemma mono-step: mono (step S)
⟨proof⟩

lemma strip-step: strip(step S c) = strip c
⟨proof⟩

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

lemma CS-unfold: CS c = step UNIV (CS c)
⟨proof⟩

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

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  $\sqsubseteq$  50)
assumes le-refl[simp]:  $x \sqsubseteq x$ 
and le-trans:  $x \sqsubseteq y \Longrightarrow y \sqsubseteq z \Longrightarrow x \sqsubseteq z$ 
begin

definition mono where mono f =  $(\forall x y. x \sqsubseteq y \longrightarrow f x \sqsubseteq f y)$ 

lemma monoD: mono f  $\Longrightarrow x \sqsubseteq y \Longrightarrow f x \sqsubseteq f y$  ⟨proof⟩

lemma mono-comp: mono f  $\Longrightarrow$  mono g  $\Longrightarrow$  mono (g o f)
⟨proof⟩

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 $x \sqsubseteq y$ and $y \sqsubseteq x$. Antisymmetry is not needed because we never compare elements for equality but only for \sqsubseteq .

```

class SL-top = preord +

```

```

fixes join :: 'a ⇒ 'a ⇒ 'a (infixl  $\sqcup$  65)
fixes Top :: 'a ( $\top$ )
assumes join-ge1 [simp]:  $x \sqsubseteq x \sqcup y$ 
and join-ge2 [simp]:  $y \sqsubseteq x \sqcup y$ 
and join-least:  $x \sqsubseteq z \implies y \sqsubseteq z \implies x \sqcup y \sqsubseteq z$ 
and top[simp]:  $x \sqsubseteq \top$ 
begin

lemma join-le-iff[simp]:  $x \sqcup y \sqsubseteq z \iff x \sqsubseteq z \wedge y \sqsubseteq z$ 
<proof>

lemma le-join-disj:  $x \sqsubseteq y \vee x \sqsubseteq z \implies x \sqsubseteq y \sqcup z$ 
<proof>

end

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

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

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

instance
<proof>

end

instantiation acom :: (preord) preord
begin

fun le-acom :: ('a::preord)acom ⇒ 'a acom ⇒ 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]:  $SKIP\ \{S\} \sqsubseteq c \iff (\exists S'. c = SKIP\ \{S'\} \wedge S \sqsubseteq S')$ 
<proof>

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

```


<proof>

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

lemma [*simp*]: $IF\ b\ THEN\ c1\ ELSE\ c2\ \{S\} \sqsubseteq c \longleftrightarrow$
 $(\exists c1' c2' S'. c = IF\ b\ THEN\ c1'\ ELSE\ c2'\ \{S'\} \wedge c1 \sqsubseteq c1' \wedge c2 \sqsubseteq c2' \wedge S \sqsubseteq S')$
<proof>

lemma [*simp*]: $\{Inv\}\ WHILE\ b\ DO\ c\ \{P\} \sqsubseteq w \longleftrightarrow$
 $(\exists Inv' c' P'. w = \{Inv'\}\ WHILE\ b\ DO\ c'\ \{P'\} \wedge c \sqsubseteq c' \wedge Inv \sqsubseteq Inv' \wedge P \sqsubseteq P')$
<proof>

instance

<proof>

end

4.1.1 Lifting

instantiation *option* :: (*preord*)*preord*
begin

fun *le-option* **where**

Some $x \sqsubseteq$ *Some* $y = (x \sqsubseteq y) \mid$

None $\sqsubseteq y = True \mid$

Some $- \sqsubseteq None = False$

lemma [*simp*]: $(x \sqsubseteq None) = (x = None)$

<proof>

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

<proof>

instance

<proof>

end

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

begin

fun *join-option* **where**

Some $x \sqcup$ *Some* $y = Some(x \sqcup y) \mid$

None $\sqcup y = y \mid$

$x \sqcup None = x$

lemma *join-None2[simp]*: $x \sqcup \text{None} = x$
(*proof*)

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

instance
(*proof*)

end

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

lemma *strip-bot-acom[simp]*: $\text{strip}(\perp_c \ c) = c$
(*proof*)

lemma *bot-acom[rule-format]*: $\text{strip } c' = c \longrightarrow \perp_c \ c \sqsubseteq c'$
(*proof*)

4.1.2 Post-fixed point iteration

definition
 $\text{pfp} :: ('a::\text{preord}) \Rightarrow 'a \Rightarrow 'a \text{ option}$ **where**
 $\text{pfp } f = \text{while-option } (\lambda x. \neg f \ x \sqsubseteq x) \ f$

lemma *pfp-pfp*: **assumes** $\text{pfp } f \ x0 = \text{Some } x$ **shows** $f \ x \sqsubseteq x$
(*proof*)

lemma *pfp-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** $\text{pfp } f \ x0 = \text{Some } x$ **shows** $x \sqsubseteq p$
(*proof*)

definition
 $\text{lpfp}_c :: ('a::\text{SL-top})\text{option } \text{acom} \Rightarrow 'a \text{ option } \text{acom} \Rightarrow \text{com} \Rightarrow 'a \text{ option } \text{acom}$
option **where**
 $\text{lpfp}_c \ f \ c = \text{pfp } f \ (\perp_c \ c)$

lemma *lpfp-pfp*: $\text{lpfp}_c \ f \ c0 = \text{Some } c \Longrightarrow f \ c \sqsubseteq c$
(*proof*)

lemma *strip-pfp*:
assumes $\bigwedge x. g(f \ x) = g \ x$ **and** $\text{pfp } f \ x0 = \text{Some } x$ **shows** $g \ x = g \ x0$
(*proof*)

lemma *strip-lpfp*: **assumes** $\bigwedge c. \text{strip}(f \ c) = \text{strip } c$ **and** $\text{lpfp}_c \ f \ c = \text{Some } c'$
shows $\text{strip } c' = c$
(*proof*)

lemma *lpfpc-least*:

assumes *mono*: $\bigwedge x y. x \sqsubseteq y \implies f x \sqsubseteq f y$

and *strip* $p = c0$ **and** $f p \sqsubseteq p$ **and** *lp*: $lpfp_c f c0 = \text{Some } c$ **shows** $c \sqsubseteq p$
<proof>

4.2 Abstract Interpretation

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

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

The interface for abstract values:

locale *Val-abs* =

fixes $\gamma :: 'av::SL\text{-top} \Rightarrow \text{val set}$

assumes *mono-gamma*: $a \sqsubseteq b \implies \gamma a \subseteq \gamma b$

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

fixes $\text{num}' :: \text{val} \Rightarrow 'av$

and $\text{plus}' :: 'av \Rightarrow 'av \Rightarrow 'av$

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

and *gamma-plus'*:

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

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

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

fun $\text{aval}' :: \text{aexp} \Rightarrow 'av \text{ st} \Rightarrow 'av$ **where**

$\text{aval}' (N n) S = \text{num}' n$ |

$\text{aval}' (V x) S = S x$ |

$\text{aval}' (\text{Plus } a1 a2) S = \text{plus}' (\text{aval}' a1 S) (\text{aval}' a2 S)$

fun $\text{step}' :: 'av \text{ st option} \Rightarrow 'av \text{ st option acom} \Rightarrow 'av \text{ st option acom}$
where

$\text{step}' S (\text{SKIP } \{P\}) = (\text{SKIP } \{S\})$ |

$\text{step}' S (x ::= e \{P\}) =$

$x ::= e \{ \text{case } S \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } S \Rightarrow \text{Some}(S(x ::= \text{aval}' e S)) \}$ |

$\text{step}' S (c1 ;; c2) = \text{step}' S c1 ;; \text{step}' (\text{post } c1) c2$ |

$\text{step}' S (\text{IF } b \text{ THEN } c1 \text{ ELSE } c2 \{P\}) =$

$\text{IF } b \text{ THEN } \text{step}' S c1 \text{ ELSE } \text{step}' S c2 \{ \text{post } c1 \sqcup \text{post } c2 \}$ |

$\text{step}' S (\{Inv\} \text{ WHILE } b \text{ DO } c \{P\}) =$

$\{S \sqcup \text{post } c\} \text{ WHILE } b \text{ DO } (\text{step}' \text{Inv } c) \{Inv\}$

definition *AI* :: $\text{com} \Rightarrow 'av \text{ st option acom option}$ **where**

$AI = lpfp_c (\text{step}' \top)$

lemma *strip-step'[simp]*: $\text{strip}(\text{step}' S c) = \text{strip } c$
<proof>

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

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]*: $\gamma_f \text{ Top} = \text{UNIV}$
<proof>

lemma *gamma-o-Top[simp]*: $\gamma_o \text{ Top} = \text{UNIV}$
<proof>

lemma *mono-gamma-f*: $f \sqsubseteq g \Longrightarrow \gamma_f f \subseteq \gamma_f g$
<proof>

lemma *mono-gamma-o*:
 $sa \sqsubseteq sa' \Longrightarrow \gamma_o sa \subseteq \gamma_o sa'$
<proof>

lemma *mono-gamma-c*: $ca \sqsubseteq ca' \Longrightarrow \gamma_c ca \leq \gamma_c ca'$
<proof>

Soundness:

lemma *aval'-sound*: $s : \gamma_f S \Longrightarrow \text{aval } a s : \gamma(\text{aval}' a S)$
<proof>

lemma *in-gamma-update*:
 $\llbracket s : \gamma_f S; i : \gamma a \rrbracket \Longrightarrow s(x := i) : \gamma_f(S(x := a))$
<proof>

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>

lemma *AI-sound*: $\text{AI } c = \text{Some } c' \Longrightarrow \text{CS } c \leq \gamma_c c'$
<proof>

end

4.2.1 Monotonicity

lemma *mono-post*: $c \sqsubseteq c' \implies \text{post } c \sqsubseteq \text{post } c'$
<proof>

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

lemma *mono-aval'*: $S \sqsubseteq S' \implies \text{aval}' e S \sqsubseteq \text{aval}' e S'$
<proof>

lemma *mono-update*: $a \sqsubseteq a' \implies S \sqsubseteq S' \implies S(x := a) \sqsubseteq S'(x := a')$
<proof>

lemma *mono-step'*: $S \sqsubseteq S' \implies c \sqsubseteq c' \implies \text{step}' S c \sqsubseteq \text{step}' S' c'$
<proof>

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* ← *xs*. *x* ∈ *set ys*]

definition *show-st* *S* = [*x*, *fun* *S* *x*]. *x* ← *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* ∈ *set(dom F)* *then dom F* *else* *x* # *dom F*)*

lemma *lookup-update*: $lookup (update S x y) = (lookup S)(x:=y)$
<proof>

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

instantiation $st :: (SL\text{-top}) SL\text{-top}$
begin

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

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

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

instance
<proof>

end

lemma *mono-lookup*: $F \sqsubseteq F' \implies lookup F x \sqsubseteq lookup F' x$
<proof>

lemma *mono-update*: $a \sqsubseteq a' \implies S \sqsubseteq S' \implies update S x a \sqsubseteq update S' x a'$
<proof>

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

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

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]*: $\gamma_f Top = UNIV$
<proof>

lemma *gamma-o-Top[simp]*: $\gamma_o Top = UNIV$
<proof>

lemma *mono-gamma-f*: $f \sqsubseteq g \implies \gamma_f f \subseteq \gamma_f g$

<proof>

lemma *mono-gamma-o*:

$$sa \sqsubseteq sa' \implies \gamma_o sa \subseteq \gamma_o sa'$$

<proof>

lemma *mono-gamma-c*: $ca \sqsubseteq ca' \implies \gamma_c ca \leq \gamma_c ca'$

<proof>

lemma *in-gamma-option-iff*:

$$x : \gamma\text{-option } r u \longleftrightarrow (\exists u'. u = \text{Some } u' \wedge x : r u')$$

<proof>

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*: $s : \gamma_f S \implies \text{aval } a s : \gamma(\text{aval}' a S)$

<proof>

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** $\gamma :: \text{'av} :: \text{SL-top} \Rightarrow \text{val set}$

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}*) =

$x ::= e \{ \text{case } S \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } S \Rightarrow \text{Some}(\text{update } S x (\text{aval}' e S)) \}$ |

step' *S* (*c1;; c2*) = *step'* *S* *c1;; step'* (*post c1*) *c2* |

step' *S* (*IF b THEN c1 ELSE c2 {P}*) =

(*let c1' = step'* *S* *c1; c2' = step'* *S* *c2*

$$\text{in IF } b \text{ THEN } c1' \text{ ELSE } c2' \{ \text{post } c1 \sqcup \text{post } c2 \} \mid$$

$$\text{step}' S (\{ \text{Inv} \} \text{ WHILE } b \text{ DO } c \{ P \}) =$$

$$\{ S \sqcup \text{post } c \} \text{ WHILE } b \text{ DO step}' \text{Inv } c \{ \text{Inv} \}$$

definition $AI :: \text{com} \Rightarrow 'av \text{ st option acom option where}$
 $AI = \text{lfp}_c (\text{step}' \top)$

lemma $\text{strip-step}'[\text{simp}]: \text{strip}(\text{step}' S c) = \text{strip } c$
 $\langle \text{proof} \rangle$

Soundness:

lemma in-gamma-update:
 $\llbracket s : \gamma_f S; i : \gamma_a a \rrbracket \Longrightarrow s(x := i) : \gamma_f(\text{update } S x a)$
 $\langle \text{proof} \rangle$

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

lemma $\text{step-preserves-le:}$
 $\llbracket S \sqsubseteq \gamma_o S'; c \leq \gamma_c c' \rrbracket \Longrightarrow \text{step } S c \leq \gamma_c (\text{step}' S' c')$
 $\langle \text{proof} \rangle$

lemma $AI\text{-sound: } AI c = \text{Some } c' \Longrightarrow CS c \leq \gamma_c c'$
 $\langle \text{proof} \rangle$

end

6.1 Monotonicity

locale $Abs\text{-Int-mono} = Abs\text{-Int} +$
assumes $\text{mono-plus': } a1 \sqsubseteq b1 \Longrightarrow a2 \sqsubseteq b2 \Longrightarrow \text{plus}' a1 a2 \sqsubseteq \text{plus}' b1 b2$
begin

lemma $\text{mono-aval': } S \sqsubseteq S' \Longrightarrow \text{aval}' e S \sqsubseteq \text{aval}' e S'$
 $\langle \text{proof} \rangle$

lemma $\text{mono-update: } a \sqsubseteq a' \Longrightarrow S \sqsubseteq S' \Longrightarrow \text{update } S x a \sqsubseteq \text{update } S' x a'$
 $\langle \text{proof} \rangle$

lemma $\text{mono-step': } S \sqsubseteq S' \Longrightarrow c \sqsubseteq c' \Longrightarrow \text{step}' S c \sqsubseteq \text{step}' S' c'$
 $\langle \text{proof} \rangle$

end

6.2 Ascending Chain Condition

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

lemma *strict-inv-image*: $strict(inv\text{-}image\ r\ f) = inv\text{-}image\ (strict\ r)\ f$
 ⟨proof⟩

lemma *acc-inv-image*:
 $acc\ r \implies acc\ (inv\text{-}image\ r\ f)$
 ⟨proof⟩

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⟩

ACC for abstract states, via measure functions.

lemma *measure-st*: **assumes** $(strict\{(x,y::'a::SL\text{-}top). x \sqsubseteq y\})^{\sim-1} \leq measure\ m$
and $\forall x\ y::'a::SL\text{-}top. x \sqsubseteq y \wedge y \sqsubseteq x \longrightarrow m\ x = m\ y$
shows $(strict\{(S,S'::'a::SL\text{-}top\ st). S \sqsubseteq S'\})^{\sim-1} \subseteq$
 $measure(\%fd. \sum x | x \in set(dom\ fd) \wedge \sim \top \sqsubseteq fun\ fd\ x. m(fun\ fd\ x)+1)$
 ⟨proof⟩

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

lemma *listrel-Cons-iff*:
 $(x\#\!xs, y\#\!ys) : listrel\ r \longleftrightarrow (x,y) \in r \wedge (xs,ys) \in listrel\ r$
 ⟨proof⟩

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

lemma *listrel-app-same-size*: $size\ xs1 = size\ ys1 \implies size\ xs2 = size\ ys2 \implies$
 $(xs1@xs2, ys1@ys2) : listrel\ r \longleftrightarrow$
 $(xs1,ys1) : listrel\ r \wedge (xs2,ys2) : listrel\ r$
 ⟨proof⟩

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

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

lemma *le-iff-le-annos*: $c1 \sqsubseteq c2 \longleftrightarrow$
 $(annos\ c1, annos\ c2) : listrel\ \{(x,y). x \sqsubseteq y\} \wedge strip\ c1 = strip\ c2$
 ⟨proof⟩

lemma *le-acom-subset-same-annos*:

$(\text{strict}\{(c,c'::'a::\text{preord } \text{acom}). c \sqsubseteq c'\})^{\wedge-1} \subseteq$
 $(\text{strict}(\text{inv-image } (\text{listrel}\{(a,a'::'a). a \sqsubseteq a'\} - \{([],[])\}) \text{ annos}))^{\wedge-1}$
 <proof>

lemma *acc-acom*: $\text{acc } \{(a,a'::'a::\text{preord}). a \sqsubseteq a'\} \implies$
 $\text{acc } \{(c,c'::'a \text{ acom}). c \sqsubseteq c'\}$
 <proof>

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

lemma *fpf-termination*:
fixes $x0 :: 'a::\text{preord}$
assumes $\text{mono}: \bigwedge x y. x \sqsubseteq y \implies f x \sqsubseteq f y$ **and** $\text{acc } \{(x::'a,y). x \sqsubseteq y\}$
and $x0 \sqsubseteq f x0$ **shows** $\exists x. \text{fpf } f x0 = \text{Some } x$
 <proof>

lemma *lpfc-termination*:
fixes $f :: (('a::\text{SL-top}) \text{option } \text{acom} \Rightarrow 'a \text{ option } \text{acom})$
assumes $\text{acc } \{(x::'a,y). x \sqsubseteq y\}$ **and** $\bigwedge x y. x \sqsubseteq y \implies f x \sqsubseteq f y$
and $\bigwedge c. \text{strip}(f c) = \text{strip } c$
shows $\exists c'. \text{lpfc } f c = \text{Some } c'$
 <proof>

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 $\exists c'. \text{AI } c = \text{Some } c'$
 <proof>

end

end

7 Backward Analysis of Expressions

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

instantiation *prod* :: $(\text{preord}, \text{preord}) \text{ preord}$
begin

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

instance
 <proof>

end

hide-const *bot*

class *L-top-bot* = *SL-top* +
fixes *meet* :: 'a ⇒ 'a ⇒ '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*: $x \sqsubseteq y \implies x \sqsubseteq z \implies x \sqsubseteq y \sqcap z$
assumes *bot*[*simp*]: $\perp \sqsubseteq x$
begin

lemma *mono-meet*: $x \sqsubseteq x' \implies y \sqsubseteq y' \implies x \sqcap y \sqsubseteq x' \sqcap y'$
{*proof*}

end

locale *Val-abs1-gamma* =
 Gamma **where** $\gamma = \gamma$ **for** $\gamma :: 'av :: L-top-bot \Rightarrow val\ set +$
assumes *inter-gamma-subset-gamma-meet*:
 $\gamma\ a1 \sqcap \gamma\ a2 \sqsubseteq \gamma(a1 \sqcap a2)$
and *gamma-Bot*[*simp*]: $\gamma\ \perp = \{\}$
begin

lemma *in-gamma-meet*: $x : \gamma\ a1 \implies x : \gamma\ a2 \implies x : \gamma(a1 \sqcap a2)$
{*proof*}

lemma *gamma-meet*[*simp*]: $\gamma(a1 \sqcap a2) = \gamma\ a1 \sqcap \gamma\ a2$
{*proof*}

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'*: $filter-plus'\ a\ a1\ a2 = (b1, b2) \implies$
 $n1 : \gamma\ a1 \implies n2 : \gamma\ a2 \implies n1+n2 : \gamma\ a \implies n1 : \gamma\ b1 \wedge n2 : \gamma\ b2$
and *filter-less'*: $filter-less'\ (n1 < n2)\ a1\ a2 = (b1, b2) \implies$
 $n1 : \gamma\ a1 \implies n2 : \gamma\ a2 \implies n1 : \gamma\ b1 \wedge n2 : \gamma\ b2$

locale *Abs-Int1* =
 Val-abs1 **where** $\gamma = \gamma$ **for** $\gamma :: 'av :: L-top-bot \Rightarrow val\ set$

begin

lemma *in-gamma-join-UpI*: $s : \gamma_o S1 \vee s : \gamma_o S2 \implies s : \gamma_o(S1 \sqcup S2)$
 ⟨proof⟩

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

lemma *aval''-sound*: $s : \gamma_o S \implies aval\ a\ s : \gamma(aval''\ a\ S)$
 ⟨proof⟩

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*: $s : \gamma_o S \implies aval\ e\ s : \gamma\ a \implies s : \gamma_o (afilter\ e\ a\ S)$
 ⟨proof⟩

lemma *bfilter-sound*: $s : \gamma_o S \implies bv = bval\ b\ s \implies s : \gamma_o (bfilter\ b\ bv\ S)$
 ⟨proof⟩

fun *step'* :: $'av\ st\ option \Rightarrow 'av\ st\ option\ acom \Rightarrow 'av\ st\ option\ acom$
where

$$\begin{aligned}
& \text{step}' S (\text{SKIP } \{P\}) = (\text{SKIP } \{S\}) \mid \\
& \text{step}' S (x ::= e \{P\}) = \\
& \quad x ::= e \{ \text{case } S \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } S \Rightarrow \text{Some}(\text{update } S \ x \ (\text{aval}' e \ S)) \} \mid \\
& \text{step}' S (c1;; c2) = \text{step}' S \ c1;; \text{step}' (\text{post } c1) \ c2 \mid \\
& \text{step}' S (\text{IF } b \ \text{THEN } c1 \ \text{ELSE } c2 \ \{P\}) = \\
& \quad (\text{let } c1' = \text{step}' (\text{bfilter } b \ \text{True } S) \ c1; \ c2' = \text{step}' (\text{bfilter } b \ \text{False } S) \ c2 \\
& \quad \text{in } \text{IF } b \ \text{THEN } c1' \ \text{ELSE } c2' \ \{\text{post } c1 \sqcup \text{post } c2\}) \mid \\
& \text{step}' S (\{\text{Inv}\} \ \text{WHILE } b \ \text{DO } c \ \{P\}) = \\
& \quad \{S \sqcup \text{post } c\} \\
& \quad \text{WHILE } b \ \text{DO } \text{step}' (\text{bfilter } b \ \text{True } \text{Inv}) \ c \\
& \quad \{\text{bfilter } b \ \text{False } \text{Inv}\}
\end{aligned}$$

definition $AI :: \text{com} \Rightarrow 'a \text{v st option acom option}$ **where**
 $AI = \text{lfp}_c (\text{step}' \top)$

lemma $\text{strip-step}'[\text{simp}]$: $\text{strip}(\text{step}' S \ c) = \text{strip } c$
 $\langle \text{proof} \rangle$

7.2 Soundness

lemma in-gamma-update :

$$\llbracket s : \gamma_f \ S; \ i : \gamma \ a \rrbracket \Longrightarrow s(x := i) : \gamma_f(\text{update } S \ x \ a)$$

$\langle \text{proof} \rangle$

lemma step-preserves-le :

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

$\langle \text{proof} \rangle$

lemma $AI\text{-sound}$: $AI \ c = \text{Some } c' \Longrightarrow CS \ c \leq \gamma_c \ c'$
 $\langle \text{proof} \rangle$

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 $\text{domo-Top}[\text{simp}]$: $\text{domo } \top = \{\}$
 $\langle \text{proof} \rangle$

lemma $\text{bot-acom-Com}[\text{simp}]$: $\perp_c \ c \in \text{Com } X$
 $\langle \text{proof} \rangle$

lemma post-in-annos : $\text{post } c : \text{set}(\text{annos } c)$
 $\langle \text{proof} \rangle$

lemma *domo-join*: $\text{domo } (S \sqcup T) \subseteq \text{domo } S \cup \text{domo } T$
(proof)

lemma *domo-afilter*: $\text{vars } a \subseteq X \implies \text{domo } S \subseteq X \implies \text{domo}(\text{afilter } a \ i \ S) \subseteq X$
(proof)

lemma *domo-bfilter*: $\text{vars } b \subseteq X \implies \text{domo } S \subseteq X \implies \text{domo}(\text{bfilter } b \ \text{bv } S) \subseteq X$
(proof)

lemma *step'-Com*:
 $\text{domo } S \subseteq X \implies \text{vars}(\text{strip } c) \subseteq X \implies c : \text{Com } X \implies \text{step}' S \ c : \text{Com } X$
(proof)

end

7.4 Monotonicity

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

lemma *mono-aval'*: $S \sqsubseteq S' \implies \text{aval}' e \ S \sqsubseteq \text{aval}' e \ S'$
(proof)

lemma *mono-aval''*: $S \sqsubseteq S' \implies \text{aval}'' e \ S \sqsubseteq \text{aval}'' e \ S'$
(proof)

lemma *mono-afilter*: $r \sqsubseteq r' \implies S \sqsubseteq S' \implies \text{afilter } e \ r \ S \sqsubseteq \text{afilter } e \ r' \ S'$
(proof)

lemma *mono-bfilter*: $S \sqsubseteq S' \implies \text{bfilter } b \ r \ S \sqsubseteq \text{bfilter } b \ r \ S'$
(proof)

lemma *mono-step'*: $S \sqsubseteq S' \implies c \sqsubseteq c' \implies \text{step}' S \ c \sqsubseteq \text{step}' S' \ c'$
(proof)

lemma *mono-step'2*: $\text{mono } (\text{step}' S)$
(proof)

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  $\gamma$ -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  $\langle$ proof $\rangle$ 
```

```
end
```

```
definition empty where empty =  $\{1..0\}$ 
```

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

```
lemma [simp]: is-empty(I  $l$   $h$ ) =
```

(*case l of Some l* \Rightarrow (*case h of Some h* \Rightarrow $h < l$ | *None* \Rightarrow *False*) | *None* \Rightarrow *False*)
 <proof>

lemma [*simp*]: *is-empty i* \Longrightarrow γ -*ivl i* = {}
 <proof>

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

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

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

fun *le-aux* **where**
le-aux (I l1 h1) (I l2 h2) = (*le-option False l2 l1* & *le-option True h1 h2*)

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

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

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

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

definition \top = {...}

instance
 <proof>

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 $\perp = \text{empty}$

instance

$\langle \text{proof} \rangle$

end

instantiation *option* :: (*minus*)*minus*

begin

fun *minus-option* **where**

Some *x* - *Some* *y* = *Some*(*x-y*) |

- - - = *None*

instance $\langle \text{proof} \rangle$

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*) \Rightarrow *I* (*l1-h2*) (*h1-l2*))

lemma *gamma-minus-ivl*:

$n1 : \gamma\text{-ivl } i1 \Rightarrow n2 : \gamma\text{-ivl } i2 \Rightarrow n1 - n2 : \gamma\text{-ivl}(\text{minus-ivl } i1 \ i2)$

$\langle \text{proof} \rangle$

definition *filter-plus-ivl* *i* *i1* *i2* = (~~*if is-empty* *i* | *is-empty* *i1* | *is-empty* *i2* then *empty* else~~
 $i1 \sqcap \text{minus-ivl } i \ i2, i2 \sqcap \text{minus-ivl } i \ i1$)

fun *filter-less-ivl* :: *bool* \Rightarrow *ivl* \Rightarrow *ivl* \Rightarrow *ivl* * *ivl* **where**

filter-less-ivl *res* (*I* *l1* *h1*) (*I* *l2* *h2*) =

(*if is-empty*(*I* *l1* *h1*) \vee *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 $\gamma = \gamma\text{-ivl}$ **and** *num'* = *num-ivl* **and** *plus'* = *plus-ivl*

$\langle \text{proof} \rangle$

global-interpretation *Val-abs1-gamma*

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

defines *aval-ivl* = *aval'*

$\langle \text{proof} \rangle$

lemma *mono-minus-ivl*:

$i1 \sqsubseteq i1' \Rightarrow i2 \sqsubseteq i2' \Rightarrow \text{minus-ivl } i1 \ i2 \sqsubseteq \text{minus-ivl } i1' \ i2'$

<proof>

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 $filter\text{-plus}' = filter\text{-plus}\text{-ivl}$ **and** $filter\text{-less}' = filter\text{-less}\text{-ivl}$

<proof>

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 $filter\text{-plus}' = filter\text{-plus}\text{-ivl}$ **and** $filter\text{-less}' = filter\text{-less}\text{-ivl}$

defines $afilter\text{-ivl} = afilter$

and $bfilter\text{-ivl} = bfilter$

and $step\text{-ivl} = step'$

and $AI\text{-ivl} = AI$

and $aval\text{-ivl}' = aval''$

<proof>

Monotonicity:

global-interpretation *Abs-Int1-mono*

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

and $test\text{-num}' = in\text{-ivl}$

and $filter\text{-plus}' = filter\text{-plus}\text{-ivl}$ **and** $filter\text{-less}' = filter\text{-less}\text{-ivl}$

<proof>

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) $\overset{\sim}{\sim}0$) (\perp_c *test2-ivl*))

value *show-acom* (((*step-ivl* \top) $\overset{\sim}{\sim}1$) (\perp_c *test2-ivl*))

value *show-acom* (((*step-ivl* \top) $\overset{\sim}{\sim}2$) (\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) $\overset{\sim}{\sim}0$) (\perp_c *test3-ivl*))

value *show-acom* (((*step-ivl* \top) $\overset{\sim}{\sim}1$) (\perp_c *test3-ivl*))

value *show-acom* (((*step-ivl* \top) $\overset{\sim}{\sim}2$) (\perp_c *test3-ivl*))

value *show-acom* (((*step-ivl* \top) $\overset{\sim}{\sim}3$) (\perp_c *test3-ivl*))

value *show-acom* (((*step-ivl* \top) $\overset{\sim}{\sim}4$) (\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  $\top$ )50) ( $\perp_c$  test4-ivl))
```

Relationships between variables are NOT captured:

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

Again, the analysis would not terminate:

```
value show-acom (((step-ivl  $\top$ )50) ( $\perp_c$  test6-ivl))
```

```
end
```

9 Widening and Narrowing

```
theory Abs-Int3
```

```
imports Abs-Int2-ivl
```

```
begin
```

```
class WN = SL-top +
```

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

```
assumes widen1:  $x \sqsubseteq x \nabla y$ 
```

```
assumes widen2:  $y \sqsubseteq x \nabla y$ 
```

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

```
assumes narrow1:  $y \sqsubseteq x \Longrightarrow y \sqsubseteq x \Delta y$ 
```

```
assumes narrow2:  $y \sqsubseteq x \Longrightarrow x \Delta y \sqsubseteq x$ 
```

9.1 Intervals

```
instantiation ivl :: WN
```

```
begin
```

```
definition widen-ivl ivl1 ivl2 =
```

```
(if is_empty ivl1 then ivl2 else // if is_empty ivl2 then ivl1 else //
case (ivl1, ivl2) of (I l1 h1, I l2 h2)  $\Rightarrow$ 
I (if le_option False l2 l1  $\wedge$  l2  $\neq$  l1 then None else l1)
(if le_option True h1 h2  $\wedge$  h1  $\neq$  h2 then None else h1))
```

```
definition narrow-ivl ivl1 ivl2 =
```

```
(if is_empty ivl1 // is_empty ivl2 then // empty else //
case (ivl1, ivl2) of (I l1 h1, I l2 h2)  $\Rightarrow$ 
I (if l1 = None then l2 else l1)
(if h1 = None then h2 else h1))
```

```
instance
```

```
<proof>
```

```
end
```

9.2 Abstract State

instantiation $st :: (WN)WN$
begin

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

definition $narrow-st\ F1\ F2 =$
 $FunDom\ (\lambda x. fun\ F1\ x\ \Delta\ fun\ F2\ x)\ (inter-list\ (dom\ F1)\ (dom\ F2))$

instance
 $\langle proof \rangle$

end

9.3 Option

instantiation $option :: (WN)WN$
begin

fun $widen-option\ where$
 $None\ \nabla\ x = x\ |$
 $x\ \nabla\ None = x\ |$
 $(Some\ x)\ \nabla\ (Some\ y) = Some(x\ \nabla\ y)$

fun $narrow-option\ where$
 $None\ \Delta\ x = None\ |$
 $x\ \Delta\ None = None\ |$
 $(Some\ x)\ \Delta\ (Some\ y) = Some(x\ \Delta\ y)$

instance
 $\langle proof \rangle$

end

9.4 Annotated commands

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

abbreviation $widen-acom :: ('a::WN)acom \Rightarrow 'a\ acom \Rightarrow 'a\ acom\ (infix\ \nabla_c\ 65)$
where $widen-acom == map2-acom\ (\nabla)$

abbreviation $\text{*narrow-acom* :: ('a::WN)acom \Rightarrow 'a\ acom \Rightarrow 'a\ acom}$ (**infix** Δ_c 65)

where $\text{*narrow-acom* == map2-acom (\Delta)}$

lemma $\text{*widen1-acom*: strip\ c = strip\ c' \Longrightarrow c \sqsubseteq c \nabla_c c'}$
<proof>

lemma $\text{*widen2-acom*: strip\ c = strip\ c' \Longrightarrow c' \sqsubseteq c \nabla_c c'}$
<proof>

lemma $\text{*narrow1-acom*: y \sqsubseteq x \Longrightarrow y \sqsubseteq x \Delta_c y}$
<proof>

lemma $\text{*narrow2-acom*: y \sqsubseteq x \Longrightarrow x \Delta_c y \sqsubseteq x}$
<proof>

9.5 Post-fixed point computation

definition $\text{*iter-widen* :: ('a\ acom \Rightarrow 'a\ acom) \Rightarrow 'a\ acom \Rightarrow ('a::WN)acom\ option}$
where $\text{*iter-widen*\ f = while-option (\lambda c. \neg f\ c \sqsubseteq c) (\lambda c. c \nabla_c f\ c)}$

definition $\text{*iter-narrow* :: ('a\ acom \Rightarrow 'a\ acom) \Rightarrow 'a\ acom \Rightarrow 'a::WN\ acom\ option}$
where $\text{*iter-narrow*\ f = while-option (\lambda c. \neg c \sqsubseteq c \Delta_c f\ c) (\lambda c. c \Delta_c f\ c)}$

definition *pfp-wn* ::

$\text{(('a::WN)option\ acom \Rightarrow 'a\ option\ acom) \Rightarrow com \Rightarrow 'a\ option\ acom\ option}$
where $\text{*pfp-wn*\ f\ c = (case\ iter-widen\ f\ (\perp_c\ c)\ of\ None \Rightarrow None$
 $\quad | \text{Some\ } c' \Rightarrow \text{*iter-narrow*\ f\ } c')$

lemma $\text{*strip-map2-acom*:$

$\text{strip\ } c1 = \text{strip\ } c2 \Longrightarrow \text{strip}(\text{map2-acom\ } f\ c1\ c2) = \text{strip\ } c1$
<proof>

lemma $\text{*iter-widen-pfp*: iter-widen\ f\ c = Some\ } c' \Longrightarrow f\ c' \sqsubseteq c'$
<proof>

lemma $\text{*strip-while*: fixes\ } f :: 'a\ acom \Rightarrow 'a\ acom$

assumes $\forall c. \text{strip}\ (f\ c) = \text{strip}\ c$ **and** $\text{while-option}\ P\ f\ c = \text{Some}\ c'$

shows $\text{strip}\ c' = \text{strip}\ c$

<proof>

lemma $\text{*strip-iter-widen*: fixes\ } f :: 'a::WN\ acom \Rightarrow 'a\ acom$

assumes $\forall c. \text{strip}\ (f\ c) = \text{strip}\ c$ **and** $\text{iter-widen}\ f\ c = \text{Some}\ c'$

shows $\text{strip}\ c' = \text{strip}\ c$

<proof>

lemma $\text{*iter-narrow-pfp*: assumes\ mono\ } f \text{ and } f\ c0 \sqsubseteq c0$

and $\text{iter-narrow}\ f\ c0 = \text{Some}\ c$

shows $f c \sqsubseteq c \wedge c \sqsubseteq c0$ (**is** ? P c)
 $\langle proof \rangle$

lemma *pfp-wn-pfp*:
 $\llbracket mono\ f; \ pfp\text{-}wn\ f\ c = Some\ c' \rrbracket \implies f\ c' \sqsubseteq c'$
 $\langle proof \rangle$

lemma *strip-pfp-wn*:
 $\llbracket \forall c. \ strip(f\ c) = strip\ c; \ pfp\text{-}wn\ f\ c = Some\ c' \rrbracket \implies strip\ c' = c$
 $\langle proof \rangle$

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

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

lemma *AI-wn-sound*: $AI\text{-}wn\ c = Some\ c' \implies CS\ c \leq \gamma_c\ c'$
 $\langle proof \rangle$

end

global-interpretation *Abs-Int2*
where $\gamma = \gamma\text{-}ivl$ **and** $num' = num\text{-}ivl$ **and** $plus' = plus\text{-}ivl$
and $test\text{-}num' = in\text{-}ivl$
and $filter\text{-}plus' = filter\text{-}plus\text{-}ivl$ **and** $filter\text{-}less' = filter\text{-}less\text{-}ivl$
defines $AI\text{-}ivl' = AI\text{-}wn$
 $\langle proof \rangle$

9.6 Tests

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

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

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 $show\text{-}acom\ (step\text{-}up\text{-}ivl\ 1\ (\perp_c\ test3\text{-}ivl))$
value $show\text{-}acom\ (step\text{-}up\text{-}ivl\ 2\ (\perp_c\ test3\text{-}ivl))$
value $show\text{-}acom\ (step\text{-}up\text{-}ivl\ 3\ (\perp_c\ test3\text{-}ivl))$
value $show\text{-}acom\ (step\text{-}up\text{-}ivl\ 4\ (\perp_c\ test3\text{-}ivl))$
value $show\text{-}acom\ (step\text{-}up\text{-}ivl\ 5\ (\perp_c\ test3\text{-}ivl))$
value $show\text{-}acom\ (step\text{-}down\text{-}ivl\ 1\ (step\text{-}up\text{-}ivl\ 5\ (\perp_c\ test3\text{-}ivl)))$
value $show\text{-}acom\ (step\text{-}down\text{-}ivl\ 2\ (step\text{-}up\text{-}ivl\ 5\ (\perp_c\ test3\text{-}ivl)))$
value $show\text{-}acom\ (step\text{-}down\text{-}ivl\ 3\ (step\text{-}up\text{-}ivl\ 5\ (\perp_c\ test3\text{-}ivl)))$

Now all the analyses terminate:

value $show\text{-}acom\text{-}opt\ (AI\text{-}ivl'\ test4\text{-}ivl)$
value $show\text{-}acom\text{-}opt\ (AI\text{-}ivl'\ test5\text{-}ivl)$
value $show\text{-}acom\text{-}opt\ (AI\text{-}ivl'\ test6\text{-}ivl)$

9.7 Termination: Intervals

definition $m\text{-}ivl :: ivl \Rightarrow nat$ **where**

$m\text{-}ivl\ ivl = (case\ ivl\ of\ I\ l\ h \Rightarrow$
 $(case\ l\ of\ None \Rightarrow 0 \mid Some\ - \Rightarrow 1) + (case\ h\ of\ None \Rightarrow 0 \mid Some\ - \Rightarrow 1))$

lemma $m\text{-}ivl\text{-}height: m\text{-}ivl\ ivl \leq 2$

$\langle proof \rangle$

lemma $m\text{-}ivl\text{-}anti\text{-}mono: (y::ivl) \sqsubseteq x \Longrightarrow m\text{-}ivl\ x \leq m\text{-}ivl\ y$

$\langle proof \rangle$

lemma $m\text{-}ivl\text{-}widen:$

$\sim y \sqsubseteq x \Longrightarrow m\text{-}ivl(x \nabla y) < m\text{-}ivl\ x$

$\langle proof \rangle$

lemma $Top\text{-}less\text{-}ivl: \top \sqsubseteq x \Longrightarrow m\text{-}ivl\ x = 0$

$\langle proof \rangle$

definition $n\text{-}ivl :: ivl \Rightarrow nat$ **where**

$n\text{-}ivl\ ivl = 2 - m\text{-}ivl\ ivl$

lemma $n\text{-}ivl\text{-}mono: (x::ivl) \sqsubseteq y \Longrightarrow n\text{-}ivl\ x \leq n\text{-}ivl\ y$

$\langle proof \rangle$

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

$\sim x \sqsubseteq x \triangle y \Longrightarrow n\text{-}ivl(x \triangle y) < n\text{-}ivl\ x$

$\langle proof \rangle$

9.8 Termination: Abstract State

definition $m\text{-}st\ m\ st = (\sum x \in set(dom\ st). m(fun\ st\ x))$

lemma $m\text{-}st\text{-}height: \text{assumes } finite\ X \text{ and } set\ (dom\ S) \subseteq X$

shows $m\text{-}st\ m\text{-}ivl\ S \leq 2 * card\ X$

$\langle proof \rangle$

lemma $m\text{-}st\text{-}anti\text{-}mono:$

$S1 \sqsubseteq S2 \Longrightarrow m\text{-}st\ m\text{-}ivl\ S2 \leq m\text{-}st\ m\text{-}ivl\ S1$

$\langle proof \rangle$

lemma $m\text{-}st\text{-}widen:$

assumes $\neg S2 \sqsubseteq S1$ **shows** $m\text{-}st\ m\text{-}ivl\ (S1 \nabla S2) < m\text{-}st\ m\text{-}ivl\ S1$

$\langle proof \rangle$

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

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

shows $n\text{-}st\ n\text{-}ivl\ X\ S1 \leq n\text{-}st\ n\text{-}ivl\ X\ S2$

<proof>

lemma *n-st-narrow*:

assumes *finite X* **and** $set(dom\ S1) \subseteq X$ $set(dom\ S2) \subseteq X$

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>

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-o-anti-mono*: $finite\ X \Longrightarrow domo\ S2 \subseteq X \Longrightarrow S1 \sqsubseteq S2 \Longrightarrow$

$m\text{-o}\ (m\text{-st}\ m\text{-ivl})\ (2 * card\ X)\ S2 \leq m\text{-o}\ (m\text{-st}\ m\text{-ivl})\ (2 * card\ X)\ S1$

<proof>

lemma *m-o-widen*: $\llbracket finite\ X; domo\ S2 \subseteq X; \neg S2 \sqsubseteq S1 \rrbracket \Longrightarrow$

$m\text{-o}\ (m\text{-st}\ m\text{-ivl})\ (2 * card\ X)\ (S1 \nabla S2) < m\text{-o}\ (m\text{-st}\ m\text{-ivl})\ (2 * card\ X)\ S1$

<proof>

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

lemma *n-o-mono*: $domo\ S1 \subseteq X \Longrightarrow domo\ S2 \subseteq X \Longrightarrow S1 \sqsubseteq S2 \Longrightarrow$

$n\text{-o}\ (n\text{-st}\ n\text{-ivl}\ X)\ S1 \leq n\text{-o}\ (n\text{-st}\ n\text{-ivl}\ X)\ S2$

<proof>

lemma *n-o-narrow*:

$\llbracket finite\ X; domo\ S1 \subseteq X; domo\ S2 \subseteq X; S2 \sqsubseteq S1; \neg S1 \sqsubseteq S1 \triangle S2 \rrbracket$

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

<proof>

lemma *domo-widen-subset*: $domo\ (S1 \nabla S2) \subseteq domo\ S1 \cup domo\ S2$

<proof>

lemma *domo-narrow-subset*: $domo\ (S1 \triangle S2) \subseteq domo\ S1 \cup domo\ S2$

<proof>

9.10 Termination: Commands

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

$strip\ c' = strip\ (c::'a::WN\ acom) \Longrightarrow strip\ (c \nabla_c\ c') = strip\ c$

<proof>

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

$strip\ c' = strip\ (c::'a::WN\ acom) \Longrightarrow strip\ (c \triangle_c\ c') = strip\ c$

<proof>

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

$annos(c1 \nabla_c\ c2) = map\ (\% (x,y).x \nabla y)\ (zip\ (annos\ c1)\ (annos(c2::'a::WN\ acom)))$

<proof>

lemma *annos-narrow-acom*[simp]: $\text{strip } c1 = \text{strip } (c2::'a::WN \text{acom}) \implies$
 $\text{annos}(c1 \Delta_c c2) = \text{map } (\% (x,y).x\Delta y) (\text{zip } (\text{annos } c1) (\text{annos}(c2::'a::WN \text{acom})))$
 ⟨proof⟩

lemma *widen-acom-Com*[simp]: $\text{strip } c2 = \text{strip } c1 \implies$
 $c1 : \text{Com } X \implies c2 : \text{Com } X \implies (c1 \nabla_c c2) : \text{Com } X$
 ⟨proof⟩

lemma *narrow-acom-Com*[simp]: $\text{strip } c2 = \text{strip } c1 \implies$
 $c1 : \text{Com } X \implies c2 : \text{Com } X \implies (c1 \Delta_c c2) : \text{Com } X$
 ⟨proof⟩

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

lemma *measure-m-c*: $\text{finite } X \implies \{(c, c \nabla_c c') \mid c c'::\text{ivl } \text{st } \text{option } \text{acom}.$
 $\text{strip } c' = \text{strip } c \wedge c : \text{Com } X \wedge c' : \text{Com } X \wedge \neg c' \sqsubseteq c\}^{-1}$
 $\subseteq \text{measure}(m\text{-c}(m\text{-o } (m\text{-st } m\text{-ivl } (2*\text{card}(X)))))$
 ⟨proof⟩

lemma *measure-n-c*: $\text{finite } X \implies \{(c, c \Delta_c c') \mid c c'.$
 $\text{strip } c = \text{strip } c' \wedge c \in \text{Com } X \wedge c' \in \text{Com } X \wedge c' \sqsubseteq c \wedge \neg c \sqsubseteq c \Delta_c c'\}^{-1}$
 $\subseteq \text{measure}(m\text{-c}(n\text{-o } (n\text{-st } n\text{-ivl } X)))$
 ⟨proof⟩

9.11 Termination: Post-Fixed Point Iterations

lemma *iter-widen-termination*:
fixes $c0 :: 'a::WN \text{acom}$
assumes $P\text{-f}: \bigwedge c. P c \implies P(f c)$
assumes $P\text{-widen}: \bigwedge c c'. P c \implies P c' \implies P(c \nabla_c c')$
and $wf(\{(c::'a \text{acom}, c \nabla_c c') \mid c c'. P c \wedge P c' \wedge \sim c' \sqsubseteq c\}^{-1})$
and $P c0$ **and** $c0 \sqsubseteq f c0$ **shows** $\exists c. \text{iter-widen } f c0 = \text{Some } c$
 ⟨proof⟩

lemma *iter-narrow-termination*:
assumes $P\text{-f}: \bigwedge c. P c \implies P(c \Delta_c f c)$
and $wf(\{(c, c \Delta_c f c) \mid c c'. P c \wedge \sim c \sqsubseteq c \Delta_c f c\}^{-1})$
and $P c0$ **shows** $\exists c. \text{iter-narrow } f c0 = \text{Some } c$
 ⟨proof⟩

lemma *iter-widen-step-ivl-termination*:
 $\exists c. \text{iter-widen } (\text{step-ivl } \top) (\perp_c c0) = \text{Some } c$
 ⟨proof⟩

lemma *iter-narrow-step-ivl-termination*:
 $c0 \in \text{Com } (\text{vars}(\text{strip } c0)) \implies \text{step-ivl } \top c0 \sqsubseteq c0 \implies$
 $\exists c. \text{iter-narrow } (\text{step-ivl } \top) c0 = \text{Some } c$
 ⟨proof⟩

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** $c' : \text{Com}(X)$

$\langle \text{proof} \rangle$

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** $c' : \text{Com}(X)$

$\langle \text{proof} \rangle$

context *Abs-Int2*

begin

lemma *iter-widen-step'-Com*:

iter-widen $(\text{step}' \top) c = \text{Some } c' \Longrightarrow \text{vars}(\text{strip } c) \subseteq X \Longrightarrow c : \text{Com}(X)$

$\Longrightarrow c' : \text{Com}(X)$

$\langle \text{proof} \rangle$

end

theorem *AI-ivl'-termination*:

$\exists c'. \text{AI-ivl}' c = \text{Some } c'$

$\langle \text{proof} \rangle$

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>.