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* ⇒ '*a*) ⇒ '*i* ⇒ '*a* **where**
lfp f i = *Glb i* {*a* : *L i*. *f a* ≤ *a*}

lemma *index-lfp*: *lfp f i* : *L i*
⟨*proof*⟩

lemma *lfp-lowerbound*:
⟦ *a* : *L i*; *f a* ≤ *a* ⟧ ⟹ *lfp f i* ≤ *a*
⟨*proof*⟩

lemma *lfp-greatest*:
⟦ *a* : *L i*; ⋀*u*. ⟦ *u* : *L i*; *f u* ≤ *u*⟧ ⟹ *a* ≤ *u* ⟧ ⟹ *a* ≤ *lfp f i*
⟨*proof*⟩

lemma *lfp-unfold*: **assumes** ⋀*x i*. *f x* : *L i* ⟷ *x* : *L i*
and *mono*: *mono f* **shows** *lfp f i* = *f* (*lfp f i*)
⟨*proof*⟩

end

end

2 Annotated Commands

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

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

fun *post* :: '*a acom* ⇒'*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*)
⟨*proof*⟩

lemma *strip-acom*[*simp*]: *strip* (*map-acom f c*) = *strip c*
⟨*proof*⟩

lemma *map-acom-SKIP*:
 map-acom f c = *SKIP* {*S'*} ⟷ (∃ *S*. *c* = *SKIP* {*S*} ∧ *S'* = *f S*)
⟨*proof*⟩

lemma *map-acom-Assign*:
 map-acom f c = *x* ::= *e* {*S'*} ⟷ (∃ *S*. *c* = *x*::=*e* {*S*} ∧ *S'* = *f S*)
⟨*proof*⟩

lemma *map-acom-Seq*:
 map-acom f c = *c1'*;;*c2'* ⟷
 (∃ *c1 c2*. *c* = *c1*;;*c2* ∧ *map-acom f c1* = *c1'* ∧ *map-acom f c2* = *c2'*)
⟨*proof*⟩

lemma *map-acom-If*:

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

lemma *map-acom-While*:
$map\text{-}acom\ f\ w = \{I'\}\ WHILE\ b\ DO\ c'\ \{P'\} \longleftrightarrow$
$(\exists I\ P\ c.\ w = \{I\}\ WHILE\ b\ DO\ c\ \{P\} \wedge map\text{-}acom\ f\ c = c' \wedge I' = f\ I \wedge P' = f\ P)$
⟨*proof*⟩

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

lemma *strip-eq-SKIP*:
$strip\ c = com.SKIP \longleftrightarrow (\exists P.\ c = SKIP\ \{P\})$
⟨*proof*⟩

lemma *strip-eq-Assign*:
$strip\ c = x{::=}e \longleftrightarrow (\exists P.\ c = x{::=}e\ \{P\})$
⟨*proof*⟩

lemma *strip-eq-Seq*:
$strip\ c = c1;;c2 \longleftrightarrow (\exists d1\ d2.\ c = d1;;d2\ \&\ strip\ d1 = c1\ \&\ strip\ d2 = c2)$
⟨*proof*⟩

lemma *strip-eq-If*:
$strip\ c = IF\ b\ THEN\ c1\ ELSE\ c2 \longleftrightarrow$
$(\exists d1\ d2\ P.\ c = IF\ b\ THEN\ d1\ ELSE\ d2\ \{P\}\ \&\ strip\ d1 = c1\ \&\ strip\ d2 = c2)$
⟨*proof*⟩

lemma *strip-eq-While*:
$strip\ c = WHILE\ b\ DO\ c1 \longleftrightarrow$
$(\exists I\ d1\ P.\ c = \{I\}\ WHILE\ b\ DO\ d1\ \{P\}\ \&\ strip\ d1 = c1)$
⟨*proof*⟩

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

lemma *size-annos-same*: $strip\ C1 = strip\ C2 \Longrightarrow size(annos\ C1) = size(annos\ C2)$
⟨*proof*⟩

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

end

3 Collecting Semantics of Commands

theory *Collecting*
imports *Complete-Lattice-ix ACom*
begin

3.1 Annotated commands as a complete lattice

instantiation *acom* :: (*order*) *order*
begin

fun *less-eq-acom* :: (*'a::order*)*acom* ⇒ *'a acom* ⇒ *bool* **where**
(*SKIP* {*S*}) ≤ (*SKIP* {*S'*}) = (*S* ≤ *S'*) |
(*x* ::= *e* {*S*}) ≤ (*x'* ::= *e'* {*S'*}) = (*x*=*x'* ∧ *e*=*e'* ∧ *S* ≤ *S'*) |
(*c1*;;*c2*) ≤ (*c1'*;;*c2'*) = (*c1* ≤ *c1'* ∧ *c2* ≤ *c2'*) |
(*IF b THEN c1 ELSE c2* {*S*}) ≤ (*IF b' THEN c1' ELSE c2'* {*S'*}) =
  (*b*=*b'* ∧ *c1* ≤ *c1'* ∧ *c2* ≤ *c2'* ∧ *S* ≤ *S'*) |
({*Inv*} *WHILE b DO c* {*P*}) ≤ ({*Inv'*} *WHILE b' DO c'* {*P'*}) =
  (*b*=*b'* ∧ *c* ≤ *c'* ∧ *Inv* ≤ *Inv'* ∧ *P* ≤ *P'*) |
less-eq-acom - - = *False*

lemma *SKIP-le*: *SKIP* {*S*} ≤ *c* ⟷ (∃ *S'*. *c* = *SKIP* {*S'*} ∧ *S* ≤ *S'*)
⟨*proof*⟩

lemma *Assign-le*: *x* ::= *e* {*S*} ≤ *c* ⟷ (∃ *S'*. *c* = *x* ::= *e* {*S'*} ∧ *S* ≤ *S'*)
⟨*proof*⟩

lemma *Seq-le*: *c1*;;*c2* ≤ *c* ⟷ (∃ *c1' c2'*. *c* = *c1'*;;*c2'* ∧ *c1* ≤ *c1'* ∧ *c2* ≤ *c2'*)
⟨*proof*⟩

lemma *If-le*: *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')
⟨*proof*⟩

lemma *While-le*: {*Inv*} *WHILE b DO c* {*P*} ≤ *w* ⟷
  (∃ *Inv' c' P'*. *w* = {*Inv'*} *WHILE b DO c'* {*P'*} ∧ *c* ≤ *c'* ∧ *Inv* ≤ *Inv'* ∧ *P* ≤
P')
⟨*proof*⟩

definition *less-acom* :: *'a acom* ⇒ *'a acom* ⇒ *bool* **where**
less-acom x y = (*x* ≤ *y* ∧ ¬ *y* ≤ *x*)

instance
⟨*proof*⟩

end

fun sub_1 :: *'a acom* ⇒ *'a acom* **where**
sub_1(*c1*;;*c2*) = *c1* |

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

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

fun $invar$ :: $'a\ acom \Rightarrow\ 'a$ **where**
$invar(\{I\}\ WHILE\ b\ DO\ c\ \{P\}) = I$

fun $lift$ :: $('a\ set \Rightarrow\ 'b) \Rightarrow com \Rightarrow\ 'a\ acom\ set \Rightarrow\ 'b\ acom$
where
$lift\ F\ com.SKIP\ M = (SKIP\ \{F\ (post\ `\ M)\})$ |
$lift\ F\ (x ::= a)\ M = (x ::= a\ \{F\ (post\ `\ M)\})$ |
$lift\ F\ (c1;;c2)\ M =$
$lift\ F\ c1\ (sub_1\ `\ M);;\ lift\ F\ c2\ (sub_2\ `\ M)$ |
$lift\ F\ (IF\ b\ THEN\ c1\ ELSE\ c2)\ M =$
$IF\ b\ THEN\ lift\ F\ c1\ (sub_1\ `\ M)\ ELSE\ lift\ F\ c2\ (sub_2\ `\ M)$
$\{F\ (post\ `\ M)\}$ |
$lift\ F\ (WHILE\ b\ DO\ c)\ M =$
$\{F\ (invar\ `\ M)\}$
$WHILE\ b\ DO\ lift\ F\ c\ (sub_1\ `\ M)$
$\{F\ (post\ `\ M)\}$

global-interpretation *Complete-Lattice-ix* $\%c.\ \{c'.\ strip\ c' = c\}\ lift\ Inter$
⟨*proof*⟩

lemma *le-post*: $c \le d \Longrightarrow post\ c \le post\ d$
⟨*proof*⟩

3.2 Collecting semantics

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

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

lemma *mono2-step*: $c1 \le c2 \Longrightarrow S1 \subseteq S2 \Longrightarrow step\ S1\ c1 \le step\ S2\ c2$
⟨*proof*⟩

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

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

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

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

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

end

4 Abstract Interpretation Abstractly

theory *Abs-Int0*
imports
HOL−Library.While-Combinator
Collecting
begin

4.1 Orderings

class *preord* =
fixes *le* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** ‹⊑› *50*)
assumes *le-refl*[*simp*]: $x \sqsubseteq x$
and *le-trans*: $x \sqsubseteq y \Longrightarrow y \sqsubseteq z \Longrightarrow x \sqsubseteq z$
begin

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

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

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

declare *le-trans*[*trans*]

end

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

class *SL-top* = *preord* +

fixes *join* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** ‹⊔› *65*)
fixes *Top* :: $'a$ (‹⊤›)
assumes *join-ge1* [*simp*]: $x \sqsubseteq x \sqcup y$
and *join-ge2* [*simp*]: $y \sqsubseteq x \sqcup y$
and *join-least*: $x \sqsubseteq z \Longrightarrow y \sqsubseteq z \Longrightarrow x \sqcup y \sqsubseteq z$
and *top*[*simp*]: $x \sqsubseteq \top$
begin

lemma *join-le-iff*[*simp*]: $x \sqcup y \sqsubseteq z \longleftrightarrow x \sqsubseteq z \wedge y \sqsubseteq z$
⟨*proof*⟩

lemma *le-join-disj*: $x \sqsubseteq y \vee x \sqsubseteq z \Longrightarrow x \sqsubseteq y \sqcup z$
⟨*proof*⟩

end

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

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

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

instance
⟨*proof*⟩

end

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

fun *le-acom* :: ($'a$::*preord*)*acom* ⇒ $'a$ *acom* ⇒ *bool* **where**
le-acom (*SKIP* {*S*}) (*SKIP* {*S'*}) = ($S \sqsubseteq S'$) |
le-acom (*x* ::= *e* {*S*}) (*x'* ::= *e'* {*S'*}) = (*x*=*x'* ∧ *e*=*e'* ∧ $S \sqsubseteq S'$) |
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'* ∧ $S \sqsubseteq S'$) |
le-acom ({*Inv*} *WHILE b DO c* {*P*}) ({*Inv'*} *WHILE b' DO c'* {*P'*}) =
  (*b*=*b'* ∧ *le-acom c c'* ∧ *Inv* ⊑ *Inv'* ∧ $P \sqsubseteq P'$) |
le-acom - - = *False*

lemma [*simp*]: *SKIP* {*S*} ⊑ *c* ⟷ (∃*S'*. *c* = *SKIP* {*S'*} ∧ $S \sqsubseteq S'$)
⟨*proof*⟩

lemma [*simp*]: *x* ::= *e* {*S*} ⊑ *c* ⟷ (∃*S'*. *c* = *x* ::= *e* {*S'*} ∧ $S \sqsubseteq S'$)

⟨*proof*⟩

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

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

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

instance
⟨*proof*⟩

end

4.1.1 Lifting

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

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

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

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

instance
⟨*proof*⟩

end

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

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

lemma *join-None2*[*simp*]: $x \sqcup None = x$
⟨*proof*⟩

definition $\top = Some\ \top$

instance
⟨*proof*⟩

end

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

lemma *strip-bot-acom*[*simp*]: $strip(\bot_c\ c) = c$
⟨*proof*⟩

lemma *bot-acom*[*rule-format*]: $strip\ c' = c \longrightarrow \bot_c\ c \sqsubseteq c'$
⟨*proof*⟩

4.1.2 Post-fixed point iteration

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

lemma *pfp-pfp*: **assumes** $pfp\ f\ x0 = Some\ x$ **shows** $f\ x \sqsubseteq x$
⟨*proof*⟩

lemma *pfp-least*:
assumes *mono*: $\bigwedge x\ y.\ x \sqsubseteq y \Longrightarrow f\ x \sqsubseteq f\ y$
and $f\ p \sqsubseteq p$ **and** $x0 \sqsubseteq p$ **and** $pfp\ f\ x0 = Some\ x$ **shows** $x \sqsubseteq p$
⟨*proof*⟩

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

lemma *lpfpc-pfp*: $lpfp_c\ f\ c0 = Some\ c \Longrightarrow f\ c \sqsubseteq c$
⟨*proof*⟩

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

lemma *strip-lpfpc*: **assumes** $\bigwedge c.\ strip(f\ c) = strip\ c$ **and** $lpfp_c\ f\ c = Some\ c'$
shows $strip\ c' = c$
⟨*proof*⟩

lemma *lpfpc-least*:
assumes *mono*: $\bigwedge x\ y.\ x \sqsubseteq y \Longrightarrow f\ x \sqsubseteq f\ y$
and *strip* $p = c0$ **and** $f\ p \sqsubseteq p$ **and** *lp*: $lpfp_c\ f\ c0 = Some\ c$ **shows** $c \sqsubseteq p$
⟨*proof*⟩

4.2 Abstract Interpretation

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

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

The interface for abstract values:

locale *Val-abs* =
fixes γ :: $'av::SL\text{-}top \Rightarrow val\ set$
assumes *mono-gamma*: $a \sqsubseteq b \Longrightarrow \gamma\ a \subseteq \gamma\ b$
and *gamma-Top*[*simp*]: $\gamma\ \top = UNIV$
fixes num' :: $val \Rightarrow 'av$
and $plus'$ :: $'av \Rightarrow 'av \Rightarrow 'av$
assumes *gamma-num'*: $n : \gamma(num'\ n)$
and *gamma-plus'*:
$n1 : \gamma\ a1 \Longrightarrow n2 : \gamma\ a2 \Longrightarrow n1{+}n2 : \gamma(plus'\ a1\ a2)$

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

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

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

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

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

lemma *strip-step'*[*simp*]: $strip(step'\ S\ c) = strip\ c$
⟨*proof*⟩

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

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

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

lemma *gamma-f-Top*[*simp*]: $\gamma_f\ Top = UNIV$
⟨*proof*⟩

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

lemma *mono-gamma-f*: $f \sqsubseteq g \Longrightarrow \gamma_f\ f \subseteq \gamma_f\ g$
⟨*proof*⟩

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

lemma *mono-gamma-c*: $ca \sqsubseteq ca' \Longrightarrow \gamma_c\ ca \le \gamma_c\ ca'$
⟨*proof*⟩

Soundness:

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

lemma *in-gamma-update*:
$[\![\ s : \gamma_f\ S;\ i : \gamma\ a\ ]\!] \Longrightarrow s(x := i) : \gamma_f(S(x := a))$
⟨*proof*⟩

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*⟩

lemma *AI-sound*: $AI\ c = Some\ c' \Longrightarrow CS\ c \le \gamma_c\ c'$
⟨*proof*⟩

end

4.2.1 Monotonicity

lemma *mono-post*: $c \sqsubseteq c' \Longrightarrow post\ c \sqsubseteq post\ c'$
⟨*proof*⟩

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′*: $S \sqsubseteq S' \Longrightarrow aval'\ e\ S \sqsubseteq aval'\ e\ S'$
⟨*proof*⟩

lemma *mono-update*: $a \sqsubseteq a' \Longrightarrow S \sqsubseteq S' \Longrightarrow S(x := a) \sqsubseteq S'(x := a')$
⟨*proof*⟩

lemma *mono-step′*: $S \sqsubseteq S' \Longrightarrow c \sqsubseteq c' \Longrightarrow step'\ S\ c \sqsubseteq step'\ S'\ c'$
⟨*proof*⟩

end

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

end

5 Abstract State with Computable Ordering

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

begin

A concrete type of state with computable $\sqsubseteq$:

datatype *′a st* = *FunDom vname* ⇒ *′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)
⟨*proof*⟩

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

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

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

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

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

instance
⟨*proof*⟩

end

lemma *mono-lookup*: $F \sqsubseteq F' \Longrightarrow lookup\ F\ x \sqsubseteq lookup\ F'\ x$
⟨*proof*⟩

lemma *mono-update*: $a \sqsubseteq a' \Longrightarrow S \sqsubseteq S' \Longrightarrow update\ S\ x\ a \sqsubseteq update\ S'\ x\ a'$
⟨*proof*⟩

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

abbreviation γ_f :: $'av$ *st* ⇒ *state set*
where γ_f == γ-*st* 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*
⟨*proof*⟩

lemma *gamma-o-Top*[*simp*]: γ_o *Top* = *UNIV*
⟨*proof*⟩

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

⟨*proof*⟩

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

lemma *mono-gamma-c*: $ca \sqsubseteq ca' \Longrightarrow \gamma_c\ ca \le \gamma_c\ ca'$
⟨*proof*⟩

lemma *in-gamma-option-iff*:
  $x : \gamma\text{-}option\ r\ u \longleftrightarrow (\exists\, u'.\ u = Some\ u' \wedge x : r\ u')$
⟨*proof*⟩

end

end

6 Computable Abstract Interpretation

theory *Abs-Int1*
imports *Abs-State*
begin

Abstract interpretation over type *st* instead of functions.

context *Gamma*
begin

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

lemma *aval′-sound*: $s : \gamma_f\ S \Longrightarrow aval\ a\ s : \gamma(aval'\ a\ S)$
⟨*proof*⟩

end

The for-clause (here and elsewhere) only serves the purpose of fixing the name of the type parameter *′av* which would otherwise be renamed to *′a*.

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

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

in IF b THEN c1′ ELSE c2′ {*post c1* ⊔ *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**
$AI = lpfp_c\ (step'\ \top)$

lemma *strip-step′*[*simp*]: *strip*(*step′ S c*) = *strip c*
⟨*proof*⟩

Soundness:

lemma *in-gamma-update*:
$[\![\ s : \gamma_f\ S;\ i : \gamma\ a\ ]\!] \Longrightarrow s(x := i) : \gamma_f(update\ S\ x\ a)$
⟨*proof*⟩

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*⟩

lemma *AI-sound*: $AI\ c = Some\ c' \Longrightarrow CS\ c \le \gamma_c\ c'$
⟨*proof*⟩

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′*: $S \sqsubseteq S' \Longrightarrow aval'\ e\ S \sqsubseteq aval'\ e\ S'$
⟨*proof*⟩

lemma *mono-update*: $a \sqsubseteq a' \Longrightarrow S \sqsubseteq S' \Longrightarrow update\ S\ x\ a \sqsubseteq update\ S'\ x\ a'$
⟨*proof*⟩

lemma *mono-step′*: $S \sqsubseteq S' \Longrightarrow c \sqsubseteq c' \Longrightarrow step'\ S\ c \sqsubseteq step'\ S'\ c'$
⟨*proof*⟩

end

6.2 Ascending Chain Condition

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

lemma *strict-inv-image*: *strict*(*inv-image r f*) = *inv-image* (*strict r*) *f*
⟨*proof*⟩

lemma *acc-inv-image*:
acc r ⟹ *acc* (*inv-image r f*)
⟨*proof*⟩

ACC for option type:

lemma *acc-option*: **assumes** *acc* {(*x*,*y*::′*a*::*preord*). *x* ⊑ *y*}
shows *acc* {(*x*,*y*::′*a*::*preord option*). *x* ⊑ *y*}
⟨*proof*⟩

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*⟩

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

lemma *listrel-Cons-iff*:
(*x*#*xs*, *y*#*ys*) : *listrel r* ⟷ (*x*,*y*) ∈ *r* ∧ (*xs*,*ys*) ∈ *listrel r*
⟨*proof*⟩

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

lemma *listrel-app-same-size*: *size xs1* = *size ys1* ⟹ *size xs2* = *size ys2* ⟹
(*xs1*@*xs2*, *ys1*@*ys2*) : *listrel r* ⟷
(*xs1*,*ys1*) : *listrel r* ∧ (*xs2*,*ys2*) : *listrel r*
⟨*proof*⟩

lemma *listrel-converse*: *listrel*(*r*^−*1*) = (*listrel r*)^−*1*
⟨*proof*⟩

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

lemma *le-iff-le-annos*: *c1* ⊑ *c2* ⟷
(*annos c1*, *annos c2*) : *listrel*{(*x*,*y*). *x* ⊑ *y*} ∧ *strip c1* = *strip c2*
⟨*proof*⟩

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

$(strict\{(c,c'::'a::preord\ acom).\ c \sqsubseteq c'\})\hat{}{-1} \subseteq$
$(strict(inv\text{-}image\ (listrel\{(a,a'::'a).\ a \sqsubseteq a'\} - \{([],[])\})\ annos))\hat{}{-1}$
⟨*proof*⟩

lemma *acc-acom*: $acc\ \{(a,a'::'a::preord).\ a \sqsubseteq a'\} \Longrightarrow$
$acc\ \{(c,c'::'a\ acom).\ c \sqsubseteq c'\}$
⟨*proof*⟩

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*⟩

lemma *lpfpc-termination*:
fixes $f :: (('a::SL\text{-}top)option\ acom \Rightarrow 'a\ option\ acom)$
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 $\exists c'.\ lpfp_c\ f\ c = Some\ c'$
⟨*proof*⟩

context *Abs-Int-mono*
begin

lemma *AI-Some-measure*:
assumes $(strict\{(x,y::'a).\ x \sqsubseteq y\})\hat{}{-1} <= measure\ m$
and $\forall x\ y::'a.\ x \sqsubseteq y \wedge y \sqsubseteq x \longrightarrow m\ x = m\ y$
shows $\exists c'.\ AI\ c = Some\ c'$
⟨*proof*⟩

end

end

7 Backward Analysis of Expressions

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

instantiation *prod* :: (*preord*,*preord*) *preord*
begin

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

instance
⟨*proof*⟩

end

hide-const *bot*

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

lemma *mono-meet*: $x \sqsubseteq x' \Longrightarrow y \sqsubseteq y' \Longrightarrow x \sqcap y \sqsubseteq x' \sqcap y'$
⟨*proof*⟩

end

locale *Val-abs1-gamma* =
  Gamma **where** $\gamma = \gamma$ **for** γ :: *'av*::*L-top-bot* ⇒ *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*: $x : \gamma\ a1 \Longrightarrow x : \gamma\ a2 \Longrightarrow x : \gamma(a1 \sqcap a2)$
⟨*proof*⟩

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

end

locale *Val-abs1* =
  Val-abs1-gamma **where** $\gamma = \gamma$
   for γ :: *'av*::*L-top-bot* ⇒ *val set* +
fixes *test-num'* :: *val* ⇒ *'av* ⇒ *bool*
and *filter-plus'* :: *'av* ⇒ *'av* ⇒ *'av* ⇒ *'av* ∗ *'av*
and *filter-less'* :: *bool* ⇒ *'av* ⇒ *'av* ⇒ *'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* ⇒ *val set*

begin

lemma *in-gamma-join-UpI*: $s : \gamma_o\ S1 \lor s : \gamma_o\ S2 \Longrightarrow s : \gamma_o(S1 \sqcup S2)$
$\langle proof \rangle$

fun *aval''* :: *aexp* ⇒ *'av st option* ⇒ *'av* **where**
aval'' e None = ⊥ |
aval'' e (*Some sa*) = *aval' e sa*

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

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 ⊥ in the *V*-case is important: ⊥ 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-⊥ values. Otherwise the (pointwise) join of two abstract states, one of which contains ⊥ 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*: $s : \gamma_o\ S \Longrightarrow aval\ e\ s : \gamma\ a \Longrightarrow s : \gamma_o\ (afilter\ e\ a\ S)$
$\langle proof \rangle$

lemma *bfilter-sound*: $s : \gamma_o\ S \Longrightarrow bv = bval\ b\ s \Longrightarrow s : \gamma_o(bfilter\ b\ bv\ S)$
$\langle proof \rangle$

fun *step'* :: *'av st option* ⇒ *'av st option acom* ⇒ *'av st option acom*
 where

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

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

lemma *strip-step'*[*simp*]: $strip(step'\ S\ c) = strip\ c$
⟨*proof*⟩

7.2 Soundness

lemma *in-gamma-update*:
$[\![\ s : \gamma_f\ S;\ i : \gamma\ a\ ]\!] \Longrightarrow s(x := i) : \gamma_f(update\ S\ x\ a)$
⟨*proof*⟩

lemma *step-preserves-le*:
$[\![\ S \subseteq \gamma_o\ S';\ cs \le \gamma_c\ ca\ ]\!] \Longrightarrow step\ S\ cs \le \gamma_c\ (step'\ S'\ ca)$
⟨*proof*⟩

lemma *AI-sound*: $AI\ c = Some\ c' \Longrightarrow CS\ c \le \gamma_c\ c'$
⟨*proof*⟩

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

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

lemma *domo-Top*[*simp*]: $domo\ \top = \{\}$
⟨*proof*⟩

lemma *bot-acom-Com*[*simp*]: $\bot_c\ c \in Com\ X$
⟨*proof*⟩

lemma *post-in-annos*: $post\ c : set(annos\ c)$
⟨*proof*⟩

lemma *domo-join*: $domo\ (S \sqcup T) \subseteq domo\ S \cup domo\ T$
⟨*proof*⟩

lemma *domo-afilter*: $vars\ a \subseteq X \Longrightarrow domo\ S \subseteq X \Longrightarrow domo(afilter\ a\ i\ S) \subseteq X$
⟨*proof*⟩

lemma *domo-bfilter*: $vars\ b \subseteq X \Longrightarrow domo\ S \subseteq X \Longrightarrow domo(bfilter\ b\ bv\ S) \subseteq X$
⟨*proof*⟩

lemma *step′-Com*:
$domo\ S \subseteq X \Longrightarrow vars(strip\ c) \subseteq X \Longrightarrow c : Com\ X \Longrightarrow step'\ S\ c : Com\ X$
⟨*proof*⟩

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′*: $a1 \sqsubseteq b1 \Longrightarrow a2 \sqsubseteq b2 \Longrightarrow r \sqsubseteq r' \Longrightarrow$
$filter\text{-}plus'\ r\ a1\ a2 \sqsubseteq filter\text{-}plus'\ r'\ b1\ b2$
and *mono-filter-less′*: $a1 \sqsubseteq b1 \Longrightarrow a2 \sqsubseteq b2 \Longrightarrow$
$filter\text{-}less'\ bv\ a1\ a2 \sqsubseteq filter\text{-}less'\ bv\ b1\ b2$
begin

lemma *mono-aval′*: $S \sqsubseteq S' \Longrightarrow aval'\ e\ S \sqsubseteq aval'\ e\ S'$
⟨*proof*⟩

lemma *mono-aval′′*: $S \sqsubseteq S' \Longrightarrow aval''\ e\ S \sqsubseteq aval''\ e\ S'$
⟨*proof*⟩

lemma *mono-afilter*: $r \sqsubseteq r' \Longrightarrow S \sqsubseteq S' \Longrightarrow afilter\ e\ r\ S \sqsubseteq afilter\ e\ r'\ S'$
⟨*proof*⟩

lemma *mono-bfilter*: $S \sqsubseteq S' \Longrightarrow bfilter\ b\ r\ S \sqsubseteq bfilter\ b\ r\ S'$
⟨*proof*⟩

lemma *mono-step′*: $S \sqsubseteq S' \Longrightarrow c \sqsubseteq c' \Longrightarrow step'\ S\ c \sqsubseteq step'\ S'\ c'$
⟨*proof*⟩

lemma *mono-step′2*: $mono\ (step'\ S)$
⟨*proof*⟩

end

end

8 Interval Analysis

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

datatype *ivl* = *I int option int option*

definition *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 ⟨*proof*⟩

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*)
⟨*proof*⟩

lemma [*simp*]: *is-empty i* ⟹ *γ-ivl i* = {}
⟨*proof*⟩

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

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 $\bot$ = *empty*

instance
⟨*proof*⟩

end

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

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

instance ⟨*proof*⟩

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*)
⟨*proof*⟩

definition *filter-plus-ivl i i1 i2* = (~~*if is-empty i 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*⟩

global-interpretation *Val-abs1-gamma*
where *γ* = *γ-ivl* **and** *num′* = *num-ivl* **and** *plus′* = *plus-ivl*
defines *aval-ivl* = *aval′*
⟨*proof*⟩

lemma *mono-minus-ivl*:
i1 ⊑ *i1′* ⟹ *i2* ⊑ *i2′* ⟹ *minus-ivl i1 i2* ⊑ *minus-ivl i1′ i2′*

⟨*proof*⟩

global-interpretation *Val-abs1*
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*⟩

global-interpretation *Abs-Int1*
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*
defines *afilter-ivl* = *afilter*
and *bfilter-ivl* = *bfilter*
and *step-ivl* = *step'*
and *AI-ivl* = *AI*
and *aval-ivl'* = *aval''*
⟨*proof*⟩

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*⟩

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*) ($\bot_c$ *test2-ivl*))
value *show-acom* (((*step-ivl* ⊤)^^*1*) ($\bot_c$ *test2-ivl*))
value *show-acom* (((*step-ivl* ⊤)^^*2*) ($\bot_c$ *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*) ($\bot_c$ *test3-ivl*))
value *show-acom* (((*step-ivl* ⊤)^^*1*) ($\bot_c$ *test3-ivl*))
value *show-acom* (((*step-ivl* ⊤)^^*2*) ($\bot_c$ *test3-ivl*))
value *show-acom* (((*step-ivl* ⊤)^^*3*) ($\bot_c$ *test3-ivl*))
value *show-acom* (((*step-ivl* ⊤)^^*4*) ($\bot_c$ *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*) ($\bot_c$ *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* ⊤)^^*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* ⇒ ′*a* ⇒ ′*a* (**infix** ‹∇› *65*)
assumes *widen1*: *x* ⊑ *x* ∇ *y*
assumes *widen2*: *y* ⊑ *x* ∇ *y*
fixes *narrow* :: ′*a* ⇒ ′*a* ⇒ ′*a* (**infix** ‹△› *65*)
assumes *narrow1*: *y* ⊑ *x* ⟹ *y* ⊑ *x* △ *y*
assumes *narrow2*: *y* ⊑ *x* ⟹ *x* △ *y* ⊑ *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*⟩

end

9.2 Abstract State

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

definition *widen-st F1 F2* =
  FunDom (λx. *fun F1 x* ∇ *fun F2 x*) (*inter-list* (*dom F1*) (*dom F2*))

definition *narrow-st F1 F2* =
  FunDom (λx. *fun F1 x* $\triangle$ *fun F2 x*) (*inter-list* (*dom F1*) (*dom F2*))

instance
⟨*proof*⟩

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 $\triangle$ *x* = *None* |
x $\triangle$ *None* = *None* |
(*Some x*) $\triangle$ (*Some y*) = *Some*(*x* $\triangle$ *y*)

instance
⟨*proof*⟩

end

9.4 Annotated commands

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

abbreviation *widen-acom* :: ('*a*::*WN*)*acom* ⇒ '*a acom* ⇒ '*a acom* (**infix** ‹∇_c›
65)

where *widen-acom* == *map2-acom* (∇)

abbreviation *narrow-acom* :: (*'a*::*WN*)*acom* ⇒ *'a acom* ⇒ *'a acom* (**infix** ‹$\triangle_c$› *65*)
where *narrow-acom* == *map2-acom* (△)

lemma *widen1-acom*: *strip c* = *strip c'* ⟹ *c* ⊑ *c* ∇_c *c'*
⟨*proof*⟩

lemma *widen2-acom*: *strip c* = *strip c'* ⟹ *c'* ⊑ *c* ∇_c *c'*
⟨*proof*⟩

lemma *narrow1-acom*: *y* ⊑ *x* ⟹ *y* ⊑ *x* $\triangle_c$ *y*
⟨*proof*⟩

lemma *narrow2-acom*: *y* ⊑ *x* ⟹ *x* $\triangle_c$ *y* ⊑ *x*
⟨*proof*⟩

9.5 Post-fixed point computation

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

definition *iter-narrow* :: (*'a acom* ⇒ *'a acom*) ⇒ *'a acom* ⇒ *'a*::*WN acom option*
where *iter-narrow f* = *while-option* (λ*c*. ¬ *c* ⊑ *c* $\triangle_c$ *f c*) (λ*c*. *c* $\triangle_c$ *f c*)

definition *pfp-wn* ::
((*'a*::*WN*)*option acom* ⇒ *'a option acom*) ⇒ *com* ⇒ *'a option acom option*
where *pfp-wn f c* = (*case iter-widen f* ($\bot_c$ *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*
⟨*proof*⟩

lemma *iter-widen-pfp*: *iter-widen f c* = *Some c'* ⟹ *f c'* ⊑ *c'*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *iter-narrow-pfp*: **assumes** *mono f* **and** *f c0* ⊑ *c0*

and *iter-narrow f c0* = *Some c*
shows $f\ c \sqsubseteq c \wedge c \sqsubseteq c0$ (**is** *?P c*)
⟨*proof*⟩

lemma *pfp-wn-pfp*:
$[\![$ *mono f*; *pfp-wn f c* = *Some c'* $]\!] \Longrightarrow f\ c' \sqsubseteq c'$
⟨*proof*⟩

lemma *strip-pfp-wn*:
$[\![\ \forall c.$ *strip*(*f c*) = *strip c*; *pfp-wn f c* = *Some c'* $]\!] \Longrightarrow$ *strip c'* = *c*
⟨*proof*⟩

locale *Abs-Int2* = *Abs-Int1-mono*
where $\gamma{=}\gamma$ **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* $\leq \gamma_c$ *c'*
⟨*proof*⟩

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*
⟨*proof*⟩

9.6 Tests

definition *step-up-ivl n* = $((\lambda c.\ c\ \nabla_c$ *step-ivl* ⊤ *c*$)$^^*n*)
definition *step-down-ivl n* = $((\lambda c.\ c\ \triangle_c$ *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* ($\bot_c$ *test3-ivl*))
value *show-acom* (*step-up-ivl 2* ($\bot_c$ *test3-ivl*))
value *show-acom* (*step-up-ivl 3* ($\bot_c$ *test3-ivl*))
value *show-acom* (*step-up-ivl 4* ($\bot_c$ *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 $m\text{-}ivl :: ivl \Rightarrow nat$ **where**
$m\text{-}ivl\ ivl = (case\ ivl\ of\ I\ l\ h \Rightarrow$
$\quad (case\ l\ of\ None \Rightarrow 0 \mid Some\ \text{-} \Rightarrow 1) + (case\ h\ of\ None \Rightarrow 0 \mid Some\ \text{-} \Rightarrow 1))$

lemma *m-ivl-height*: $m\text{-}ivl\ ivl \le 2$
⟨*proof*⟩

lemma *m-ivl-anti-mono*: $(y::ivl) \sqsubseteq x \Longrightarrow m\text{-}ivl\ x \le m\text{-}ivl\ y$
⟨*proof*⟩

lemma *m-ivl-widen*:
$\sim y \sqsubseteq x \Longrightarrow m\text{-}ivl(x \nabla y) < m\text{-}ivl\ x$
⟨*proof*⟩

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

definition $n\text{-}ivl :: ivl \Rightarrow nat$ **where**
$n\text{-}ivl\ ivl = 2 - m\text{-}ivl\ ivl$

lemma *n-ivl-mono*: $(x::ivl) \sqsubseteq y \Longrightarrow n\text{-}ivl\ x \le n\text{-}ivl\ y$
⟨*proof*⟩

lemma *n-ivl-narrow*:
$\sim x \sqsubseteq x \triangle y \Longrightarrow n\text{-}ivl(x \triangle y) < n\text{-}ivl\ x$
⟨*proof*⟩

9.8 Termination: Abstract State

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

lemma *m-st-height*: **assumes** *finite X* **and** $set\ (dom\ S) \subseteq X$
shows $m\text{-}st\ m\text{-}ivl\ S \le 2 * card\ X$
⟨*proof*⟩

lemma *m-st-anti-mono*:
$S1 \sqsubseteq S2 \Longrightarrow m\text{-}st\ m\text{-}ivl\ S2 \le m\text{-}st\ m\text{-}ivl\ S1$
⟨*proof*⟩

lemma *m-st-widen*:
assumes $\neg\ S2 \sqsubseteq S1$ **shows** $m\text{-}st\ m\text{-}ivl\ (S1 \nabla S2) < m\text{-}st\ m\text{-}ivl\ S1$
⟨*proof*⟩

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

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

lemma *n-st-narrow*:
assumes *finite X* **and** $set(dom\ S1) \subseteq X$ $set(dom\ S2) \subseteq X$
and $S2 \sqsubseteq S1$ $\neg\ S1 \sqsubseteq S1 \triangle S2$
shows $n\text{-}st\ n\text{-}ivl\ X\ (S1 \triangle S2) < n\text{-}st\ n\text{-}ivl\ X\ S1$
⟨*proof*⟩

9.9 Termination: Option

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

lemma *m-o-anti-mono*: $finite\ X \Longrightarrow domo\ S2 \subseteq X \Longrightarrow S1 \sqsubseteq S2 \Longrightarrow$
$m\text{-}o\ (m\text{-}st\ m\text{-}ivl)\ (2 * card\ X)\ S2 \leq m\text{-}o\ (m\text{-}st\ m\text{-}ivl)\ (2 * card\ X)\ S1$
⟨*proof*⟩

lemma *m-o-widen*: $[\![\ finite\ X;\ domo\ S2 \subseteq X;\ \neg\ S2 \sqsubseteq S1\ ]\!] \Longrightarrow$
$m\text{-}o\ (m\text{-}st\ m\text{-}ivl)\ (2 * card\ X)\ (S1\ \nabla\ S2) < m\text{-}o\ (m\text{-}st\ m\text{-}ivl)\ (2 * card\ X)\ S1$
⟨*proof*⟩

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

lemma *n-o-mono*: $domo\ S1 \subseteq X \Longrightarrow domo\ S2 \subseteq X \Longrightarrow S1 \sqsubseteq S2 \Longrightarrow$
$n\text{-}o\ (n\text{-}st\ n\text{-}ivl\ X)\ S1 \leq n\text{-}o\ (n\text{-}st\ n\text{-}ivl\ X)\ S2$
⟨*proof*⟩

lemma *n-o-narrow*:
$[\![\ finite\ X;\ domo\ S1 \subseteq X;\ domo\ S2 \subseteq X;\ S2 \sqsubseteq S1;\ \neg\ S1 \sqsubseteq S1 \triangle S2\ ]\!]$
$\Longrightarrow n\text{-}o\ (n\text{-}st\ n\text{-}ivl\ X)\ (S1 \triangle S2) < n\text{-}o\ (n\text{-}st\ n\text{-}ivl\ X)\ S1$
⟨*proof*⟩

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

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

9.10 Termination: Commands

lemma *strip-widen-acom*[*simp*]:
$strip\ c' = strip\ (c::'a::WN\ acom) \Longrightarrow\ strip\ (c\ \nabla_c\ c') = strip\ c$
⟨*proof*⟩

lemma *strip-narrow-acom*[*simp*]:
$strip\ c' = strip\ (c::'a::WN\ acom) \Longrightarrow\ strip\ (c \triangle_c c') = strip\ c$
⟨*proof*⟩

lemma *annos-widen-acom*[*simp*]: *strip c1 = strip* (*c2*::'*a*::*WN acom*) $\Longrightarrow$
annos(*c1* ∇_c *c2*) = *map* (%(*x*,*y*).*x*∇*y*) (*zip* (*annos c1*) (*annos*(*c2*::'*a*::*WN acom*)))
⟨*proof*⟩

lemma *annos-narrow-acom*[*simp*]: *strip c1 = strip* (*c2*::'*a*::*WN acom*) $\Longrightarrow$
annos(*c1* $\triangle_c$ *c2*) = *map* (%(*x*,*y*).*x*$\triangle$*y*) (*zip* (*annos c1*) (*annos*(*c2*::'*a*::*WN acom*)))
⟨*proof*⟩

lemma *widen-acom-Com*[*simp*]: *strip c2 = strip c1* $\Longrightarrow$
c1 : *Com X* $\Longrightarrow$ *c2* : *Com X* $\Longrightarrow$ (*c1* ∇_c *c2*) : *Com X*
⟨*proof*⟩

lemma *narrow-acom-Com*[*simp*]: *strip c2 = strip c1* $\Longrightarrow$
c1 : *Com X* $\Longrightarrow$ *c2* : *Com X* $\Longrightarrow$ (*c1* $\triangle_c$ *c2*) : *Com X*
⟨*proof*⟩

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

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

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*)))
⟨*proof*⟩

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'* $\wedge$ ~ *c'* $\sqsubseteq$ *c*}^−*1*)
and *P c0* **and** *c0* $\sqsubseteq$ *f c0* **shows** ∃ *c*. *iter-widen f c0 = Some c*
⟨*proof*⟩

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* $\wedge$ ~ *c* $\sqsubseteq$ *c* $\triangle_c$ *f c*}^−*1*)
and *P c0* **shows** ∃ *c*. *iter-narrow f c0 = Some c*
⟨*proof*⟩

lemma *iter-winden-step-ivl-termination*:
∃ *c*. *iter-widen* (*step-ivl* $\top$) ($\bot_c$ *c0*) = *Some c*
⟨*proof*⟩

lemma *iter-narrow-step-ivl-termination*:

$c0 \in Com\ (vars(strip\ c0)) \Longrightarrow step\text{-}ivl\ \top\ c0 \sqsubseteq c0 \Longrightarrow$
$\exists\, c.\ iter\text{-}narrow\ (step\text{-}ivl\ \top)\ c0 = Some\ c$
$\langle proof \rangle$

lemma *while-Com*:
fixes $c :: {}'a\ st\ option\ acom$
assumes $while\text{-}option\ P\ f\ c = Some\ c'$
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** $c' : Com(X)$
$\langle proof \rangle$

lemma *iter-widen-Com*: **fixes** $f :: {}'a{::}WN\ st\ option\ acom \Rightarrow {}'a\ st\ option\ acom$
assumes $iter\text{-}widen\ f\ c = Some\ c'$
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** $c' : Com(X)$
$\langle proof \rangle$

context *Abs-Int2*
begin

lemma *iter-widen-step′-Com*:
$iter\text{-}widen\ (step'\ \top)\ c = Some\ c' \Longrightarrow vars(strip\ c) \subseteq X \Longrightarrow c : Com(X)$
$\Longrightarrow c' : Com(X)$
$\langle proof \rangle$

end

theorem *AI-ivl′-termination*:
$\exists\, c'.\ AI\text{-}ivl'\ c = Some\ c'$
$\langle proof \rangle$

end

References

[1] T. Nipkow. Abstract interpretation of annotated commands. In Beringer and Felty, editors, *Interactive Theorem Proving (ITP 2012)*, volume 7406 of *LNCS*, pages 116–132. Springer, 2012.

[2] T. Nipkow and G. Klein. *Concrete Semantics with Isabelle/HOL*. Springer, 2014. 298 pp. http://concrete-semantics.org.