Abstract Interpretation of Annotated Commands

Tobias Nipkow

March 17, 2025

Abstract

This is the Isabelle formalization of the material decribed 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 leats 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*: $A \subseteq L\ i \Longrightarrow a \in A \Longrightarrow (Glb\ i\ A) \leq a$
and *Glb-greatest*: $b : L\ i \Longrightarrow \forall\, a{\in}A.\ b \leq a \Longrightarrow b \leq (Glb\ i\ A)$
and *Glb-in-L*: $A \subseteq L\ i \Longrightarrow Glb\ i\ A : L\ i$
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*: $lfp\ f\ i : L\ i$
by(*auto simp*: *lfp-def intro*: *Glb-in-L*)

lemma *lfp-lowerbound*:
$[\![\ a : L\ i;\ \ f\ a \leq a\ ]\!] \Longrightarrow lfp\ f\ i \leq a$
by (*auto simp add*: *lfp-def intro*: *Glb-lower*)

lemma *lfp-greatest*:
$[\![\ a : L\ i;\ \ \bigwedge u.\ [\![\ u : L\ i; f\ u \leq u]\!] \Longrightarrow a \leq u\ ]\!] \Longrightarrow a \leq lfp\ f\ i$
by (*auto simp add*: *lfp-def intro*: *Glb-greatest*)

lemma *lfp-unfold*: **assumes** $\bigwedge x\ i.\ f\ x : L\ i \longleftrightarrow x : L\ i$
and *mono*: *mono f* **shows** $lfp\ f\ i = f\ (lfp\ f\ i)$
proof−
note *assms*(*1*)[*simp*] *index-lfp*[*simp*]
have *1*: $f\ (lfp\ f\ i) \leq lfp\ f\ i$
apply(*rule lfp-greatest*)
apply *simp*
by (*blast intro*: *lfp-lowerbound monoD*[*OF mono*] *order-trans*)
have $lfp\ f\ i \leq f\ (lfp\ f\ i)$
by (*fastforce intro*: *1 monoD*[*OF mono*] *lfp-lowerbound*)
with *1* **show** *?thesis* **by**(*blast intro*: *order-antisym*)
qed

end

end

2 Annotated Commands

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

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

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

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

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

fun *anno* :: *'a* ⇒ *com* ⇒ *'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* ⇒ *'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* ⇒ *'b*) ⇒ *'a acom* ⇒ *'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*)
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'*} ⟷ (∃ *S*. *c* = *SKIP* {*S*} ∧ *S'* = *f S*)
by (*cases c*) *auto*

lemma *map-acom-Assign*:

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

lemma *map-acom-If*:
map-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-acom f c1 = c1'* $\wedge$ *map-acom f c2 = c2'* $\wedge\ S' = f\ S)$
by (*cases c*) *auto*

lemma *map-acom-While*:
map-acom f w = {I'} WHILE b DO c' {P'} $\longleftrightarrow$
$(\exists I\ P\ c.\ w =$ *{I} WHILE b DO c* $\{P\} \wedge$ *map-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* $\Longrightarrow$ *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*}) $\le$ (*SKIP* {*S'*}) = (*S* $\le$ *S'*) |
(*x* ::= *e* {*S*}) $\le$ (*x'* ::= *e'* {*S'*}) = (*x*=*x'* $\wedge$ *e*=*e'* $\wedge$ *S* $\le$ *S'*) |
(*c1*;;*c2*) $\le$ (*c1'*;;*c2'*) = (*c1* $\le$ *c1'* $\wedge$ *c2* $\le$ *c2'*) |
(*IF b THEN c1 ELSE c2* {*S*}) $\le$ (*IF b' THEN c1' ELSE c2'* {*S'*}) =
  (*b*=*b'* $\wedge$ *c1* $\le$ *c1'* $\wedge$ *c2* $\le$ *c2'* $\wedge$ *S* $\le$ *S'*) |
({*Inv*} *WHILE b DO c* {*P*}) $\le$ ({*Inv'*} *WHILE b' DO c'* {*P'*}) =
  (*b*=*b'* $\wedge$ *c* $\le$ *c'* $\wedge$ *Inv* $\le$ *Inv'* $\wedge$ *P* $\le$ *P'*) |
less-eq-acom - - = *False*

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

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

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

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

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

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

definition less-acom :: 'a acom ⇒ 'a acom ⇒ bool where
less-acom x y = (x ≤ y ∧ ¬ y ≤ 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 sub₁ :: 'a acom ⇒ 'a acom where
sub₁(c1;;c2) = c1 |
sub₁(IF b THEN c1 ELSE c2 {S}) = c1 |
sub₁({I} WHILE b DO c {P}) = c

fun sub₂ :: 'a acom ⇒ 'a acom where
sub₂(c1;;c2) = c2 |
sub₂(IF b THEN c1 ELSE c2 {S}) = c2

fun invar :: 'a acom ⇒ 'a where
invar({I} WHILE b DO c {P}) = I

fun lift :: ('a set ⇒ 'b) ⇒ com ⇒ 'a acom set ⇒ '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 (sub₁ ‘ M);; lift F c2 (sub₂ ‘ M) |
lift F (IF b THEN c1 ELSE c2) M =
  IF b THEN lift F c1 (sub₁ ‘ M) ELSE lift F c2 (sub₂ ‘ M)
  {F (post ‘ M)} |
lift F (WHILE b DO c) M =
 {F (invar ‘ M)}
 WHILE b DO lift F c (sub₁ ‘ 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*:$A \Longrightarrow$ *lift Inter* (*strip a*) $A \leq a$
  proof(*induction a arbitrary*: *A*)
    case *Seq* **from** *Seq.prems* **show** *?case* **by**(*force intro*!: *Seq.IH*)
  next
    case *If* **from** *If.prems* **show** *?case* **by**(*force intro*!: *If.IH*)
  next
    case *While* **from** *While.prems* **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.prems* **show** *?case*
      by (*force intro*!: *Seq.IH simp*:*Seq-le*)
  next
    case *If* **from** *If.prems* **show** *?case* **by** (*force simp*: *If-le intro*!: *If.IH*)
  next
    case *While* **from** *While.prems* **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.prems* **show** *?case*
      by (*fastforce simp*: *strip-eq-Seq subset-iff intro*!: *Seq.IH*)
  next
    case *If* **from** *If.prems* **show** *?case*
      by (*fastforce intro*!: *If.IH simp*: *strip-eq-If*)
  next
    case *While* **from** *While.prems* **show** *?case*
      by(*fastforce intro*: *While.IH simp*: *strip-eq-While*)
  qed *auto*
  thus *?case* **by** *auto*
qed

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

3.2 Collecting semantics

fun *step* :: *state set* ⇒ *state set acom* ⇒ *state set acom* **where**
step S (*SKIP* {*P*}) = (*SKIP* {*S*}) |
step S (*x* ::= *e* {*P*}) =
  (*x* ::= *e* {{*s'*. ∃ *s*∈*S*. *s'* = *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*:*S*. *bval b s*} *c1 ELSE step* {*s*:*S*. ¬ *bval b s*} *c2*
   {*post c1* ∪ *post c2*} |
step S ({*Inv*} *WHILE b DO c* {*P*}) =
  {*S* ∪ *post c*} *WHILE b DO* (*step* {*s*:*Inv*. *bval b s*} *c*) {{*s*:*Inv*. ¬ *bval b s*}}

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

lemma *mono2-step*: *c1* ≤ *c2* ⟹ *S1* ⊆ *S2* ⟹ *step S1 c1* ≤ *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** *mono f* = (∀ *x y*. *x* ⊑ *y* ⟶ *f x* ⊑ *f y*)

lemma *monoD*: *mono f* ⟹ *x* ⊑ *y* ⟹ *f x* ⊑ *f y* **by**(*simp add*: *mono-def*)

lemma *mono-comp*: *mono f* ⟹ *mono g* ⟹ *mono* (*g o 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 $x \sqsubseteq y$ 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** ‹⊔› *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* ⊔ *y* ⊑ *z* ⟷ *x* ⊑ *z* ∧ *y* ⊑ *z*
by (*metis join-ge1 join-ge2 join-least le-trans*)

lemma *le-join-disj*: *x* ⊑ *y* ∨ *x* ⊑ *z* ⟹ *x* ⊑ *y* ⊔ *z*
by (*metis join-ge1 join-ge2 le-trans*)

end

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

definition *f* ⊑ *g* = (∀ *x*. *f x* ⊑ *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* ⇒ '*a acom* ⇒ *bool* **where**
le-acom (*SKIP* {*S*}) (*SKIP* {*S'*}) = (*S* ⊑ *S'*) |
le-acom (*x* ::= *e* {*S*}) (*x'* ::= *e'* {*S'*}) = (*x*=*x'* ∧ *e*=*e'* ∧ *S* ⊑ *S'*) |
le-acom (*c1*;;*c2*) (*c1'*;;*c2'*) = (*le-acom c1 c1'* ∧ *le-acom c2 c2'*) |
le-acom (*IF b THEN c1 ELSE c2* {*S*}) (*IF b' THEN c1' ELSE c2'* {*S'*}) =
  (*b*=*b'* ∧ *le-acom c1 c1'* ∧ *le-acom c2 c2'* ∧ *S* ⊑ *S'*) |
le-acom ({*Inv*} *WHILE b DO c* {*P*}) ({*Inv'*} *WHILE b' DO c'* {*P'*}) =
  (*b*=*b'* ∧ *le-acom c c'* ∧ *Inv* ⊑ *Inv'* ∧ *P* ⊑ *P'*) |
le-acom - - = *False*

lemma [*simp*]: *SKIP* {*S*} ⊑ *c* ⟷ (∃ *S'*. *c* = *SKIP* {*S'*} ∧ *S* ⊑ *S'*)
by (*cases c*) *auto*

lemma [*simp*]: *x* ::= *e* {*S*} ⊑ *c* ⟷ (∃ *S'*. *c* = *x* ::= *e* {*S'*} ∧ *S* ⊑ *S'*)
by (*cases c*) *auto*

lemma [*simp*]: *c1*;;*c2* ⊑ *c* ⟷ (∃ *c1' c2'*. *c* = *c1'*;;*c2'* ∧ *c1* ⊑ *c1'* ∧ *c2* ⊑ *c2'*)
by (*cases c*) *auto*

lemma [*simp*]: *IF b THEN c1 ELSE c2* {*S*} ⊑ *c* ⟷
  (∃ *c1' c2' S'*. *c* = *IF b THEN c1' ELSE c2'* {*S'*} ∧ *c1* ⊑ *c1'* ∧ *c2* ⊑ *c2'* ∧ *S* ⊑ *S'*)
by (*cases c*) *auto*

lemma [*simp*]: {*Inv*} *WHILE b DO c* {*P*} ⊑ *w* ⟷
  (∃ *Inv' c' P'*. *w* = {*Inv'*} *WHILE b DO c'* {*P'*} ∧ *c* ⊑ *c'* ∧ *Inv* ⊑ *Inv'* ∧ *P* ⊑ *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 ⊑ *Some y* = (*x* ⊑ *y*) |
None ⊑ *y* = *True* |
Some - ⊑ *None* = *False*

lemma [*simp*]: (*x* ⊑ *None*) = (*x* = *None*)
by (*cases x*) *simp-all*

lemma [*simp*]: (*Some x* ⊑ *u*) = (∃ *y*. *u* = *Some y* & *x* ⊑ *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 ⊔ *Some y* = *Some*(*x* ⊔ *y*) |
None ⊔ *y* = *y* |
x ⊔ *None* = *x*

lemma *join-None2*[*simp*]: *x* ⊔ *None* = *x*
by (*cases x*) *simp-all*

definition ⊤ = *Some* ⊤

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* ⇒ (*'a*::*SL-top*)*option acom* (‹⊥$_c$›) **where**
⊥$_c$ = *anno None*

lemma *strip-bot-acom*[*simp*]: *strip*(⊥$_c$ *c*) = *c*
by(*simp add*: *bot-acom-def*)

lemma *bot-acom*[*rule-format*]: *strip c'* = *c* ⟶ ⊥$_c$ *c* ⊑ *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
  pfp :: ((*'a*::*preord*) ⇒ *'a*) ⇒ *'a* ⇒ *'a option* **where**
pfp f = *while-option* (λ*x*. ¬ *f x* ⊑ *x*) *f*

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

lemma *pfp-least*:
assumes *mono*: ⋀*x y*. *x* ⊑ *y* ⟹ *f x* ⊑ *f y*
and *f p* ⊑ *p* **and** *x0* ⊑ *p* **and** *pfp f x0* = *Some x* **shows** *x* ⊑ *p*
proof−
  { fix *x* **assume** *x* ⊑ *p*
    hence *f x* ⊑ *f p* **by**(*rule mono*)
    from *this* ‹*f p* ⊑ *p*› **have** *f x* ⊑ *p* **by**(*rule le-trans*)

```
  }
  thus x ⊑ p using assms(2−) while-option-rule[where P = %x. x ⊑ p]
    unfolding pfp-def by blast
qed

definition
 lpfp_c :: (('a::SL-top)option acom ⇒ 'a option acom) ⇒ com ⇒ 'a option acom
option where
lpfp_c f c = pfp f (⊥_c c)

lemma lpfpc-pfp: lpfp_c f c0 = Some c ⟹ f c ⊑ c
by(simp add: pfp-pfp lpfp_c-def)

lemma strip-pfp:
assumes ⋀x. g(f x) = g x and pfp 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 pfp-def by metis

lemma strip-lpfpc: assumes ⋀c. strip(f c) = strip c and lpfp_c f c = Some c'
shows strip c' = c
using assms(1) strip-pfp[OF - assms(2)[simplified lpfp_c-def]]
by(metis strip-bot-acom)

lemma lpfpc-least:
assumes mono: ⋀x y. x ⊑ y ⟹ f x ⊑ f y
and strip p = c0 and f p ⊑ p and lp: lpfp_c f c0 = Some c shows c ⊑ p
using pfp-least[OF - - bot-acom[OF ‹strip p = c0›] lp[simplified lpfp_c-def]]
  mono ‹f p ⊑ p›
by blast
```

4.2 Abstract Interpretation

```
definition γ-fun :: ('a ⇒ 'b set) ⇒ ('c ⇒ 'a) ⇒ ('c ⇒ 'b)set where
γ-fun γ F = {f. ∀ x. f x ∈ γ(F x)}

fun γ-option :: ('a ⇒ 'b set) ⇒ 'a option ⇒ '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 ⇒ val set
  assumes mono-gamma: a ⊑ b ⟹ γ a ⊆ γ b
  and gamma-Top[simp]: γ ⊤ = UNIV
fixes num' :: val ⇒ 'av
and plus' :: 'av ⇒ 'av ⇒ 'av
  assumes gamma-num': n : γ(num' n)
  and gamma-plus':
 n1 : γ a1 ⟹ n2 : γ a2 ⟹ n1+n2 : γ(plus' a1 a2)
```

type-synonym *'av st* = (*vname* ⇒ *'av*)

locale *Abs-Int-Fun* = *Val-abs* γ **for** γ :: *'av*::*SL-top* ⇒ *val set*
begin

fun *aval'* :: *aexp* ⇒ *'av st* ⇒ *'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* ⇒ *'av st option acom* ⇒ *'av st option acom*
where
step' S (*SKIP* {*P*}) = (*SKIP* {*S*}) |
step' S (*x* ::= *e* {*P*}) =
x ::= *e* {*case S of None* ⇒ *None* | *Some S* ⇒ *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* ⊔ *post c2*} |
step' S ({*Inv*} *WHILE b DO c* {*P*}) =
{*S* ⊔ *post c*} *WHILE b DO* (*step' Inv c*) {*Inv*}

definition *AI* :: *com* ⇒ *'av st option acom option* **where**
AI = $lpfp_c$ (*step'* ⊤)

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

abbreviation γ_f :: *'av st* ⇒ *state set*
where γ_f == γ-*fun* γ

abbreviation γ_o :: *'av st option* ⇒ *state set*
where γ_o == γ-*option* γ_f

abbreviation γ_c :: *'av st option acom* ⇒ *state set acom*
where γ_c == *map-acom* γ_o

lemma *gamma-f-Top*[*simp*]: γ_f *Top* = *UNIV*
by(*simp add*: *Top-fun-def* γ-*fun-def*)

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

lemma *mono-gamma-f*: *f* ⊑ *g* ⟹ γ_f *f* ⊆ γ_f *g*
by(*auto simp*: *le-fun-def* γ-*fun-def dest*: *mono-gamma*)

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

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

Soundness:

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

lemma *in-gamma-update*:
$[\![\ s : \gamma_f\ S;\ i : \gamma\ a\ ]\!] \Longrightarrow s(x := i) : \gamma_f(S(x := a))$
by(*simp add*: γ*-fun-def*)

lemma *step-preserves-le*:
$[\![\ S \subseteq \gamma_o\ S';\ c \leq \gamma_c\ c'\ ]\!] \Longrightarrow step\ S\ c \leq \gamma_c\ (step'\ S'\ c')$
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**
$c' = IF\ b\ THEN\ c1'\ ELSE\ c2'\ \{P'\}$
$P \subseteq \gamma_o\ P'\ c1 \leq \gamma_c\ c1'\ c2 \leq \gamma_c\ c2'$
by (*fastforce simp*: *If-le map-acom-If*)
moreover have $post\ c1 \subseteq \gamma_o(post\ c1' \sqcup post\ c2')$
by (*metis* (*no-types*) ‹*c1* $\leq \gamma_c$ *c1'*› *join-ge1 le-post mono-gamma-o order-trans post-map-acom*)
moreover have $post\ c2 \subseteq \gamma_o(post\ c1' \sqcup post\ c2')$
by (*metis* (*no-types*) ‹*c2* $\leq \gamma_c$ *c2'*› *join-ge2 le-post mono-gamma-o order-trans post-map-acom*)
ultimately show *?case* **using** ‹$S \subseteq \gamma_o\ S'$› **by** (*simp add*: *If.IH subset-iff*)
next
case (*While I b c1 P*)
then obtain *c1' I' P'* **where**
$c' = \{I'\}\ WHILE\ b\ DO\ c1'\ \{P'\}$
$I \subseteq \gamma_o\ I'\ P \subseteq \gamma_o\ P'\ c1 \leq \gamma_c\ c1'$
by (*fastforce simp*: *map-acom-While While-le*)
moreover have $S \cup post\ c1 \subseteq \gamma_o\ (S' \sqcup post\ c1')$
using ‹$S \subseteq \gamma_o\ S'$› *le-post*[*OF* ‹*c1* $\leq \gamma_c$ *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'* $\Longrightarrow$ *CS c* $\le \gamma_c$ *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 lpfpc-pfp*[*OF 1*])
have *3*: *strip* (γ_c (*step'* $\top$ *c'*)) = *c*
by(*simp add*: *strip-lpfpc*[*OF - 1*])
have *lfp* (*step UNIV*) *c* $\le \gamma_c$ (*step'* $\top$ *c'*)
proof(*rule lfp-lowerbound*[*simplified*,*OF 3*])
show *step UNIV* (γ_c (*step'* $\top$ *c'*)) $\le \gamma_c$ (*step'* $\top$ *c'*)
proof(*rule step-preserves-le*[*OF - -*])
show *UNIV* $\subseteq \gamma_o$ $\top$ **by** *simp*
show γ_c (*step'* $\top$ *c'*) $\le \gamma_c$ *c'* **by**(*rule mono-gamma-c*[*OF 2*])
qed
qed
with *2* **show** *lfp* (*step UNIV*) *c* $\le \gamma_c$ *c'*
by (*blast intro*: *mono-gamma-c order-trans*)
qed

end

4.2.1 Monotonicity

lemma *mono-post*: *c* $\sqsubseteq$ *c'* $\Longrightarrow$ *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* $\Longrightarrow$ *a2* $\sqsubseteq$ *b2* $\Longrightarrow$ *plus' a1 a2* $\sqsubseteq$ *plus' b1 b2*
begin

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

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

lemma *mono-step'*: *S* $\sqsubseteq$ *S'* $\Longrightarrow$ *c* $\sqsubseteq$ *c'* $\Longrightarrow$ *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* ⇒ *'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* ⊤)

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*)
by(*rule ext*)(*auto simp*: *lookup-def update-def*)

definition *γ-st γ F* = {*f*. ∀ *x*. *f x* ∈ *γ*(*lookup F x*)}

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

definition *le-st F G* = (∀ *x* ∈ *set*(*dom G*). *lookup F x* ⊑ *fun G x*)

definition
join-st F G =
FunDom (λ*x*. *fun F x* ⊔ *fun G x*) (*inter-list* (*dom F*) (*dom G*))

definition ⊤ = *FunDom* (λ*x*. ⊤) []

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*: $F \sqsubseteq F' \Longrightarrow lookup\ F\ x \sqsubseteq lookup\ F'\ x$
by(*auto simp add*: *lookup-def le-st-def*)

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

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

abbreviation γ_f :: *'av st* $\Rightarrow$ *state set*
where γ_f == γ-*st* γ

abbreviation γ_o :: *'av st option* $\Rightarrow$ *state set*
where γ_o == γ-*option* γ_f

abbreviation γ_c :: *'av st option acom* $\Rightarrow$ *state set acom*
where γ_c == *map-acom* γ_o

lemma *gamma-f-Top*[*simp*]: γ_f *Top* = *UNIV*
by(*auto simp*: *Top-st-def* γ*-st-def lookup-def*)

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

lemma *mono-gamma-f*: $f \sqsubseteq g \Longrightarrow \gamma_f\ f \subseteq \gamma_f\ g$
apply(*simp add*:γ*-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*:
$sa \sqsubseteq sa' \Longrightarrow \gamma_o\ sa \subseteq \gamma_o\ sa'$
by(*induction sa sa' rule*: *le-option.induct*)(*simp-all add*: *mono-gamma-f*)

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

lemma *in-gamma-option-iff*:
$x : \gamma\text{-}option\ r\ u \longleftrightarrow (\exists\, u'.\ u = Some\ u' \wedge x : r\ u')$
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* ⇒ $'av$ *st* ⇒ $'av$ **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*: $s : \gamma_f\ S \Longrightarrow aval\ a\ s : \gamma(aval'\ a\ S)$
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** γ=γ **for** γ :: $'av$::*SL-top* ⇒ *val set*
begin

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

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

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*:
  $[\![\ s : \gamma_f\ S;\ i : \gamma\ a\ ]\!] \Longrightarrow s(x := i) : \gamma_f(update\ S\ x\ a)$
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*:
  $[\![\ S \subseteq \gamma_o\ S';\ c \le \gamma_c\ c'\ ]\!] \Longrightarrow step\ S\ c \le \gamma_c\ (step'\ S'\ c')$

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**
$c' = IF\ b\ THEN\ c1'\ ELSE\ c2'\ \{P'\}$
$P \subseteq \gamma_o\ P'\ c1 \le \gamma_c\ c1'\ c2 \le \gamma_c\ c2'$
by (*fastforce simp*: *If-le map-acom-If*)
moreover have *post c1* $\subseteq \gamma_o$(*post c1'* $\sqcup$ *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 *post c2* $\subseteq \gamma_o$(*post c1'* $\sqcup$ *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** ‹$S \subseteq \gamma_o\ S'$› **by** (*simp add*: *If.IH subset-iff*)
next
case (*While I b c1 P*)
then obtain *c1' I' P'* **where**
$c' = \{I'\}\ WHILE\ b\ DO\ c1'\ \{P'\}$
$I \subseteq \gamma_o\ I'\ P \subseteq \gamma_o\ P'\ c1 \le \gamma_c\ c1'$
by (*fastforce simp*: *map-acom-While While-le*)
moreover have $S \cup$ *post c1* $\subseteq \gamma_o\ (S' \sqcup$ *post c1'*)
using ‹$S \subseteq \gamma_o\ S'$› *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'* $\Longrightarrow$ *CS c* $\le \gamma_c$ *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 lpfpc-pfp*[*OF 1*])
have *3*: $strip\ (\gamma_c\ (step'\ \top\ c')) = c$
by(*simp add*: *strip-lpfpc*[*OF - 1*])
have *lfp* (*step UNIV*) *c* $\le \gamma_c$ (*step'* $\top$ *c'*)
proof(*rule lfp-lowerbound*[*simplified*,*OF 3*])
show *step UNIV* (γ_c (*step'* $\top$ *c'*)) $\le \gamma_c$ (*step'* $\top$ *c'*)
proof(*rule step-preserves-le*[*OF - -*])
show *UNIV* $\subseteq \gamma_o\ \top$ **by** *simp*
show $\gamma_c\ (step'\ \top\ c') \le \gamma_c\ c'$ **by**(*rule mono-gamma-c*[*OF 2*])
qed
qed
from *this 2* **show** *lfp* (*step UNIV*) *c* $\le \gamma_c$ *c'*

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

end

6.1 Monotonicity

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

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

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

lemma *mono-step'*: $S \sqsubseteq S' \Longrightarrow c \sqsubseteq c' \Longrightarrow step'\ S\ c \sqsubseteq step'\ S'\ c'$
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* == $r \cap -(r^{-1})$
abbreviation *acc r* == $wf((strict\ r)^{-1})$

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

lemma *acc-inv-image*:
$acc\ r \Longrightarrow acc\ (inv\text{-}image\ r\ f)$
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** ‹*?Q* = {}› ‹*xo* : *Qo*›
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 bexI)
      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 ‹$?Y' \cap ?X <= ?X'$› **have** $?Y' \cap ?X - \{u\} <= ?X' - \{u\}$ **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** ‹$u \notin ?Y'$› **by** *auto*
thus *?thesis* **by** *metis*
qed
also have $\ldots < (\sum y \in ?Y' \cap ?X - \{u\}.\ m(?g\ y)+1) + (\sum y \in \{u\}.\ m(?f\ y)+1)$
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 $\ldots \leq (\sum y \in ?X' - \{u\}.\ m(?f\ y)+1)$
by(*simp add*: *sum-mono2*[*OF* - ‹$?Y' \cap ?X - \{u\} <= ?X' - \{u\}$›])
also have $\ldots + (\sum y \in \{u\}.\ m(?f\ y)+1) = (\sum y \in (?X' - \{u\}) \cup \{u\}.\ m(?f\ y)+1)$
using ‹u:$?X'$› **by**(*subst sum.union-disjoint*[*symmetric*]) *auto*
also have $\ldots = (\sum x \in ?X'.\ m(?f\ x)+1)$
using ‹$u : ?X'$› **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*:
$(x\#xs,\ y\#ys) : \mathit{listrel}\ r \longleftrightarrow (x,y) \in r \wedge (xs,ys) \in \mathit{listrel}\ r$
by (*blast intro*:*listrel.Cons*)

lemma *listrel-app*: $(xs1,ys1) : \mathit{listrel}\ r \Longrightarrow (xs2,ys2) : \mathit{listrel}\ r$
$\Longrightarrow (xs1@xs2,\ ys1@ys2) : \mathit{listrel}\ r$
by(*auto simp add*: *listrel-iff-zip*)

lemma *listrel-app-same-size*: *size xs1 = size ys1* $\Longrightarrow$ *size xs2 = size ys2* $\Longrightarrow$
$(xs1@xs2,\ ys1@ys2) : \mathit{listrel}\ r \longleftrightarrow$
$(xs1,ys1) : \mathit{listrel}\ r \wedge (xs2,ys2) : \mathit{listrel}\ r$
by(*auto simp add*: *listrel-iff-zip*)

lemma *listrel-converse*: $\mathit{listrel}(r\hat{}-1) = (\mathit{listrel}\ r)\hat{}-1$
proof−
{ fix *xs ys*
have $(xs,ys) : \mathit{listrel}(r\hat{}-1) \longleftrightarrow (ys,xs) : \mathit{listrel}\ r$
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** $r :: ('a*'a)set$ **assumes** *refl r* **and** *trans r*
and *acc r* **shows** *acc* (*listrel r* − {([],[])})
proof−
have *refl*: !!*x*. (*x*,*x*) : *r* **using** ‹*refl r*› **unfolding** *refl-on-def* **by** *blast*
have *trans*: !!*x y z*. (*x*,*y*) : *r* ⟹ (*y*,*z*) : *r* ⟹ (*x*,*z*) : *r*
using ‹*trans r*› **unfolding** *trans-def* **by** *blast*
from *assms*(*3*) **obtain** *mx* :: *'a set* ⇒ *'a* **where**
mx: !!*S x*. *x*:*S* ⟹ *mx S* : *S* ∧ (∀ *y*. (*mx S*,*y*) : *strict r* ⟶ *y* ∉ *S*)
by(*simp add*: *wf-eq-minimal*) *metis*
let *?R* = *listrel r* − {([], [])}
{ fix *Q* **and** *xs* :: *'a list*
have *xs* ∈ *Q* ⟹ ∃ *ys*. *ys*∈*Q* ∧ (∀ *zs*. (*ys*, *zs*) ∈ *strict ?R* ⟶ *zs* ∉ *Q*)
(**is** - ⟹ ∃ *ys*. *?P Q ys*)
proof(*induction xs arbitrary*: *Q rule*: *length-induct*)
case (*1 xs*)
{ have !!*ys Q*. *size ys* < *size xs* ⟹ *ys* : *Q* ⟹ ∃ *ms*. *?P Q ms*
using *1.IH* **by** *blast*
} note *IH* = *this*
show *?case*
proof(*cases xs*)
case *Nil* **with** ‹*xs* : *Q*› **have** *?P Q* [] **by** *auto*
thus *?thesis* **by** *blast*
next
case (*Cons x ys*)
let *?Q1* = {*a*. ∃ *bs*. *size bs* = *size ys* ∧ *a*#*bs* : *Q*}
have *x* : *?Q1* **using** ‹*xs* : *Q*› *Cons* **by** *auto*
from *mx*[*OF this*] **obtain** *m1* **where**
1: *m1* ∈ *?Q1* ∧ (∀ *y*. (*m1*,*y*) ∈ *strict r* ⟶ *y* ∉ *?Q1*) **by** *blast*
then obtain *ms1* **where** *size ms1* = *size ys m1*#*ms1* : *Q* **by** *blast*+
hence *size ms1* < *size xs* **using** *Cons* **by** *auto*
let *?Q2* = {*bs*. ∃ *m1'*. (*m1'*,*m1*):*r* ∧ (*m1*,*m1'*):*r* ∧ *m1'*#*bs* : *Q* ∧ *size bs* = *size ms1*}
have *ms1* : *?Q2* **using** ‹*m1*#*ms1* : *Q*› **by**(*blast intro*: *refl*)
from *IH*[*OF* ‹*size ms1* < *size xs*› *this*]
obtain *ms* **where** *2*: *?P ?Q2 ms* **by** *auto*
then obtain *m1'* **where** *m1'*: (*m1'*,*m1*) : *r* ∧ (*m1*,*m1'*) : *r* ∧ *m1'*#*ms* : *Q*
by *blast*
hence ∀ *ab*. (*m1'*#*ms*,*ab*) : *strict ?R* ⟶ *ab* ∉ *Q* **using** *1 2*
apply (*auto simp*: *listrel-Cons-iff*)
apply (*metis* ‹*length ms1* = *length ys*› *listrel-eq-len trans*)
by (*metis* ‹*length ms1* = *length ys*› *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* $\longleftrightarrow$
(*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'*})$\hat{}-1$ $\subseteq$
(*strict*(*inv-image* (*listrel*{(*a*,*a'*::'*a*). *a* $\sqsubseteq$ *a'*} $-$ {([],[])}) *annos*))$\hat{}-1$
by(*auto simp*: *le-iff-le-annos*)

lemma *acc-acom*: *acc* {(*a*,*a'*::'*a*::*preord*). *a* $\sqsubseteq$ *a'*} $\Longrightarrow$
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 *pfp-termination*:
fixes *x0* :: '*a*::*preord*
assumes *mono*: $\bigwedge x\ y.\ x \sqsubseteq y \Longrightarrow f\ x \sqsubseteq f\ y$ **and** *acc* {(*x*::'*a*,*y*). *x* $\sqsubseteq$ *y*}
and *x0* $\sqsubseteq$ *f x0* **shows** $\exists x.$ *pfp f x0* = *Some x*
proof(*simp add*: *pfp-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** $\bigwedge x\ y.\ x \sqsubseteq y \Longrightarrow f\ x \sqsubseteq f\ y$
and $\bigwedge c.$ *strip*(*f c*) = *strip c*
shows $\exists c'.$ *lpfp*$_c$ *f c* = *Some c'*
unfolding *lpfp*$_c$-*def*
apply(*rule pfp-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 (*strict*{(*x*,*y*::'*a*). *x* ⊑ *y*})^−*1* <= *measure m*
and ∀ *x y*::'*a*. *x* ⊑ *y* ∧ *y* ⊑ *x* ⟶ *m x* = *m y*
shows ∃ *c'*. *AI c* = *Some c'*
unfolding *AI-def*
apply(*rule lpfpc-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* = (*fst p1* ⊑ *fst p2* ∧ *snd p1* ⊑ *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* ⇒ '*a* ⇒ '*a* (**infixl** ‹⊓› *65*)
and *bot* :: '*a* (‹⊥›)
assumes *meet-le1* [*simp*]: *x* ⊓ *y* ⊑ *x*
and *meet-le2* [*simp*]: *x* ⊓ *y* ⊑ *y*
and *meet-greatest*: *x* ⊑ *y* ⟹ *x* ⊑ *z* ⟹ *x* ⊑ *y* ⊓ *z*
assumes *bot*[*simp*]: ⊥ ⊑ *x*
begin

lemma *mono-meet*: *x* ⊑ *x'* ⟹ *y* ⊑ *y'* ⟹ *x* ⊓ *y* ⊑ *x'* ⊓ *y'*

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

end

locale *Val-abs1-gamma* =
  Gamma **where** $\gamma = \gamma$ **for** γ :: *'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*]: $\gamma\ \bot = \{\}$
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 γ :: *'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** γ :: *'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 = $\bot$ |
aval'' e (*Some sa*) = *aval' e sa*

lemma *aval''-sound*: $s : \gamma_o\ S \Longrightarrow$ *aval a s* : γ(*aval'' a S*)
by(*cases S*)(*simp add*: *aval'-sound*)+

7.1 Backward analysis

```
fun afilter :: aexp ⇒ 'av ⇒ 'av st option ⇒ '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 ⇒ None | Some S ⇒
  let a' = lookup S x ⊓ a in
  if a' ⊑ ⊥ 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 $\bot$ in the V-case is important: $\bot$ indicates that a variable has no possible values, i.e. that the current program point is unreachable. But then the abstract state should collapse to *None*. Put differently, we maintain the invariant that in an abstract state of the form *Some s*, all variables are mapped to non-$\bot$ values. Otherwise the (pointwise) join of two abstract states, one of which contains $\bot$ values, may produce too large a result, thus making the analysis less precise.

```
fun bfilter :: bexp ⇒ bool ⇒ 'av st option ⇒ 'av st option where
bfilter (Bc v) res S = (if v=res then S else None) |
bfilter (Not b) res S = bfilter b (¬ res) S |
bfilter (And b1 b2) res S =
  (if res then bfilter b1 True (bfilter b2 True S)
   else bfilter b1 False S ⊔ 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 : γ_o S ⟹ aval e s : γ a ⟹ s : γ_o (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 : γ_f S' using ‹s : γ_o S›
    by(auto simp: in-gamma-option-iff)
  moreover hence s x : γ (lookup S' x) by(simp add: γ-st-def)
  moreover have s x : γ 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 : γ_o S ⟹ bv = bval b s ⟹ s : γ_o(bfilter b bv S)
proof(induction b arbitrary: S bv)
  case Bc thus ?case by simp
```

next
  case (*Not b*) **thus** *?case* **by** *simp*
next
  case (*And b1 b2*) **thus** *?case*
    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* ⇒ *'av st option acom* ⇒ *'av st option acom*
 where
step' S (*SKIP* {*P*}) = (*SKIP* {*S*}) |
step' S (*x* ::= *e* {*P*}) =
  x ::= *e* {*case S of None* ⇒ *None* | *Some S* ⇒ *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* ⊔ *post c2*}) |
step' S ({*Inv*} *WHILE b DO c* {*P*}) =
   {*S* ⊔ *post c*}
   WHILE b DO step' (*bfilter b True Inv*) *c*
   {*bfilter b False Inv*}

definition *AI* :: *com* ⇒ *'av st option acom option* **where**
AI = *lpfp*$_c$ (*step'* ⊤)

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*:
  ⟦ *s* : γ_f *S*; *i* : γ *a* ⟧ ⟹ *s*(*x* := *i*) : γ_f(*update S x a*)
by(*simp add*: γ-*st-def lookup-update*)

lemma *step-preserves-le*:
  ⟦ *S* ⊆ γ_o *S'*; *cs* ≤ γ_c *ca* ⟧ ⟹ *step S cs* ≤ γ_c (*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-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 cs1 cs2 P*)
then obtain *ca1 ca2 Pa* **where**
ca= *IF b THEN ca1 ELSE ca2* {*Pa*}
$P \subseteq \gamma_o$ *Pa cs1* $\leq \gamma_c$ *ca1 cs2* $\leq \gamma_c$ *ca2*
by (*fastforce simp*: *If-le map-acom-If*)
moreover have *post cs1* $\subseteq \gamma_o$(*post ca1* $\sqcup$ *post ca2*)
by (*metis* (*no-types*) ‹*cs1* $\leq \gamma_c$ *ca1*› *join-ge1 le-post mono-gamma-o order-trans post-map-acom*)
moreover have *post cs2* $\subseteq \gamma_o$(*post ca1* $\sqcup$ *post ca2*)
by (*metis* (*no-types*) ‹*cs2* $\leq \gamma_c$ *ca2*› *join-ge2 le-post mono-gamma-o order-trans post-map-acom*)
ultimately show *?case* **using** ‹$S \subseteq \gamma_o$ 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 $\subseteq \gamma_o$ *Ia P* $\subseteq \gamma_o$ *Pa cs1* $\leq \gamma_c$ *ca1*
by (*fastforce simp*: *map-acom-While While-le*)
moreover have $S \cup$ *post cs1* $\subseteq \gamma_o$ ($S' \sqcup$ *post ca1*)
using ‹$S \subseteq \gamma_o$ S'› *le-post*[*OF* ‹*cs1* $\leq \gamma_c$ *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'* $\Longrightarrow$ *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 lpfpc-pfp*[*OF 1*])
have *3*: *strip* (γ_c (*step'* $\top$ *c'*)) = *c*
by(*simp add*: *strip-lpfpc*[*OF - 1*])
have *lfp* (*step UNIV*) *c* $\leq \gamma_c$ (*step'* $\top$ *c'*)
proof(*rule lfp-lowerbound*[*simplified*,*OF 3*])
show *step UNIV* (γ_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 γ_c (*step'* $\top$ *c'*) $\leq \gamma_c$ *c'* **by**(*rule mono-gamma-c*[*OF 2*])
qed
qed
from *this 2* **show** *lfp* (*step UNIV*) *c* $\leq \gamma_c$ *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 *domo S* = (*case S of None* ⇒ {} | *Some S'* ⇒ *set*(*dom S'*))

definition *Com* :: *vname set* ⇒ *'a st option acom set* **where**
Com X = {*c*. ∀ *S* ∈ *set*(*annos c*). *domo S* ⊆ *X*}

lemma *domo-Top*[*simp*]: *domo* ⊤ = {}
by(*simp add*: *domo-def Top-st-def Top-option-def*)

lemma *bot-acom-Com*[*simp*]: ⊥$_c$ *c* ∈ *Com X*
by(*simp add*: *bot-acom-def Com-def domo-def*)

lemma *post-in-annos*: *post c* : *set*(*annos c*)
by(*induction c*) *simp-all*

lemma *domo-join*: *domo* (*S* ⊔ *T*) ⊆ *domo S* ∪ *domo T*
by(*auto simp*: *domo-def join-st-def split*: *option.split*)

lemma *domo-afilter*: *vars a* ⊆ *X* ⟹ *domo S* ⊆ *X* ⟹ *domo*(*afilter a i S*) ⊆ *X*
apply(*induction a arbitrary*: *i S*)
apply(*simp add*: *domo-def*)
apply(*simp add*: *domo-def Let-def update-def lookup-def split*: *option.splits*)
apply *blast*
apply(*simp split*: *prod.split*)
done

lemma *domo-bfilter*: *vars b* ⊆ *X* ⟹ *domo S* ⊆ *X* ⟹ *domo*(*bfilter b bv S*) ⊆ *X*
apply(*induction b arbitrary*: *bv S*)
apply(*simp add*: *domo-def*)
apply(*simp*)
apply(*simp*)
apply *rule*
apply (*metis le-sup-iff subset-trans*[*OF domo-join*])
apply(*simp split*: *prod.split*)
by (*metis domo-afilter*)

lemma *step'-Com*:
domo S ⊆ *X* ⟹ *vars*(*strip c*) ⊆ *X* ⟹ *c* : *Com X* ⟹ *step' S c* : *Com X*
apply(*induction c arbitrary*: *S*)
apply(*simp add*: *Com-def*)
apply(*simp add*: *Com-def domo-def update-def split*: *option.splits*)
apply(*simp* (*no-asm-use*) *add*: *Com-def ball-Un*)
apply (*metis post-in-annos*)
apply(*simp* (*no-asm-use*) *add*: *Com-def ball-Un*)
apply *rule*
apply (*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'*: $a1 \sqsubseteq b1 \Longrightarrow a2 \sqsubseteq b2 \Longrightarrow plus'\ a1\ a2 \sqsubseteq plus'\ b1\ b2$
and *mono-filter-plus'*: $a1 \sqsubseteq b1 \Longrightarrow a2 \sqsubseteq b2 \Longrightarrow r \sqsubseteq r' \Longrightarrow$
filter-plus' r a1 a2 $\sqsubseteq$ *filter-plus' r' b1 b2*
and *mono-filter-less'*: $a1 \sqsubseteq b1 \Longrightarrow a2 \sqsubseteq b2 \Longrightarrow$
filter-less' bv a1 a2 $\sqsubseteq$ *filter-less' bv b1 b2*
begin

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

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

lemma *mono-afilter*: $r \sqsubseteq r' \Longrightarrow S \sqsubseteq S' \Longrightarrow afilter\ e\ r\ S \sqsubseteq afilter\ e\ r'\ S'$
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*: $S \sqsubseteq S' \Longrightarrow bfilter\ b\ r\ S \sqsubseteq bfilter\ b\ r\ S'$
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 snd-conv*)
done

lemma *mono-step'*: $S \sqsubseteq S' \Longrightarrow c \sqsubseteq c' \Longrightarrow step'\ S\ c \sqsubseteq step'\ S'\ c'$
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*) ⇒ {*l*..*h*} |
I (*Some l*) *None* ⇒ {*l*..} |
I None (*Some h*) ⇒ {..*h*} |
I None None ⇒ *UNIV*)

abbreviation *I-Some-Some* :: *int* ⇒ *int* ⇒ *ivl* (‹{-...-}›) **where**
{*lo*...*hi*} == *I* (*Some lo*) (*Some hi*)
abbreviation *I-Some-None* :: *int* ⇒ *ivl* (‹{-...}›) **where**
{*lo*...} == *I* (*Some lo*) *None*
abbreviation *I-None-Some* :: *int* ⇒ *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* ⇒ *ivl* ⇒ *bool* **where**
in-ivl k (*I* (*Some l*) (*Some h*)) ⟷ *l* ≤ *k* ∧ *k* ≤ *h* |
in-ivl k (*I* (*Some l*) *None*) ⟷ *l* ≤ *k* |
in-ivl k (*I None* (*Some h*)) ⟷ *k* ≤ *h* |
in-ivl k (*I None None*) ⟷ *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...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* ⇒ (*case h of Some h* ⇒ *h*<*l* | *None* ⇒ *False*) | *None* ⇒ *False*)
by(*auto split*:*option.split*)

lemma [*simp*]: *is-empty i* ⟹ *γ-ivl i* = {}
by(*auto simp add*: *γ-ivl-def split*: *ivl.split option.split*)

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*) ⇒ *I* (*l1*+*l2*) (*h1*+*h2*))

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

definition *le-option* :: *bool* ⇒ *int option* ⇒ *int option* ⇒ *bool* **where**
le-option pos x y =
(*case x of* (*Some i*) ⇒ (*case y of Some j* ⇒ *i*≤*j* | *None* ⇒ *pos*)
| *None* ⇒ (*case y of Some j* ⇒ ¬*pos* | *None* ⇒ *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 ⊑ *i2* =
(*if is-empty i1 then True else*
if is-empty i2 then False else le-aux i1 i2)

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

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

definition *i1* ⊔ *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*) ⇒
I (*min-option False l1 l2*) (*max-option True h1 h2*))

definition ⊤ = {...}

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* ⊓ *i2* = (*if is-empty i1* ∨ *is-empty i2 then empty else*
  case (*i1*,*i2*) *of* (*I l1 h1*, *I l2 h2*) ⇒
    I (*max-option False l1 l2*) (*min-option True h1 h2*))

definition ⊥ = *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 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 γ = *γ-ivl* **and** *num′* = *num-ivl* **and** *plus′* = *plus-ivl*
defines *aval-ivl* = *aval′*
proof (*standard*, *goal-cases*)
  case *1* **thus** *?case*
   by(*auto simp add*: *γ-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 γ-ivl-def empty-def*)
qed

lemma *mono-minus-ivl*:
  i1 $\sqsubseteq$ *i1′* $\Longrightarrow$ *i2* $\sqsubseteq$ *i2′* $\Longrightarrow$ *minus-ivl i1 i2* $\sqsubseteq$ *minus-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 γ = *γ-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*
proof (*standard*, *goal-cases*)
  case *1* **thus** *?case*
    by (*simp add*: *γ-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: *γ-ivl-def min-option-def max-option-def le-option-def split*: *if-splits option.splits*)
qed

global-interpretation *Abs-Int1*
where γ = *γ-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 *afilter-ivl* = *afilter*
and *bfilter-ivl* = *bfilter*
and *step-ivl* = *step′*
and *AI-ivl* = *AI*
and *aval-ivl′* = *aval″*
..

Monotonicity:

global-interpretation *Abs-Int1-mono*
where $\gamma = \gamma$*-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*
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* ⊤)^^*0*) (⊥$_c$ *test2-ivl*))
value *show-acom* (((*step-ivl* ⊤)^^*1*) (⊥$_c$ *test2-ivl*))
value *show-acom* (((*step-ivl* ⊤)^^*2*) (⊥$_c$ *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* ⊤)^^*0*) (⊥$_c$ *test3-ivl*))
value *show-acom* (((*step-ivl* ⊤)^^*1*) (⊥$_c$ *test3-ivl*))
value *show-acom* (((*step-ivl* ⊤)^^*2*) (⊥$_c$ *test3-ivl*))
value *show-acom* (((*step-ivl* ⊤)^^*3*) (⊥$_c$ *test3-ivl*))
value *show-acom* (((*step-ivl* ⊤)^^*4*) (⊥$_c$ *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* ⊤)^^*50*) (⊥$_c$ *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* $\top$)$^{\frown}$*50*) ($\bot_c$ *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** ‹∇› *65*)
assumes *widen1*: $x \sqsubseteq x \nabla y$
assumes *widen2*: $y \sqsubseteq x \nabla y$
fixes *narrow* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infix** ‹$\triangle$› *65*)
assumes *narrow1*: $y \sqsubseteq x \Longrightarrow y \sqsubseteq x \triangle y$
assumes *narrow2*: $y \sqsubseteq x \Longrightarrow x \triangle y \sqsubseteq x$

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*) ⇒
I (*if le-option False l2 l1* ∧ *l2* ≠ *l1 then None else l1*)
(*if le-option True h1 h2* ∧ *h1* ≠ *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*) ⇒
I (*if l1* = *None then l2 else l1*)
(*if h1* = *None then h2 else h1*))

instance
proof qed
(*auto simp add*: *widen-ivl-def narrow-ivl-def le-option-def le-ivl-def empty-def split*: *ivl.split option.split if-splits*)

end

9.2 Abstract State

instantiation *st* :: (*WN*) *WN*
begin

definition *widen-st F1 F2* =

FunDom (λ*x*. *fun F1 x* ▽ *fun F2 x*) (*inter-list* (*dom F1*) (*dom F2*))

definition *narrow-st F1 F2* =
FunDom (λ*x*. *fun F1 x* △ *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 ▽ *x* = *x* |
x ▽ *None* = *x* |
(*Some x*) ▽ (*Some y*) = *Some*(*x* ▽ *y*)

fun *narrow-option* **where**
None △ *x* = *None* |
x △ *None* = *None* |
(*Some x*) △ (*Some y*) = *Some*(*x* △ *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* :: $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a\ acom \Rightarrow 'a\ acom \Rightarrow 'a\ acom$ **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** $\langle\nabla_c\rangle$ *65*)
where *widen-acom* == *map2-acom* (∇)

abbreviation *narrow-acom* :: $('a::WN)acom \Rightarrow 'a\ acom \Rightarrow 'a\ acom$ (**infix** $\langle\triangle_c\rangle$ *65*)
where *narrow-acom* == *map2-acom* ($\triangle$)

lemma *widen1-acom*: *strip c* = *strip c'* $\Longrightarrow c \sqsubseteq c\ \nabla_c\ c'$
by(*induct c c' rule*: *le-acom.induct*)(*simp-all add*: *widen1*)

lemma *widen2-acom*: *strip c* = *strip c'* $\Longrightarrow c' \sqsubseteq c\ \nabla_c\ c'$
by(*induct c c' rule*: *le-acom.induct*)(*simp-all add*: *widen2*)

lemma *narrow1-acom*: $y \sqsubseteq x \Longrightarrow y \sqsubseteq x\ \triangle_c\ y$
by(*induct y x rule*: *le-acom.induct*) (*simp-all add*: *narrow1*)

lemma *narrow2-acom*: $y \sqsubseteq x \Longrightarrow x\ \triangle_c\ y \sqsubseteq x$
by(*induct y x rule*: *le-acom.induct*) (*simp-all add*: *narrow2*)

9.5 Post-fixed point computation

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

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

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

lemma *strip-map2-acom*:
strip c1 = strip c2 ⟹ 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′ ⟹ f c′ ⊑ c′*
by(*auto simp add*: *iter-widen-def dest*: *while-option-stop*)

lemma *strip-while*: **fixes** *f* :: *′a acom ⇒ ′a acom*
assumes ∀ *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* = *λc′*. *strip c′ = strip c*, *OF* - *assms*(*2*)]
by (*metis assms*(*1*))

lemma *strip-iter-widen*: **fixes** *f* :: *′a*::*WN acom ⇒ ′a acom*
assumes ∀ *c*. *strip* (*f c*) = *strip c* **and** *iter-widen f c = Some c′*
shows *strip c′ = strip c*
proof−
have ∀ *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* ⊑ *c0*
and *iter-narrow f c0 = Some c*
shows *f c* ⊑ *c* ∧ *c* ⊑ *c0* (**is** *?P c*)
proof−
{ fix *c* **assume** *?P c*
note *1* = *conjunct1*[*OF this*] **and** *2* = *conjunct2*[*OF this*]
let *?c′* = *c* △*c f c*
have *?P ?c′*
proof
have *f ?c′* ⊑ *f c* **by**(*rule monoD*[*OF ‹mono f›* *narrow2-acom*[*OF 1*]])
also have *. . .* ⊑ *?c′* **by**(*rule narrow1-acom*[*OF 1*])
finally show *f ?c′* ⊑ *?c′* **.**
have *?c′* ⊑ *c* **by** (*rule narrow2-acom*[*OF 1*])
also have *c* ⊑ *c0* **by**(*rule 2*)
finally show *?c′* ⊑ *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*:
⟦ *mono f*; *pfp-wn f c = Some c′* ⟧ ⟹ *f c′* ⊑ *c′*
unfolding *pfp-wn-def*
by (*auto dest*: *iter-widen-pfp iter-narrow-pfp split*: *option.splits*)

lemma *strip-pfp-wn*:

⟦ ∀ c. strip(f c) = strip c; pfp-wn f c = Some c' ⟧ ⟹ 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 γ=γ **for** γ :: 'av::{*WN*,*L-top-bot*} ⇒ *val set*
begin

definition *AI-wn* :: *com* ⇒ 'av *st option acom option* **where**
AI-wn = *pfp-wn* (*step'* ⊤)

lemma *AI-wn-sound*: *AI-wn c* = *Some c'* ⟹ *CS c* ≤ γ_c *c'*
proof(*simp add*: *CS-def AI-wn-def*)
assume *1*: *pfp-wn* (*step'* ⊤) *c* = *Some c'*
from *pfp-wn-pfp*[*OF mono-step'2 1*]
have *2*: *step'* ⊤ *c'* ⊑ *c'* **.**
have *3*: *strip* (γ_c (*step'* ⊤ *c'*)) = *c* **by**(*simp add*: *strip-pfp-wn*[*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

end

global-interpretation *Abs-Int2*
where γ = *γ-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* = ((λc. *c* ∇_c *step-ivl* ⊤ *c*)^^*n*)
definition *step-down-ivl n* = ((λc. *c* △_c *step-ivl* ⊤ *c*)^^*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* (⊥_c *test3-ivl*))
value *show-acom* (*step-up-ivl 2* (⊥_c *test3-ivl*))
value *show-acom* (*step-up-ivl 3* (⊥_c *test3-ivl*))
value *show-acom* (*step-up-ivl 4* (⊥_c *test3-ivl*))

value *show-acom* (*step-up-ivl 5* ($\bot_c$ *test3-ivl*))
value *show-acom* (*step-down-ivl 1* (*step-up-ivl 5* ($\bot_c$ *test3-ivl*)))
value *show-acom* (*step-down-ivl 2* (*step-up-ivl 5* ($\bot_c$ *test3-ivl*)))
value *show-acom* (*step-down-ivl 3* (*step-up-ivl 5* ($\bot_c$ *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* ⇒ *nat* **where**
m-ivl ivl = (*case ivl of I l h* ⇒
(*case l of None* ⇒ *0* | *Some -* ⇒ *1*) + (*case h of None* ⇒ *0* | *Some -* ⇒ *1*))

lemma *m-ivl-height*: *m-ivl ivl* ≤ *2*
by(*simp add*: *m-ivl-def split*: *ivl.split option.split*)

lemma *m-ivl-anti-mono*: (*y*::*ivl*) ⊑ *x* ⟹ *m-ivl x* ≤ *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*:
~ *y* ⊑ *x* ⟹ *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*: ⊤ ⊑ *x* ⟹ *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* ⇒ *nat* **where**
n-ivl ivl = *2* − *m-ivl ivl*

lemma *n-ivl-mono*: (*x*::*ivl*) ⊑ *y* ⟹ *n-ivl x* ≤ *n-ivl y*
unfolding *n-ivl-def* **by** (*metis diff-le-mono2 m-ivl-anti-mono*)

lemma *n-ivl-narrow*:
~ *x* ⊑ *x* △ *y* ⟹ *n-ivl*(*x* △ *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* = (∑ *x*∈*set*(*dom st*). *m*(*fun st x*))

lemma *m-st-height*: **assumes** *finite X* **and** *set* (*dom S*) ⊆ *X*
shows *m-st m-ivl S* ≤ *2* ∗ *card X*

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

lemma *m-st-anti-mono*:
S1 $\sqsubseteq$ *S2* $\Longrightarrow$ *m-st m-ivl S2* $\leq$ *m-st m-ivl S1*
proof(*auto simp*: *m-st-def le-st-def lookup-def split*: *if-splits*)
let *?X* = *set*(*dom S1*) **let** *?Y* = *set*(*dom S2*)
let *?f* = *fun S1* **let** *?g* = *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-ivl*(*?g y*) $\leq$ *m-ivl*(*?f y*) **by**(*simp add*: *m-ivl-anti-mono*)
have *0*: $\forall x{\in}?Y - ?X.$ *m-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 ... $= (\sum y{\in}?Y - ?X.\ m\text{-}ivl(?g\ y)) + (\sum y{\in}?Y \cap ?X.\ m\text{-}ivl(?g\ y))$
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 ... $\leq (\sum y{\in}?Y \cap ?X.\ m\text{-}ivl(?f\ y))$
by(*rule sum-mono*)(*simp add*: *1*)
also have ... $\leq (\sum y{\in}?X.\ m\text{-}ivl(?f\ y))$
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-st m-ivl* (*S1* ∇ *S2*) $<$ *m-st m-ivl S1*
proof−
{ let *?X* = *set*(*dom S1*) **let** *?Y* = *set*(*dom S2*)
let *?f* = *fun S1* **let** *?g* = *fun S2*
fix *x* **assume** *x* $\in$ *?X* $\neg$ *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 ‹*x*:*?X*› ‹*x*:*?Y*› ‹$\neg$ *lookup S2 x* $\sqsubseteq$ *?f x*›
by (*metis IntI m-ivl-widen lookup-def*)

```
    qed
    also have ... ≤ ?R by(simp add: sum-mono2[OF - Int-lower1])
    finally show ?thesis .
  next
    assume x ~: ?Y
    have ?L ≤ (∑ x∈?X∩?Y. m-ivl(?f x))
    proof(rule sum-mono, simp)
      fix x assume x:?X ∧ x:?Y show m-ivl(?f x ∇ ?g x) ≤ m-ivl (?f x)
        by (metis m-ivl-anti-mono widen1)
    qed
    also have ... < m-ivl(?f x) + ...
      using m-ivl-widen[OF ‹¬ lookup S2 x ⊑ ?f x›]
      by (metis Nat.le-refl add-strict-increasing gr0I not-less0)
    also have ... = (∑ y∈insert x (?X∩?Y). m-ivl(?f y))
      using ‹x ~: ?Y› by simp
    also have ... ≤ (∑ x∈?X. m-ivl(?f x))
      by(rule sum-mono2)(insert ‹x:?X›, 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-st m X st = (∑ x∈X. m(lookup st x))

lemma n-st-mono: assumes set(dom S1) ⊆ X set(dom S2) ⊆ X S1 ⊑ S2
shows n-st n-ivl X S1 ≤ n-st n-ivl X S2
proof−
 have (∑ x∈X. n-ivl(lookup S1 x)) ≤ (∑ x∈X. n-ivl(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-st-narrow:
assumes finite X and set(dom S1) ⊆ X set(dom S2) ⊆ X
and S2 ⊑ S1 ¬ S1 ⊑ S1 △ S2
shows n-st n-ivl X (S1 △ S2) < n-st n-ivl X S1
proof−
 have 1: ∀ x∈X. n-ivl (lookup (S1 △ S2) x) ≤ n-ivl (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: ∃ x∈X. n-ivl (lookup (S1 △ S2) x) < n-ivl (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 (∑ x∈X. n-ivl(lookup (S1 △ S2) x)) < (∑ x∈X. n-ivl(lookup S1 x))
  apply(rule sum-strict-mono-ex1[OF ‹finite X›]) using 1 2 by blast+
```

thus *?thesis* **by**(*simp add*: *n-st-def*)
qed

9.9 Termination: Option

definition *m-o m n opt* = (*case opt of None* ⇒ *n+1* | *Some x* ⇒ *m x*)

lemma *m-o-anti-mono*: *finite X* ⟹ *domo S2* ⊆ *X* ⟹ *S1* ⊑ *S2* ⟹
m-o (*m-st m-ivl*) (*2* ∗ *card X*) *S2* ≤ *m-o* (*m-st m-ivl*) (*2* ∗ *card X*) *S1*
apply(*induction S1 S2 rule*: *le-option.induct*)
apply(*auto simp*: *domo-def m-o-def m-st-anti-mono le-SucI m-st-height*
split: *option.splits*)
done

lemma *m-o-widen*: ⟦ *finite X*; *domo S2* ⊆ *X*; ¬ *S2* ⊑ *S1* ⟧ ⟹
m-o (*m-st m-ivl*) (*2* ∗ *card X*) (*S1* ▽ *S2*) < *m-o* (*m-st m-ivl*) (*2* ∗ *card X*) *S1*
by(*auto simp*: *m-o-def domo-def m-st-height less-Suc-eq-le m-st-widen*
split: *option.split*)

definition *n-o n opt* = (*case opt of None* ⇒ *0* | *Some x* ⇒ *n x* + *1*)

lemma *n-o-mono*: *domo S1* ⊆ *X* ⟹ *domo S2* ⊆ *X* ⟹ *S1* ⊑ *S2* ⟹
n-o (*n-st n-ivl X*) *S1* ≤ *n-o* (*n-st n-ivl X*) *S2*
apply(*induction S1 S2 rule*: *le-option.induct*)
apply(*auto simp*: *domo-def n-o-def n-st-mono*
split: *option.splits*)
done

lemma *n-o-narrow*:
⟦ *finite X*; *domo S1* ⊆ *X*; *domo S2* ⊆ *X*; *S2* ⊑ *S1*; ¬ *S1* ⊑ *S1* △ *S2* ⟧
⟹ *n-o* (*n-st n-ivl X*) (*S1* △ *S2*) < *n-o* (*n-st n-ivl X*) *S1*
apply(*induction S1 S2 rule*: *narrow-option.induct*)
apply(*auto simp*: *n-o-def domo-def n-st-narrow*)
done

lemma *domo-widen-subset*: *domo* (*S1* ▽ *S2*) ⊆ *domo S1* ∪ *domo S2*
apply(*induction S1 S2 rule*: *widen-option.induct*)
apply (*auto simp*: *domo-def widen-st-def*)
done

lemma *domo-narrow-subset*: *domo* (*S1* △ *S2*) ⊆ *domo S1* ∪ *domo S2*
apply(*induction S1 S2 rule*: *narrow-option.induct*)
apply (*auto simp*: *domo-def narrow-st-def*)
done

9.10 Termination: Commands

lemma *strip-widen-acom*[*simp*]:
strip c' = *strip* (*c*::*'a*::*WN acom*) ⟹ *strip* (*c* ∇$_c$ *c'*) = *strip c*
by(*induction widen*::*'a*⇒*'a*⇒*'a c c' rule*: *map2-acom.induct*) *simp-all*

lemma *strip-narrow-acom*[*simp*]:
strip c′ = *strip* (*c*::′*a*::*WN acom*) ⟹ *strip* (*c* △$_c$ *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*) ⟹
annos(*c1* ▽$_c$ *c2*) = *map* (%(*x*,*y*).*x*▽*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*) ⟹
annos(*c1* △$_c$ *c2*) = *map* (%(*x*,*y*).*x*△*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* ⟹
c1 : *Com X* ⟹ *c2* : *Com X* ⟹ (*c1* ▽$_c$ *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* ⟹
c1 : *Com X* ⟹ *c2* : *Com X* ⟹ (*c1* △$_c$ *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* ∑ *i*=*0*..<*size as*. *m*(*as*!*i*))

lemma *measure-m-c*: *finite X* ⟹ {(*c*, *c* ▽$_c$ *c′*) |*c c′*::*ivl st option acom*.
strip c′ = *strip c* ∧ *c* : *Com X* ∧ *c′* : *Com X* ∧ ¬ *c′* ⊑ *c*}$^{-1}$

$\subseteq$ *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** *bexI*)
prefer *2* **apply** *simp*
apply(*erule m-o-widen*)
apply (*simp*)+
done

lemma *measure-n-c*: *finite X* $\Longrightarrow$ {(*c*, *c* $\triangle_c$ *c'*) |*c c'*.
strip c = *strip c'* $\wedge$ *c* $\in$ *Com X* $\wedge$ *c'* $\in$ *Com X* $\wedge$ *c'* $\sqsubseteq$ *c* $\wedge$ $\neg$ *c* $\sqsubseteq$ *c* $\triangle_c$ *c'*}$^{-1}$
$\subseteq$ *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** *bexI*)
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* $\Longrightarrow$ *P*(*f c*)
assumes *P-widen*: $\bigwedge$*c c'*. *P c* $\Longrightarrow$ *P c'* $\Longrightarrow$ *P*(*c* ∇_c *c'*)
and *wf*({(*c*::*'a acom*,*c* ∇_c *c'*)|*c c'*. *P c* $\wedge$ *P c'* $\wedge$ $\sim$ *c'* $\sqsubseteq$ *c*}$\widehat{\ }-1$)
and *P c0* **and** *c0* $\sqsubseteq$ *f c0* **shows** $\exists$ *c*. *iter-widen f c0* = *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* ‹*P c0*›)
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*: $\bigwedge c.\ P\ c \Longrightarrow P(c \triangle_c\ f\ c)$
and *wf*: $wf(\{(c,\ c \triangle_c\ f\ c) | c\ c'.\ P\ c \wedge {\sim}\ c \sqsubseteq c \triangle_c\ f\ c\}^{-1})$
and *P c0* **shows** $\exists\, c.$ *iter-narrow f c0* $=$ *Some c*
proof(*simp add*: *iter-narrow-def*, *rule wf-while-option-Some*[**where** $P = P$])
  show *wf* $\{(c', c).\ (P\ c \wedge \neg\ c \sqsubseteq c \triangle_c\ f\ c) \wedge c' = c \triangle_c\ f\ c\}$
    apply(*rule wf-subset*[*OF wf*]) **by**(*blast intro*: *P-f*)
next
  show *P c0* **by**(*rule* ‹*P c0*›)
next
  fix *c* **assume** *P c* **thus** *P* $(c \triangle_c\ f\ c)$ **by**(*simp add*: *P-f*)
qed

lemma *iter-winden-step-ivl-termination*:
  $\exists\, c.$ *iter-widen* (*step-ivl* $\top$) $(\bot_c\ c0) =$ *Some c*
apply(*rule iter-widen-termination*[**where**
  $P = \%c.$ *strip c* $=$ *c0* $\wedge$ *c* : *Com*(*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** $X =$ *vars c0*, *OF finite-cvars*])
apply *blast*
done

lemma *iter-narrow-step-ivl-termination*:
  c0 $\in$ *Com* (*vars*(*strip c0*)) $\Longrightarrow$ *step-ivl* $\top$ *c0* $\sqsubseteq$ *c0* $\Longrightarrow$
  $\exists\, c.$ *iter-narrow* (*step-ivl* $\top$) *c0* $=$ *Some c*
apply(*rule iter-narrow-termination*[**where**
  $P = \%c.$ *strip c* $=$ *strip c0* $\wedge$ *c* : *Com*(*vars*(*strip c0*)) $\wedge$ *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 = *vars*(*strip c0*), *OF finite-cvars*])
apply *auto*
by (*metis bot-least domo-Top order-refl step'-Com strip-step'*)

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

lemma *iter-widen-Com*: **fixes** f :: $'a$::*WN st option acom* $\Rightarrow$ $'a$ *st option acom*
assumes *iter-widen f c* = *Some c'*
and $\forall$ c. c : *Com*(X) $\longrightarrow$ *vars*(*strip c*) $\subseteq$ X $\longrightarrow$ f c : *Com*(X)
and !!c. *strip*(f c) = *strip c*
and c : *Com*(X) **and** *vars* (*strip c*) $\subseteq$ X **shows** c' : *Com*(X)
proof−
  have $\forall$ c. c : *Com*(X) $\longrightarrow$ *vars*(*strip c*) $\subseteq$ X $\longrightarrow$ c ∇_c f c : *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*:
  iter-widen (*step'* $\top$) c = *Some c'* $\Longrightarrow$ *vars*(*strip c*) $\subseteq$ X $\Longrightarrow$ c : *Com*(X)
  $\Longrightarrow$ c' : *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'. *AI-ivl'* c = *Some c'*
apply(*auto simp*: *AI-wn-def pfp-wn-def iter-winden-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.