Concrete Semantics

TN & GK

February 5, 2012

Contents

1 Arithmetic and Boolean Expressions

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

1.1 Arithmetic Expressions

type-synonym *vname* = *string*
type-synonym *val* = *int*
type-synonym *state* = *vname* ⇒ *val*

datatype *aexp* = *N int* | *V vname* | *Plus aexp aexp*

fun *aval* :: *aexp* ⇒ *state* ⇒ *val* **where**
aval (*N n*) *s* = *n* |
aval (*V x*) *s* = *s x* |
aval (*Plus a1 a2*) *s* = *aval a1 s* + *aval a2 s*

value *aval* (*Plus* (*V* ''*x*'') (*N 5*)) (λ*x*. *if x* = ''*x*'' *then 7 else 0*)

The same state more concisely:

value *aval* (*Plus* (*V* ''*x*'') (*N 5*)) ((λ*x*. *0*) (''*x*'':= *7*))

A little syntax magic to write larger states compactly:

definition *null-state* (<>) **where**
null-state ≡ λ*x*. *0*
syntax
-State :: *updbinds* => 'a (<->)
translations
-State ms => *-Update* <> *ms*

We can now write a series of updates to the function λ*x*. *0* compactly:

lemma <*a* := *Suc 0*, *b* := *2*> = (<> (*a* := *Suc 0*)) (*b* := *2*)
by (*rule refl*)

value *aval* (*Plus* (*V* ''*x*'') (*N 5*)) <''*x*'' := *7*>

Variables that are not mentioned in the state are 0 by default in the <>(*a* := *b*) syntax:

value *aval* (*Plus* (*V* ''*x*'') (*N 5*)) <''*y*'' := *7*>

Note that this <...> syntax works for any function space $\tau_1 \Rightarrow \tau_2$ where τ_2 has a *0*.

1.2 Constant Folding

Evaluate constant subsexpressions:

fun *asimp-const* :: *aexp* ⇒ *aexp* **where**
asimp-const (*N n*) = *N n* |
asimp-const (*V x*) = *V x* |
asimp-const (*Plus* a_1 a_2) =
  (*case* (*asimp-const* a_1, *asimp-const* a_2) *of*
    (*N* n_1, *N* n_2) ⇒ *N*(n_1+n_2) |
    (b_1,b_2) ⇒ *Plus* b_1 b_2)

theorem *aval-asimp-const*:
  aval (*asimp-const a*) *s* = *aval a s*
apply(*induction a*)
apply (*auto split*: *aexp.split*)
done

Now we also eliminate all occurrences 0 in additions. The standard method: optimized versions of the constructors:

fun *plus* :: *aexp* ⇒ *aexp* ⇒ *aexp* **where**
plus (*N* i_1) (*N* i_2) = *N*(i_1+i_2) |
plus (*N i*) *a* = (*if i=0 then a else Plus* (*N i*) *a*) |
plus a (*N i*) = (*if i=0 then a else Plus a* (*N i*)) |
plus a_1 a_2 = *Plus* a_1 a_2

lemma *aval-plus*[*simp*]:
  aval (*plus a1 a2*) *s* = *aval a1 s* + *aval a2 s*
apply(*induction a1 a2 rule*: *plus.induct*)
apply *simp-all*
done

fun *asimp* :: *aexp* ⇒ *aexp* **where**
asimp (*N n*) = *N n* |
asimp (*V x*) = *V x* |
asimp (*Plus* a_1 a_2) = *plus* (*asimp* a_1) (*asimp* a_2)

Note that in *asimp-const* the optimized constructor was inlined. Making it a separate function *AExp.plus* improves modularity of the code and the proofs.

value *asimp* (*Plus* (*Plus* (*N 0*) (*N 0*)) (*Plus* (*V* ″*x*″) (*N 0*)))

theorem *aval-asimp*[*simp*]:
  aval (*asimp a*) *s* = *aval a s*
apply(*induction a*)

apply *simp-all*
done

end

theory *BExp* **imports** *AExp* **begin**

1.3 Boolean Expressions

datatype *bexp* = *Bc bool* | *Not bexp* | *And bexp bexp* | *Less aexp aexp*

fun *bval* :: *bexp* ⇒ *state* ⇒ *bool* **where**
bval (*Bc v*) *s* = *v* |
bval (*Not b*) *s* = (¬ *bval b s*) |
bval (*And* b_1 b_2) *s* = (*bval* b_1 *s* ∧ *bval* b_2 *s*) |
bval (*Less* a_1 a_2) *s* = (*aval* a_1 *s* < *aval* a_2 *s*)

value *bval* (*Less* (*V* ″*x*″) (*Plus* (*N 3*) (*V* ″*y*″)))
<″*x*″ := *3*, ″*y*″ := *1*>

To improve automation:

lemma *bval-And-if*[*simp*]:
bval (*And b1 b2*) *s* = (*if bval b1 s then bval b2 s else False*)
by(*simp*)

declare *bval.simps*(*3*)[*simp del*] — remove the original eqn

1.4 Constant Folding

Optimizing constructors:

fun *less* :: *aexp* ⇒ *aexp* ⇒ *bexp* **where**
less (*N* n_1) (*N* n_2) = *Bc*(n_1 < n_2) |
less a_1 a_2 = *Less* a_1 a_2

lemma [*simp*]: *bval* (*less a1 a2*) *s* = (*aval a1 s* < *aval a2 s*)
apply(*induction a1 a2 rule*: *less.induct*)
apply *simp-all*
done

fun *and* :: *bexp* ⇒ *bexp* ⇒ *bexp* **where**
and (*Bc True*) *b* = *b* |
and b (*Bc True*) = *b* |
and (*Bc False*) *b* = *Bc False* |
and b (*Bc False*) = *Bc False* |

and b_1 b_2 = *And* b_1 b_2

lemma *bval-and*[*simp*]: *bval* (*and b1 b2*) *s* = (*bval b1 s* ∧ *bval b2 s*)
apply(*induction b1 b2 rule*: *and.induct*)
apply *simp-all*
done

fun *not* :: *bexp* ⇒ *bexp* **where**
not (*Bc True*) = *Bc False* |
not (*Bc False*) = *Bc True* |
not b = *Not b*

lemma *bval-not*[*simp*]: *bval* (*not b*) *s* = (¬ *bval b s*)
apply(*induction b rule*: *not.induct*)
apply *simp-all*
done

Now the overall optimizer:

fun *bsimp* :: *bexp* ⇒ *bexp* **where**
bsimp (*Bc v*) = *Bc v* |
bsimp (*Not b*) = *not*(*bsimp b*) |
bsimp (*And* b_1 b_2) = *and* (*bsimp* b_1) (*bsimp* b_2) |
bsimp (*Less* a_1 a_2) = *less* (*asimp* a_1) (*asimp* a_2)

value *bsimp* (*And* (*Less* (*N 0*) (*N 1*)) *b*)

value *bsimp* (*And* (*Less* (*N 1*) (*N 0*)) (*B True*))

theorem *bval* (*bsimp b*) *s* = *bval b s*
apply(*induction b*)
apply *simp-all*
done

end

2 Stack Machine and Compilation

theory *ASM* **imports** *AExp* **begin**

2.1 Stack Machine

datatype *instr* = *LOADI val* | *LOAD vname* | *ADD*

type-synonym *stack* = *val list*

abbreviation *hd2 xs* == *hd*(*tl xs*)
abbreviation *tl2 xs* == *tl*(*tl xs*)

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

fun *exec1* :: *instr* ⇒ *state* ⇒ *stack* ⇒ *stack* **where**
exec1 (*LOADI n*) - *stk* = *n* # *stk* |
exec1 (*LOAD x*) *s stk* = *s*(*x*) # *stk* |
exec1 ADD - *stk* = (*hd2 stk* + *hd stk*) # *tl2 stk*

fun *exec* :: *instr list* ⇒ *state* ⇒ *stack* ⇒ *stack* **where**
exec [] - *stk* = *stk* |
exec (*i*#*is*) *s stk* = *exec is s* (*exec1 i s stk*)

value *exec* [*LOADI 5*, *LOAD* ″*y*″, *ADD*]
<″*x*″ := *42*, ″*y*″ := *43*> [*50*]

lemma *exec-append*[*simp*]:
exec (*is1*@*is2*) *s stk* = *exec is2 s* (*exec is1 s stk*)
apply(*induction is1 arbitrary*: *stk*)
apply (*auto*)
done

2.2 Compilation

fun *comp* :: *aexp* ⇒ *instr list* **where**
comp (*N n*) = [*LOADI n*] |
comp (*V x*) = [*LOAD x*] |
comp (*Plus* e_1 e_2) = *comp* e_1 @ *comp* e_2 @ [*ADD*]

value *comp* (*Plus* (*Plus* (*V* ″*x*″) (*N 1*)) (*V* ″*z*″))

theorem *exec-comp*: *exec* (*comp a*) *s stk* = *aval a s* # *stk*
apply(*induction a arbitrary*: *stk*)
apply (*auto*)
done

end

theory *Star* **imports** *Main*
begin

inductive
star :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *'a* ⇒ *'a* ⇒ *bool*
for *r* **where**
refl: *star r x x* |
step: *r x y* ⟹ *star r y z* ⟹ *star r x z*

hide-fact (**open**) *refl step* — names too generic

lemma *star-trans*:
star r x y ⟹ *star r y z* ⟹ *star r x z*
proof(*induction rule*: *star.induct*)
case *refl* **thus** *?case* **.**
next
case *step* **thus** *?case* **by** (*metis star.step*)
qed

lemmas *star-induct* = *star.induct*[*of r*:: *'a*∗*'b* ⇒ *'a*∗*'b* ⇒ *bool*, *split-format*(*complete*)]

declare *star.refl*[*simp*,*intro*]

lemma *star-step1*[*simp*, *intro*]: *r x y* ⟹ *star r x y*
by(*metis star.refl star.step*)

code-pred *star* **.**

end

3 IMP — A Simple Imperative Language

theory *Com* **imports** *BExp* **begin**

datatype
com = *SKIP*
| *Assign vname aexp* (- ::= - [*1000*, *61*] *61*)
| *Semi com com* (-;/ - [*60*, *61*] *60*)
| *If bexp com com* ((*IF* -/ *THEN* -/ *ELSE* -) [*0*, *0*, *61*] *61*)
| *While bexp com* ((*WHILE* -/ *DO* -) [*0*, *61*] *61*)

end

theory *Big-Step* **imports** *Com* **begin**

3.1 Big-Step Semantics of Commands

inductive
  big-step :: *com* × *state* ⇒ *state* ⇒ *bool* (**infix** ⇒ *55*)
where
Skip: (*SKIP*,*s*) ⇒ *s* |
Assign: (*x* ::= *a*,*s*) ⇒ *s*(*x* := *aval a s*) |
Semi: ⟦ (*c*$_1$,*s*$_1$) ⇒ *s*$_2$; (*c*$_2$,*s*$_2$) ⇒ *s*$_3$ ⟧ ⟹
    (*c*$_1$;*c*$_2$, *s*$_1$) ⇒ *s*$_3$ |

IfTrue: ⟦ *bval b s*; (*c*$_1$,*s*) ⇒ *t* ⟧ ⟹
    (*IF b THEN c*$_1$ *ELSE c*$_2$, *s*) ⇒ *t* |
IfFalse: ⟦ ¬*bval b s*; (*c*$_2$,*s*) ⇒ *t* ⟧ ⟹
    (*IF b THEN c*$_1$ *ELSE c*$_2$, *s*) ⇒ *t* |

WhileFalse: ¬*bval b s* ⟹ (*WHILE b DO c*,*s*) ⇒ *s* |
WhileTrue: ⟦ *bval b s*$_1$; (*c*,*s*$_1$) ⇒ *s*$_2$; (*WHILE b DO c*, *s*$_2$) ⇒ *s*$_3$ ⟧ ⟹
    (*WHILE b DO c*, *s*$_1$) ⇒ *s*$_3$

schematic-lemma *ex*: (''*x*'' ::= *N 5*; ''*y*'' ::= *V* ''*x*'', *s*) ⇒ *?t*
apply(*rule Semi*)
apply(*rule Assign*)
apply *simp*
apply(*rule Assign*)
done

thm *ex*[*simplified*]

We want to execute the big-step rules:

code-pred *big-step* **.**

For inductive definitions we need command `values` instead of `value`.

values {*t*. (*SKIP*, λ-. *0*) ⇒ *t*}

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

values {*map t* [''*x*''] |*t*. (*SKIP*, <''*x*'' := *42*>) ⇒ *t*}

values {*map t* [''*x*''] |*t*. (''*x*'' ::= *N 2*, <''*x*'' := *42*>) ⇒ *t*}

values {*map t* [''*x*'',''*y*''] |*t*.
  (*WHILE Less* (*V* ''*x*'') (*V* ''*y*'') *DO* (''*x*'' ::= *Plus* (*V* ''*x*'') (*N 5*)),
   <''*x*'' := *0*, ''*y*'' := *13*>) ⇒ *t*}

Proof automation:

declare *big-step.intros* [*intro*]

The standard induction rule

$$\begin{array}{l}
[\![x1 \Rightarrow x2;\ \bigwedge s.\ P\ (SKIP,\ s)\ s;\ \bigwedge x\ a\ s.\ P\ (x ::= a,\ s)\ (s(x := aval\ a\ s));\\
\ \bigwedge c_1\ s_1\ s_2\ c_2\ s_3.\\
\quad [\![(c_1,\ s_1) \Rightarrow s_2;\ P\ (c_1,\ s_1)\ s_2;\ (c_2,\ s_2) \Rightarrow s_3;\ P\ (c_2,\ s_2)\ s_3]\!]\\
\quad \Longrightarrow P\ (c_1;\ c_2,\ s_1)\ s_3;\\
\ \bigwedge b\ s\ c_1\ t\ c_2.\\
\quad [\![bval\ b\ s;\ (c_1,\ s) \Rightarrow t;\ P\ (c_1,\ s)\ t]\!] \Longrightarrow P\ (IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s)\\
t;\\
\ \bigwedge b\ s\ c_2\ t\ c_1.\\
\quad [\![\neg\ bval\ b\ s;\ (c_2,\ s) \Rightarrow t;\ P\ (c_2,\ s)\ t]\!] \Longrightarrow P\ (IF\ b\ THEN\ c_1\ ELSE\ c_2,\\
s)\ t;\\
\ \bigwedge b\ s\ c.\ \neg\ bval\ b\ s \Longrightarrow P\ (WHILE\ b\ DO\ c,\ s)\ s;\\
\ \bigwedge b\ s_1\ c\ s_2\ s_3.\\
\quad [\![bval\ b\ s_1;\ (c,\ s_1) \Rightarrow s_2;\ P\ (c,\ s_1)\ s_2;\ (WHILE\ b\ DO\ c,\ s_2) \Rightarrow s_3;\\
\quad P\ (WHILE\ b\ DO\ c,\ s_2)\ s_3]\!]\\
\quad \Longrightarrow P\ (WHILE\ b\ DO\ c,\ s_1)\ s_3]\!]\\
\Longrightarrow P\ x1\ x2
\end{array}$$

thm *big-step.induct*

A customized induction rule for (c,s) pairs:

lemmas *big-step-induct* = *big-step.induct*[*split-format*(*complete*)]
thm *big-step-induct*

$$\begin{array}{l}
[\![(x1a,\ x1b) \Rightarrow x2a;\ \bigwedge s.\ P\ SKIP\ s\ s;\ \bigwedge x\ a\ s.\ P\ (x ::= a)\ s\ (s(x := aval\ a\\
s));\\
\ \bigwedge c_1\ s_1\ s_2\ c_2\ s_3.\\
\quad [\![(c_1,\ s_1) \Rightarrow s_2;\ P\ c_1\ s_1\ s_2;\ (c_2,\ s_2) \Rightarrow s_3;\ P\ c_2\ s_2\ s_3]\!]\\
\quad \Longrightarrow P\ (c_1;\ c_2)\ s_1\ s_3;\\
\ \bigwedge b\ s\ c_1\ t\ c_2.\\
\quad [\![bval\ b\ s;\ (c_1,\ s) \Rightarrow t;\ P\ c_1\ s\ t]\!] \Longrightarrow P\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ s\ t;\\
\ \bigwedge b\ s\ c_2\ t\ c_1.\\
\quad [\![\neg\ bval\ b\ s;\ (c_2,\ s) \Rightarrow t;\ P\ c_2\ s\ t]\!] \Longrightarrow P\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ s\ t;\\
\ \bigwedge b\ s\ c.\ \neg\ bval\ b\ s \Longrightarrow P\ (WHILE\ b\ DO\ c)\ s\ s;\\
\ \bigwedge b\ s_1\ c\ s_2\ s_3.\\
\quad [\![bval\ b\ s_1;\ (c,\ s_1) \Rightarrow s_2;\ P\ c\ s_1\ s_2;\ (WHILE\ b\ DO\ c,\ s_2) \Rightarrow s_3;\\
\quad P\ (WHILE\ b\ DO\ c)\ s_2\ s_3]\!]\\
\quad \Longrightarrow P\ (WHILE\ b\ DO\ c)\ s_1\ s_3]\!]\\
\Longrightarrow P\ x1a\ x1b\ x2a
\end{array}$$

3.2 Rule inversion

What can we deduce from $(SKIP,\ s) \Rightarrow t$? That $s = t$. This is how we can automatically prove it:

inductive-cases *skipE*[*elim*!]: $(SKIP,s) \Rightarrow t$
thm *skipE*

This is an *elimination rule*. The [elim] attribute tells auto, blast and friends (but not simp!) to use it automatically; [elim!] means that it is applied eagerly.

Similarly for the other commands:

inductive-cases *AssignE*[*elim*!]: $(x ::= a,s) \Rightarrow t$
thm *AssignE*
inductive-cases *SemiE*[*elim*!]: $(c1;c2,s1) \Rightarrow s3$
thm *SemiE*
inductive-cases *IfE*[*elim*!]: $(IF\ b\ THEN\ c1\ ELSE\ c2,s) \Rightarrow t$
thm *IfE*

inductive-cases *WhileE*[*elim*]: $(WHILE\ b\ DO\ c,s) \Rightarrow t$
thm *WhileE*

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

An automatic example:

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

Rule inversion by hand via the "cases" method:

```
lemma assumes (IF b THEN SKIP ELSE SKIP, s) ⇒ t
shows t = s
proof−
  from assms show ?thesis
  proof cases  — inverting assms
    case IfTrue thm IfTrue
    thus ?thesis by blast
  next
    case IfFalse thus ?thesis by blast
  qed
qed
```

```
lemma assign-simp:
  (x ::= a,s) ⇒ s' ⟷ (s' = s(x := aval a s))
  by auto
```

3.3 Command Equivalence

We call two statements c and c' equivalent wrt. the big-step semantics when *c started in s terminates in s′ iff c′ started in the same s also terminates in the same s′*. Formally:

abbreviation
equiv-c :: *com* ⇒ *com* ⇒ *bool* (**infix** ∼ *50*) **where**
$c \sim c' == (\forall s\ t.\ (c,s) \Rightarrow t\ =\ (c',s) \Rightarrow t)$

Warning: ∼ is the symbol written `\ < s i m >` (without spaces).

As an example, we show that loop unfolding is an equivalence transformation on programs:

lemma *unfold-while*:
(*WHILE b DO c*) ∼ (*IF b THEN c*; *WHILE b DO c ELSE SKIP*) (**is** *?w* ∼ *?iw*)
proof −
— to show the equivalence, we look at the derivation tree for
— each side and from that construct a derivation tree for the other side
{ fix *s t* **assume** (*?w*, *s*) ⇒ *t*
— as a first thing we note that, if *b* is *False* in state *s*,
— then both statements do nothing:
{ assume ¬*bval b s*
hence *t* = *s* **using** ‹(*?w*,*s*) ⇒ *t*› **by** *blast*
hence (*?iw*, *s*) ⇒ *t* **using** ‹¬*bval b s*› **by** *blast*
}
moreover
— on the other hand, if *b* is *True* in state *s*,
— then only the *WhileTrue* rule can have been used to derive (*?w*, *s*) ⇒ *t*
{ assume *bval b s*
with ‹(*?w*, *s*) ⇒ *t*› **obtain** *s′* **where**
(*c*, *s*) ⇒ *s′* **and** (*?w*, *s′*) ⇒ *t* **by** *auto*
— now we can build a derivation tree for the *IF*
— first, the body of the True-branch:
hence (*c*; *?w*, *s*) ⇒ *t* **by** (*rule Semi*)
— then the whole *IF*
with ‹*bval b s*› **have** (*?iw*, *s*) ⇒ *t* **by** (*rule IfTrue*)
}
ultimately
— both cases together give us what we want:
have (*?iw*, *s*) ⇒ *t* **by** *blast*
}
moreover
— now the other direction:

```
{ fix s t assume (?iw, s) ⇒ t
  — again, if b is False in state s, then the False-branch
  — of the IF is executed, and both statements do nothing:
  { assume ¬bval b s
    hence s = t using ⟨(?iw, s) ⇒ t⟩ by blast
    hence (?w, s) ⇒ t using ⟨¬bval b s⟩ by blast
  }
  moreover
  — on the other hand, if b is True in state s,
  — then this time only the IfTrue rule can have be used
  { assume bval b s
    with ⟨(?iw, s) ⇒ t⟩ have (c; ?w, s) ⇒ t by auto
    — and for this, only the Semi-rule is applicable:
    then obtain s' where
      (c, s) ⇒ s' and (?w, s') ⇒ t by auto
   — with this information, we can build a derivation tree for the WHILE

    with ⟨bval b s⟩
    have (?w, s) ⇒ t by (rule WhileTrue)
  }
  ultimately
  — both cases together again give us what we want:
  have (?w, s) ⇒ t by blast
}
ultimately
show ?thesis by blast
qed
```

Luckily, such lengthy proofs are seldom necessary. Isabelle can prove many such facts automatically.

```
lemma while-unfold:
  (WHILE b DO c) ∼ (IF b THEN c; WHILE b DO c ELSE SKIP)
by blast

lemma triv-if:
  (IF b THEN c ELSE c) ∼ c
by blast

lemma commute-if:
  (IF b1 THEN (IF b2 THEN c11 ELSE c12) ELSE c2)
   ∼
   (IF b2 THEN (IF b1 THEN c11 ELSE c2) ELSE (IF b1 THEN c12
ELSE c2))
by blast
```

3.4 Execution is deterministic

This proof is automatic.

theorem *big-step-determ*: $[\![(c,s) \Rightarrow t; (c,s) \Rightarrow u]\!] \Longrightarrow u = t$
by (*induction arbitrary*: *u rule*: *big-step.induct*) *blast+*

This is the proof as you might present it in a lecture. The remaining cases are simple enough to be proved automatically:

theorem
$(c,s) \Rightarrow t \implies (c,s) \Rightarrow t' \implies t' = t$
proof (*induction arbitrary*: t' *rule*: *big-step.induct*)
— the only interesting case, *WhileTrue*:
fix *b c s s1 t* t'
— The assumptions of the rule:
assume *bval b s* **and** $(c,s) \Rightarrow s1$ **and** $(WHILE\ b\ DO\ c,s1) \Rightarrow t$
— Ind.Hyp; note the $\bigwedge$ because of arbitrary:
assume *IHc*: $\bigwedge t'.\ (c,s) \Rightarrow t' \Longrightarrow t' = s1$
assume *IHw*: $\bigwedge t'.\ (WHILE\ b\ DO\ c,s1) \Rightarrow t' \Longrightarrow t' = t$
— Premise of implication:
assume $(WHILE\ b\ DO\ c,s) \Rightarrow t'$
with ‹*bval b s*› **obtain** $s1'$ **where**
c: $(c,s) \Rightarrow s1'$ **and**
w: $(WHILE\ b\ DO\ c,s1') \Rightarrow t'$
by *auto*
from *c IHc* **have** $s1' = s1$ **by** *blast*
with *w IHw* **show** $t' = t$ **by** *blast*
qed *blast+* — prove the rest automatically

end

4 Small-Step Semantics of Commands

theory *Small-Step* **imports** *Star Big-Step* **begin**

4.1 The transition relation

inductive
small-step :: *com* $*$ *state* $\Rightarrow$ *com* $*$ *state* $\Rightarrow$ *bool* (**infix** $\rightarrow$ *55*)
where
Assign: $(x ::= a,\ s) \rightarrow (SKIP,\ s(x := aval\ a\ s))$ |

Semi1: $(SKIP;c_2,s) \rightarrow (c_2,s)$ |
Semi2: $(c_1,s) \rightarrow (c_1',s') \Longrightarrow (c_1;c_2,s) \rightarrow (c_1';c_2,s')$ |

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

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

abbreviation *small-steps* :: $com * state \Rightarrow com * state \Rightarrow bool$ (**infix** $\rightarrow*$ *55*)
where $x \rightarrow* y == star\ small\text{-}step\ x\ y$

4.2 Executability

code-pred *small-step* **.**

values $\{(c',map\ t\ [''x'',''y'',''z'']) \mid c'\ t.$
$(''x'' ::= V\ ''z''; ''y'' ::= V\ ''x'',$
$<''x'' := 3,\ ''y'' := 7,\ ''z'' := 5>) \rightarrow* (c',t)\}$

4.3 Proof infrastructure

4.3.1 Induction rules

The default induction rule *small-step.induct* only works for lemmas of the form $a \rightarrow b \Longrightarrow \ldots$ where a and b are not already pairs $(DUMMY,DUMMY)$. We can generate a suitable variant of *small-step.induct* for pairs by "splitting" the arguments $\rightarrow$ into pairs:

lemmas *small-step-induct* = *small-step.induct*[*split-format*(*complete*)]

4.3.2 Proof automation

declare *small-step.intros*[*simp*,*intro*]

Rule inversion:

inductive-cases *SkipE*[*elim*!]: $(SKIP,s) \rightarrow ct$
thm *SkipE*
inductive-cases *AssignE*[*elim*!]: $(x::=a,s) \rightarrow ct$
thm *AssignE*
inductive-cases *SemiE*[*elim*]: $(c1;c2,s) \rightarrow ct$
thm *SemiE*
inductive-cases *IfE*[*elim*!]: $(IF\ b\ THEN\ c1\ ELSE\ c2,s) \rightarrow ct$
inductive-cases *WhileE*[*elim*]: $(WHILE\ b\ DO\ c,\ s) \rightarrow ct$

A simple property:

lemma *deterministic*:
cs $\rightarrow$ *cs'* $\Longrightarrow$ *cs* $\rightarrow$ *cs''* $\Longrightarrow$ *cs''* = *cs'*
apply(*induction arbitrary*: *cs''* *rule*: *small-step.induct*)
apply *blast+*
done

4.4 Equivalence with big-step semantics

lemma *star-semi2*: (*c1*,*s*) $\rightarrow*$ (*c1'*,*s'*) $\Longrightarrow$ (*c1*;*c2*,*s*) $\rightarrow*$ (*c1'*;*c2*,*s'*)
proof(*induction rule*: *star-induct*)
case *refl* **thus** *?case* **by** *simp*
next
case *step*
thus *?case* **by** (*metis Semi2 star.step*)
qed

lemma *semi-comp*:
$[\![$ (*c1*,*s1*) $\rightarrow*$ (*SKIP*,*s2*); (*c2*,*s2*) $\rightarrow*$ (*SKIP*,*s3*) $]\!]$
$\Longrightarrow$ (*c1*;*c2*, *s1*) $\rightarrow*$ (*SKIP*,*s3*)
by(*blast intro*: *star.step star-semi2 star-trans*)

The following proof corresponds to one on the board where one would show chains of $\rightarrow$ and $\rightarrow*$ steps.

lemma *big-to-small*:
cs $\Rightarrow$ *t* $\Longrightarrow$ *cs* $\rightarrow*$ (*SKIP*,*t*)
proof (*induction rule*: *big-step.induct*)
fix *s* **show** (*SKIP*,*s*) $\rightarrow*$ (*SKIP*,*s*) **by** *simp*
next
fix *x a s* **show** (*x* ::= *a*,*s*) $\rightarrow*$ (*SKIP*, *s*(*x* := *aval a s*)) **by** *auto*
next
fix *c1 c2 s1 s2 s3*
assume (*c1*,*s1*) $\rightarrow*$ (*SKIP*,*s2*) **and** (*c2*,*s2*) $\rightarrow*$ (*SKIP*,*s3*)
thus (*c1*;*c2*, *s1*) $\rightarrow*$ (*SKIP*,*s3*) **by** (*rule semi-comp*)
next
fix *s*::*state* **and** *b c0 c1 t*
assume *bval b s*
hence (*IF b THEN c0 ELSE c1*,*s*) $\rightarrow$ (*c0*,*s*) **by** *simp*
moreover assume (*c0*,*s*) $\rightarrow*$ (*SKIP*,*t*)
ultimately
show (*IF b THEN c0 ELSE c1*,*s*) $\rightarrow*$ (*SKIP*,*t*) **by** (*metis star.simps*)
next
fix *s*::*state* **and** *b c0 c1 t*
assume $\neg$*bval b s*
hence (*IF b THEN c0 ELSE c1*,*s*) $\rightarrow$ (*c1*,*s*) **by** *simp*

moreover assume (*c1*,*s*) $\rightarrow*$ (*SKIP*,*t*)
ultimately
show (*IF b THEN c0 ELSE c1*,*s*) $\rightarrow*$ (*SKIP*,*t*) **by** (*metis star.simps*)
next
fix *b c* **and** *s*::*state*
assume *b*: ¬*bval b s*
let *?if* = *IF b THEN c*; *WHILE b DO c ELSE SKIP*
have (*WHILE b DO c*,*s*) → (*?if*, *s*) **by** *blast*
moreover have (*?if*,*s*) → (*SKIP*, *s*) **by** (*simp add*: *b*)
ultimately show (*WHILE b DO c*,*s*) $\rightarrow*$ (*SKIP*,*s*) **by**(*metis star.refl star.step*)
next
fix *b c s s′ t*
let *?w* = *WHILE b DO c*
let *?if* = *IF b THEN c*; *?w ELSE SKIP*
assume *w*: (*?w*,*s′*) $\rightarrow*$ (*SKIP*,*t*)
assume *c*: (*c*,*s*) $\rightarrow*$ (*SKIP*,*s′*)
assume *b*: *bval b s*
have (*?w*,*s*) → (*?if*, *s*) **by** *blast*
moreover have (*?if*, *s*) → (*c*; *?w*, *s*) **by** (*simp add*: *b*)
moreover have (*c*; *?w*,*s*) $\rightarrow*$ (*SKIP*,*t*) **by**(*rule semi-comp*[*OF c w*])
ultimately show (*WHILE b DO c*,*s*) $\rightarrow*$ (*SKIP*,*t*) **by** (*metis star.simps*)
qed

Each case of the induction can be proved automatically:

lemma *cs* ⇒ *t* ⟹ *cs* $\rightarrow*$ (*SKIP*,*t*)
proof (*induction rule*: *big-step.induct*)
case *Skip* **show** *?case* **by** *blast*
next
case *Assign* **show** *?case* **by** *blast*
next
case *Semi* **thus** *?case* **by** (*blast intro*: *semi-comp*)
next
case *IfTrue* **thus** *?case* **by** (*blast intro*: *star.step*)
next
case *IfFalse* **thus** *?case* **by** (*blast intro*: *star.step*)
next
case *WhileFalse* **thus** *?case*
by (*metis star.step star-step1 small-step.IfFalse small-step.While*)
next
case *WhileTrue*
thus *?case*
by(*metis While semi-comp small-step.IfTrue star.step*[*of small-step*])
qed

lemma *small1-big-continue*:
$cs \rightarrow cs' \Longrightarrow cs' \Rightarrow t \Longrightarrow cs \Rightarrow t$
apply (*induction arbitrary*: *t rule*: *small-step.induct*)
apply *auto*
done

lemma *small-big-continue*:
$cs \rightarrow* cs' \Longrightarrow cs' \Rightarrow t \Longrightarrow cs \Rightarrow t$
apply (*induction rule*: *star.induct*)
apply (*auto intro*: *small1-big-continue*)
done

lemma *small-to-big*: $cs \rightarrow* (SKIP,t) \Longrightarrow cs \Rightarrow t$
by (*metis small-big-continue Skip*)

Finally, the equivalence theorem:

theorem *big-iff-small*:
$cs \Rightarrow t = cs \rightarrow* (SKIP,t)$
by(*metis big-to-small small-to-big*)

4.5 Final configurations and infinite reductions

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

lemma *finalD*: *final* $(c,s) \Longrightarrow c = SKIP$
apply(*simp add*: *final-def*)
apply(*induction c*)
apply *blast+*
done

lemma *final-iff-SKIP*: *final* $(c,s) = (c = SKIP)$
by (*metis SkipE finalD final-def*)

Now we can show that $\Rightarrow$ yields a final state iff $\rightarrow$ terminates:

lemma *big-iff-small-termination*:
$(EX\ t.\ cs \Rightarrow t) \longleftrightarrow (EX\ cs'.\ cs \rightarrow* cs' \wedge final\ cs')$
by(*simp add*: *big-iff-small final-iff-SKIP*)

This is the same as saying that the absence of a big step result is equivalent with absence of a terminating small step sequence, i.e. with nontermination. Since $\rightarrow$ is determininistic, there is no difference between may and must terminate.

end

5 A Compiler for IMP

theory *Compiler* **imports** *Big-Step*
begin

5.1 List setup

We are going to define a small machine language where programs are lists of instructions. For nicer algebraic properties in our lemmas later, we prefer *int* to *nat* as program counter.

Therefore, we define notation for size and indexing for lists on *int*:

abbreviation *isize xs* == *int* (*length xs*)

fun *inth* :: *'a list* ⇒ *int* ⇒ *'a* (**infixl** !! *100*) **where**
(*x* # *xs*) !! *n* = (*if n* = *0 then x else xs* !! (*n* − *1*))

The only additional lemma we need is indexing over append:

lemma *inth-append* [*simp*]:
0 ≤ *n* ⟹
(*xs* @ *ys*) !! *n* = (*if n* < *isize xs then xs* !! *n else ys* !! (*n* − *isize xs*))
by (*induction xs arbitrary*: *n*) (*auto simp*: *algebra-simps*)

5.2 Instructions and Stack Machine

datatype *instr* =
LOADI int |
LOAD vname |
ADD |
STORE vname |
JMP int |
JMPLESS int |
JMPGE int

type-synonym *stack* = *val list*
type-synonym *config* = *int* × *state* × *stack*

abbreviation *hd2 xs* == *hd*(*tl xs*)
abbreviation *tl2 xs* == *tl*(*tl xs*)

inductive *iexec1* :: *instr* ⇒ *config* ⇒ *config* ⇒ *bool*
((-/ ⊢*i* (- →/ -)) [*59*,*0*,*59*] *60*)
where
LOADI n ⊢*i* (*i*,*s*,*stk*) → (*i*+*1*,*s*, *n*#*stk*) |
LOAD x ⊢*i* (*i*,*s*,*stk*) → (*i*+*1*,*s*, *s x* # *stk*) |

ADD $\vdash i\ (i,s,stk) \to (i+1,s,\ (hd2\ stk\ +\ hd\ stk)\ \#\ tl2\ stk)$ |
STORE x $\vdash i\ (i,s,stk) \to (i+1,s(x := hd\ stk),tl\ stk)$ |
JMP n $\vdash i\ (i,s,stk) \to (i+1+n,s,stk)$ |
JMPLESS n $\vdash i\ (i,s,stk) \to (if\ hd2\ stk < hd\ stk\ then\ i+1+n\ else\ i+1,s,tl2\ stk)$ |
JMPGE n $\vdash i\ (i,s,stk) \to (if\ hd2\ stk >= hd\ stk\ then\ i+1+n\ else\ i+1,s,tl2\ stk)$

code-pred *iexec1* **.**

declare *iexec1.intros*

definition
exec1 :: *instr list* ⇒ *config* ⇒ *config* ⇒ *bool* ((-/ ⊢ (- →/ -)) [59,0,59] 60)
where
$P \vdash c \to c' =$
$(\exists\, i\ s\ stk.\ c = (i,s,stk) \wedge P!!i \vdash i\ (i,s,stk) \to c' \wedge 0 \le i \wedge i < isize\ P)$

declare *exec1-def* [*simp*]

lemma *exec1I* [*intro*, *code-pred-intro*]:
assumes $P!!i \vdash i\ (i,s,stk) \to c'\ 0 \le i\ i < isize\ P$
shows $P \vdash (i,s,stk) \to c'$
using *assms* **by** *simp*

inductive *exec* :: *instr list* ⇒ *config* ⇒ *config* ⇒ *bool* (-/ ⊢ (- →*/ -) 50)
where
refl: $P \vdash c \to* c$ |
step: $P \vdash c \to c' \Longrightarrow P \vdash c' \to* c'' \Longrightarrow P \vdash c \to* c''$

declare *refl*[*intro*] *step*[*intro*]

lemmas *exec-induct* = *exec.induct*[*split-format*(*complete*)]

code-pred *exec* **by** *force*

values
{(*i*,*map t* [''*x*'',''*y*'']*,stk*) | *i t stk*.
[*LOAD* ''*y*'', *STORE* ''*x*''] ⊢
(*0*, <''*x*'' := *3*, ''*y*'' := *4*>, []) →* (*i*,*t*,*stk*)}

5.3 Verification infrastructure

lemma *exec-trans*: $P \vdash c \to* c' \Longrightarrow P \vdash c' \to* c'' \Longrightarrow P \vdash c \to* c''$
by (*induction rule*: *exec.induct*) *fastforce*+

inductive-cases *iexec1-cases* [*elim*!]:
$LOADI\ n \vdash i\ c \to c'$
$LOAD\ x \vdash i\ c \to c'$
$ADD \vdash i\ c \to c'$
$STORE\ n \vdash i\ c \to c'$
$JMP\ n \vdash i\ c \to c'$
$JMPLESS\ n \vdash i\ c \to c'$
$JMPGE\ n \vdash i\ c \to c'$

Simplification rules for *iexec1*.

lemma *iexec1-simps* [*simp*]:
$LOADI\ n \vdash i\ c \to c' = (\exists\, i\ s\ stk.\ c = (i,\ s,\ stk) \wedge c' = (i + 1,\ s,\ n \#\ stk))$
$LOAD\ x \vdash i\ c \to c' = (\exists\, i\ s\ stk.\ c = (i,\ s,\ stk) \wedge c' = (i + 1,\ s,\ s\ x \#\ stk))$
$ADD \vdash i\ c \to c' =$
$(\exists\, i\ s\ stk.\ c = (i,\ s,\ stk) \wedge c' = (i + 1,\ s,\ (hd2\ stk + hd\ stk) \#\ tl2\ stk))$
$STORE\ x \vdash i\ c \to c' =$
$(\exists\, i\ s\ stk.\ c = (i,\ s,\ stk) \wedge c' = (i + 1,\ s(x := hd\ stk),\ tl\ stk))$
$JMP\ n \vdash i\ c \to c' = (\exists\, i\ s\ stk.\ c = (i,\ s,\ stk) \wedge c' = (i + 1 + n,\ s,\ stk))$
$JMPLESS\ n \vdash i\ c \to c' =$
$(\exists\, i\ s\ stk.\ c = (i,\ s,\ stk) \wedge$
$\quad c' = (if\ hd2\ stk < hd\ stk\ then\ i + 1 + n\ else\ i + 1,\ s,\ tl2\ stk))$
$JMPGE\ n \vdash i\ c \to c' =$
$(\exists\, i\ s\ stk.\ c = (i,\ s,\ stk) \wedge$
$\quad c' = (if\ hd\ stk \leq hd2\ stk\ then\ i + 1 + n\ else\ i + 1,\ s,\ tl2\ stk))$
by (*auto split del*: *split-if intro*!: *iexec1.intros*)

Below we need to argue about the execution of code that is embedded in larger programs. For this purpose we show that execution is preserved by appending code to the left or right of a program.

lemma *exec1-appendR*: $P \vdash c \to c' \Longrightarrow P@P' \vdash c \to c'$
by *auto*

lemma *exec-appendR*: $P \vdash c \to* c' \Longrightarrow P@P' \vdash c \to* c'$
by (*induction rule*: *exec.induct*) (*fastforce intro*: *exec1-appendR*)+

lemma *iexec1-shiftI*:
assumes $X \vdash i\ (i,s,stk) \to (i',s',stk')$
shows $X \vdash i\ (n+i,s,stk) \to (n+i',s',stk')$

using *assms* **by** *cases auto*

lemma *iexec1-shiftD*:
assumes $X \vdash i\ (n+i,s,stk) \to (n+i',s',stk')$
shows $X \vdash i\ (i,s,stk) \to (i',s',stk')$
using *assms* **by** *cases auto*

lemma *iexec-shift* [*simp*]:
$(X \vdash i\ (n+i,s,stk) \to (n+i',s',stk')) = (X \vdash i\ (i,s,stk) \to (i',s',stk'))$
by (*blast intro*: *iexec1-shiftI dest*: *iexec1-shiftD*)

lemma *exec1-appendL*:
$P \vdash (i,s,stk) \to (i',s',stk') \Longrightarrow$
$P' @ P \vdash (isize(P')+i,s,stk) \to (isize(P')+i',s',stk')$
by *simp*

lemma *exec-appendL*:
$P \vdash (i,s,stk) \to* (i',s',stk') \Longrightarrow$
$P' @ P \vdash (isize(P')+i,s,stk) \to* (isize(P')+i',s',stk')$
by (*induction rule*: *exec-induct*) (*blast intro*!: *exec1-appendL*)+

Now we specialise the above lemmas to enable automatic proofs of $P \vdash c \to* c'$ where P is a mixture of concrete instructions and pieces of code that we already know how they execute (by induction), combined by @ and #. Backward jumps are not supported. The details should be skipped on a first reading.

If we have just executed the first instruction of the program, drop it:

lemma *exec-Cons-1* [*intro*]:
$P \vdash (0,s,stk) \to* (j,t,stk') \Longrightarrow$
$instr\#P \vdash (1,s,stk) \to* (1+j,t,stk')$
by (*drule exec-appendL*[**where** P'=[*instr*]]) *simp*

lemma *exec-appendL-if* [*intro*]:
$isize\ P' <= i$
$\Longrightarrow P \vdash (i - isize\ P',s,stk) \to* (i',s',stk')$
$\Longrightarrow P' @ P \vdash (i,s,stk) \to* (isize\ P' + i',s',stk')$
by (*drule exec-appendL*[**where** P'=P'']) *simp*

Split the execution of a compound program up into the excution of its parts:

lemma *exec-append-trans*[*intro*]:
$P \vdash (0,s,stk) \to* (i',s',stk') \Longrightarrow$
$isize\ P \leq i' \Longrightarrow$
$P' \vdash\ (i' - isize\ P,s',stk') \to* (i'',s'',stk'') \Longrightarrow$

$j'' = isize\ P + i''$
$\Longrightarrow$
$P\ @\ P' \vdash (0,s,stk) \rightarrow* (j'',s'',stk'')$
by(*metis exec-trans*[*OF exec-appendR exec-appendL-if*])

declare *Let-def*[*simp*]

5.4 Compilation

fun *acomp* :: *aexp* $\Rightarrow$ *instr list* **where**
acomp (*N n*) = [*LOADI n*] |
acomp (*V x*) = [*LOAD x*] |
acomp (*Plus a1 a2*) = *acomp a1* @ *acomp a2* @ [*ADD*]

lemma *acomp-correct*[*intro*]:
acomp a $\vdash$ (*0*,*s*,*stk*) $\rightarrow*$ (*isize*(*acomp a*),*s*,*aval a s*#*stk*)
by (*induction a arbitrary*: *stk*) *fastforce*+

fun *bcomp* :: *bexp* $\Rightarrow$ *bool* $\Rightarrow$ *int* $\Rightarrow$ *instr list* **where**
bcomp (*Bc v*) *c n* = (*if v*=*c then* [*JMP n*] *else* []) |
bcomp (*Not b*) *c n* = *bcomp b* ($\neg$*c*) *n* |
bcomp (*And b1 b2*) *c n* =
(*let cb2* = *bcomp b2 c n*;
m = (*if c then isize cb2 else isize cb2*+*n*);
cb1 = *bcomp b1 False m*
in cb1 @ *cb2*) |
bcomp (*Less a1 a2*) *c n* =
acomp a1 @ *acomp a2* @ (*if c then* [*JMPLESS n*] *else* [*JMPGE n*])

value
bcomp (*And* (*Less* (*V* ''*x*'') (*V* ''*y*'')) (*Not*(*Less* (*V* ''*u*'') (*V* ''*v*'')))) *False 3*

lemma *bcomp-correct*[*intro*]:
0 $\leq$ *n* $\Longrightarrow$
bcomp b c n $\vdash$
(*0*,*s*,*stk*) $\rightarrow*$ (*isize*(*bcomp b c n*) + (*if c* = *bval b s then n else 0*),*s*,*stk*)
proof(*induction b arbitrary*: *c n*)
case *Not*
from *Not*(*1*)[**where** *c*=$^\sim$*c*] *Not*(*2*) **show** *?case* **by** *fastforce*
next
case (*And b1 b2*)
from *And*(*1*)[*of if c then isize* (*bcomp b2 c n*) *else isize* (*bcomp b2 c n*)

```
+ n
                False]
      And(2)[of n  c] And(3)
  show ?case by fastforce
qed fastforce+

fun ccomp :: com ⇒ instr list where
ccomp SKIP = [] |
ccomp (x ::= a) = acomp a @ [STORE x] |
ccomp (c1;c2) = ccomp c1 @ ccomp c2 |
ccomp (IF b THEN c1 ELSE c2) =
  (let cc1 = ccomp c1; cc2 = ccomp c2; cb = bcomp b False (isize cc1 + 1)
   in cb @ cc1 @ JMP (isize cc2) # cc2) |
ccomp (WHILE b DO c) =
 (let cc = ccomp c; cb = bcomp b False (isize cc + 1)
  in cb @ cc @ [JMP (−(isize cb + isize cc + 1))])


value ccomp
 (IF Less (V ''u'') (N 1) THEN ''u'' ::= Plus (V ''u'') (N 1)
  ELSE ''v'' ::= V ''u'')

value ccomp (WHILE Less (V ''u'') (N 1) DO (''u'' ::= Plus (V ''u'') (N
1)))
```

5.5 Preservation of semantics

```
lemma ccomp-bigstep:
  (c,s) ⇒ t ⟹ ccomp c ⊢ (0,s,stk) →∗ (isize(ccomp c),t,stk)
proof(induction arbitrary: stk rule: big-step-induct)
  case (Assign x a s)
  show ?case by (fastforce simp:fun-upd-def cong: if-cong)
next
  case (Semi c1 s1 s2 c2 s3)
  let ?cc1 = ccomp c1  let ?cc2 = ccomp c2
  have ?cc1 @ ?cc2 ⊢ (0,s1,stk) →∗ (isize ?cc1,s2,stk)
    using Semi.IH(1) by fastforce
  moreover
  have ?cc1 @ ?cc2 ⊢ (isize ?cc1,s2,stk) →∗ (isize(?cc1 @ ?cc2),s3,stk)
    using Semi.IH(2) by fastforce
  ultimately show ?case by simp (blast intro: exec-trans)
next
  case (WhileTrue b s1 c s2 s3)
  let ?cc = ccomp c
```

let *?cb* = *bcomp b False* (*isize ?cc* + *1*)
let *?cw* = *ccomp*(*WHILE b DO c*)
have *?cw* ⊢ (*0*,*s1*,*stk*) →∗ (*isize ?cb* + *isize ?cc*,*s2*,*stk*)
using *WhileTrue.IH*(*1*) *WhileTrue.hyps*(*1*) **by** *fastforce*
moreover
have *?cw* ⊢ (*isize ?cb* + *isize ?cc*,*s2*,*stk*) →∗ (*0*,*s2*,*stk*)
by *fastforce*
moreover
have *?cw* ⊢ (*0*,*s2*,*stk*) →∗ (*isize ?cw*,*s3*,*stk*) **by**(*rule WhileTrue.IH*(*2*))
ultimately show *?case* **by**(*blast intro*: *exec-trans*)
qed *fastforce*+

end

6 A Typed Language

theory *Types* **imports** *Star Complex-Main* **begin**

6.1 Arithmetic Expressions

datatype *val* = *Iv int* | *Rv real*

type-synonym *vname* = *string*
type-synonym *state* = *vname* ⇒ *val*

datatype *aexp* = *Ic int* | *Rc real* | *V vname* | *Plus aexp aexp*

inductive *taval* :: *aexp* ⇒ *state* ⇒ *val* ⇒ *bool* **where**
taval (*Ic i*) *s* (*Iv i*) |
taval (*Rc r*) *s* (*Rv r*) |
taval (*V x*) *s* (*s x*) |
taval a1 s (*Iv i1*) ⟹ *taval a2 s* (*Iv i2*)
⟹ *taval* (*Plus a1 a2*) *s* (*Iv*(*i1*+*i2*)) |
taval a1 s (*Rv r1*) ⟹ *taval a2 s* (*Rv r2*)
⟹ *taval* (*Plus a1 a2*) *s* (*Rv*(*r1*+*r2*))

inductive-cases [*elim*!]:
taval (*Ic i*) *s v* *taval* (*Rc i*) *s v*
taval (*V x*) *s v*
taval (*Plus a1 a2*) *s v*

6.2 Boolean Expressions

datatype *bexp* = *Bc bool* | *Not bexp* | *And bexp bexp* | *Less aexp aexp*

inductive *tbval* :: *bexp* ⇒ *state* ⇒ *bool* ⇒ *bool* **where**
tbval (*Bc v*) *s v* |
tbval b s bv ⟹ *tbval* (*Not b*) *s* (¬ *bv*) |
tbval b1 s bv1 ⟹ *tbval b2 s bv2* ⟹ *tbval* (*And b1 b2*) *s* (*bv1* & *bv2*) |
taval a1 s (*Iv i1*) ⟹ *taval a2 s* (*Iv i2*) ⟹ *tbval* (*Less a1 a2*) *s* (*i1* < *i2*)
|
taval a1 s (*Rv r1*) ⟹ *taval a2 s* (*Rv r2*) ⟹ *tbval* (*Less a1 a2*) *s* (*r1* < *r2*)

6.3 Syntax of Commands

datatype
com = *SKIP*
| *Assign vname aexp* (- ::= - [*1000*, *61*] *61*)
| *Semi com com* (-; - [*60*, *61*] *60*)
| *If bexp com com* (*IF* - *THEN* - *ELSE* - [*0*, *0*, *61*] *61*)
| *While bexp com* (*WHILE* - *DO* - [*0*, *61*] *61*)

6.4 Small-Step Semantics of Commands

inductive
small-step :: (*com* × *state*) ⇒ (*com* × *state*) ⇒ *bool* (**infix** → *55*)
where
Assign: *taval a s v* ⟹ (*x* ::= *a*, *s*) → (*SKIP*, *s*(*x* := *v*)) |

Semi1: (*SKIP*;*c*,*s*) → (*c*,*s*) |
Semi2: (*c1*,*s*) → (*c1*′,*s*′) ⟹ (*c1*;*c2*,*s*) → (*c1*′;*c2*,*s*′) |

IfTrue: *tbval b s True* ⟹ (*IF b THEN c1 ELSE c2*,*s*) → (*c1*,*s*) |
IfFalse: *tbval b s False* ⟹ (*IF b THEN c1 ELSE c2*,*s*) → (*c2*,*s*) |

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

lemmas *small-step-induct* = *small-step.induct*[*split-format*(*complete*)]

6.5 The Type System

datatype *ty* = *Ity* | *Rty*

type-synonym *tyenv* = *vname* ⇒ *ty*

inductive *atyping* :: *tyenv* ⇒ *aexp* ⇒ *ty* ⇒ *bool*
((*1-*/ ⊢/ (- :/ -)) [*50*,*0*,*50*] *50*)
where
Ic-ty: Γ ⊢ *Ic i* : *Ity* |
Rc-ty: Γ ⊢ *Rc r* : *Rty* |
V-ty: Γ ⊢ *V x* : Γ *x* |
Plus-ty: Γ ⊢ *a1* : τ ⟹ Γ ⊢ *a2* : τ ⟹ Γ ⊢ *Plus a1 a2* : τ

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

inductive *btyping* :: *tyenv* ⇒ *bexp* ⇒ *bool* (**infix** ⊢ *50*)
where
B-ty: Γ ⊢ *Bc v* |
Not-ty: Γ ⊢ *b* ⟹ Γ ⊢ *Not b* |
And-ty: Γ ⊢ *b1* ⟹ Γ ⊢ *b2* ⟹ Γ ⊢ *And b1 b2* |
Less-ty: Γ ⊢ *a1* : τ ⟹ Γ ⊢ *a2* : τ ⟹ Γ ⊢ *Less a1 a2*

inductive *ctyping* :: *tyenv* ⇒ *com* ⇒ *bool* (**infix** ⊢ *50*) **where**
Skip-ty: Γ ⊢ *SKIP* |
Assign-ty: Γ ⊢ *a* : Γ(*x*) ⟹ Γ ⊢ *x* ::= *a* |
Semi-ty: Γ ⊢ *c1* ⟹ Γ ⊢ *c2* ⟹ Γ ⊢ *c1*;*c2* |
If-ty: Γ ⊢ *b* ⟹ Γ ⊢ *c1* ⟹ Γ ⊢ *c2* ⟹ Γ ⊢ *IF b THEN c1 ELSE c2* |
While-ty: Γ ⊢ *b* ⟹ Γ ⊢ *c* ⟹ Γ ⊢ *WHILE b DO c*

inductive-cases [*elim*!]:
Γ ⊢ *x* ::= *a* Γ ⊢ *c1*;*c2*
Γ ⊢ *IF b THEN c1 ELSE c2*
Γ ⊢ *WHILE b DO c*

6.6 Well-typed Programs Do Not Get Stuck

fun *type* :: *val* ⇒ *ty* **where**
type (*Iv i*) = *Ity* |
type (*Rv r*) = *Rty*

lemma [*simp*]: *type v* = *Ity* ⟷ (∃ *i*. *v* = *Iv i*)
by (*cases v*) *simp-all*

lemma [*simp*]: *type v* = *Rty* ⟷ (∃ *r*. *v* = *Rv r*)
by (*cases v*) *simp-all*

definition *styping* :: *tyenv* ⇒ *state* ⇒ *bool* (**infix** ⊢ *50*)
where Γ ⊢ *s* ⟷ (∀ *x*. *type* (*s x*) = Γ *x*)

lemma *apreservation*:
  Γ ⊢ *a* : τ ⟹ *taval a s v* ⟹ Γ ⊢ *s* ⟹ *type v* = τ
apply(*induction arbitrary*: *v rule*: *atyping.induct*)
apply (*fastforce simp*: *styping-def*)+
done

lemma *aprogress*: Γ ⊢ *a* : τ ⟹ Γ ⊢ *s* ⟹ ∃ *v*. *taval a s v*
proof(*induction rule*: *atyping.induct*)
  case (*Plus-ty* Γ *a1 t a2*)
  then obtain *v1 v2* **where** *v*: *taval a1 s v1 taval a2 s v2* **by** *blast*
  show *?case*
  proof (*cases v1*)
    case *Iv*
    with *Plus-ty v* **show** *?thesis*
      by(*fastforce intro*: *taval.intros*(*4*) *dest*!: *apreservation*)
  next
    case *Rv*
    with *Plus-ty v* **show** *?thesis*
      by(*fastforce intro*: *taval.intros*(*5*) *dest*!: *apreservation*)
  qed
qed (*auto intro*: *taval.intros*)

lemma *bprogress*: Γ ⊢ *b* ⟹ Γ ⊢ *s* ⟹ ∃ *v*. *tbval b s v*
proof(*induction rule*: *btyping.induct*)
  case (*Less-ty* Γ *a1 t a2*)
  then obtain *v1 v2* **where** *v*: *taval a1 s v1 taval a2 s v2*
    by (*metis aprogress*)
  show *?case*
  proof (*cases v1*)
    case *Iv*
    with *Less-ty v* **show** *?thesis*
      by (*fastforce intro*!: *tbval.intros*(*4*) *dest*!:*apreservation*)
  next
    case *Rv*
    with *Less-ty v* **show** *?thesis*
      by (*fastforce intro*!: *tbval.intros*(*5*) *dest*!:*apreservation*)
  qed
qed (*auto intro*: *tbval.intros*)

theorem *progress*:
  Γ ⊢ *c* ⟹ Γ ⊢ *s* ⟹ *c* ≠ *SKIP* ⟹ ∃ *cs'*. (*c*,*s*) → *cs'*

proof(*induction rule*: *ctyping.induct*)
case *Skip-ty* **thus** *?case* **by** *simp*
next
case *Assign-ty*
thus *?case* **by** (*metis Assign aprogress*)
next
case *Semi-ty* **thus** *?case* **by** *simp* (*metis Semi1 Semi2*)
next
case (*If-ty* Γ *b c1 c2*)
then obtain *bv* **where** *tbval b s bv* **by** (*metis bprogress*)
show *?case*
proof(*cases bv*)
assume *bv*
with ‹*tbval b s bv*› **show** *?case* **by** *simp* (*metis IfTrue*)
next
assume ¬*bv*
with ‹*tbval b s bv*› **show** *?case* **by** *simp* (*metis IfFalse*)
qed
next
case *While-ty* **show** *?case* **by** (*metis While*)
qed

theorem *styping-preservation*:
(*c*,*s*) → (*c*′,*s*′) ⟹ Γ ⊢ *c* ⟹ Γ ⊢ *s* ⟹ Γ ⊢ *s*′
proof(*induction rule*: *small-step-induct*)
case *Assign* **thus** *?case*
by (*auto simp*: *styping-def*) (*metis Assign*(*1*,*3*) *apreservation*)
qed *auto*

theorem *ctyping-preservation*:
(*c*,*s*) → (*c*′,*s*′) ⟹ Γ ⊢ *c* ⟹ Γ ⊢ *c*′
by (*induct rule*: *small-step-induct*) (*auto simp*: *ctyping.intros*)

abbreviation *small-steps* :: *com* ∗ *state* ⇒ *com* ∗ *state* ⇒ *bool* (**infix** →∗ *55*)
where *x* →∗ *y* == *star small-step x y*

theorem *type-sound*:
(*c*,*s*) →∗ (*c*′,*s*′) ⟹ Γ ⊢ *c* ⟹ Γ ⊢ *s* ⟹ *c*′ ≠ *SKIP*
⟹ ∃ *cs*″. (*c*′,*s*′) → *cs*″
apply(*induction rule*:*star-induct*)
apply (*metis progress*)
by (*metis styping-preservation ctyping-preservation*)

end

7 Definite Assignment Analysis

theory *Vars* **imports** *BExp*
begin

7.1 Show sets of variables as lists

Sets may be infinite and are not always displayed by element if computed as values. Lists do not have this problem.

We define a function *show* that converts a set of variable names into a list. To keep things simple, we rely on the convention that we only use single letter names.

definition
letters :: *string list* **where**
letters = *map* (λ*c*. [*c*]) *chars*

definition
show :: *string set* ⇒ *string list* **where**
show S = *filter* (λ*x*. *x* ∈ *S*) *letters*

value *show* {*''x''*, *''z''*}

7.2 The Variables in an Expression

We need to collect the variables in both arithmetic and boolean expressions. For a change we do not introduce two functions, e.g. *avars* and *bvars*, but we overload the name *vars* via a *type class*, a device that originated with Haskell:

class *vars* =
fixes *vars* :: *'a* ⇒ *vname set*

This defines a type class "vars" with a single function of (coincidentally) the same name. Then we define two separated instances of the class, one for *aexp* and one for *bexp*:

instantiation *aexp* :: *vars*
begin

fun *vars-aexp* :: *aexp* ⇒ *vname set* **where**
vars (*N n*) = {} |
vars (*V x*) = {*x*} |

vars (*Plus* a_1 a_2) = *vars* a_1 ∪ *vars* a_2

instance ..

end

value *vars*(*Plus* (*V* ''*x*'') (*V* ''*y*''))

We need to convert functions to lists before we can view them:

value *show* (*vars*(*Plus* (*V* ''*x*'') (*V* ''*y*'')))

instantiation *bexp* :: *vars*
begin

fun *vars-bexp* :: *bexp* ⇒ *vname set* **where**
vars (*Bc v*) = {} |
vars (*Not b*) = *vars b* |
vars (*And* b_1 b_2) = *vars* b_1 ∪ *vars* b_2 |
vars (*Less* a_1 a_2) = *vars* a_1 ∪ *vars* a_2

instance ..

end

value *show* (*vars*(*Less* (*Plus* (*V* ''*z*'') (*V* ''*y*'')) (*V* ''*x*'')))

abbreviation
eq-on :: ('*a* ⇒ '*b*) ⇒ ('*a* ⇒ '*b*) ⇒ '*a set* ⇒ *bool*
((- =/ -/ *on* -) [*50*,*0*,*50*] *50*) **where**
f = *g on X* == ∀ *x* ∈ *X*. *f x* = *g x*

lemma *aval-eq-if-eq-on-vars*[*simp*]:
s_1 = s_2 *on vars a* ⟹ *aval a* s_1 = *aval a* s_2
apply(*induction a*)
apply *simp-all*
done

lemma *bval-eq-if-eq-on-vars*:
s_1 = s_2 *on vars b* ⟹ *bval b* s_1 = *bval b* s_2
proof(*induction b*)
case (*Less a1 a2*)
hence *aval a1* s_1 = *aval a1* s_2 **and** *aval a2* s_1 = *aval a2* s_2 **by** *simp-all*
thus *?case* **by** *simp*
qed *simp-all*

end

theory *Def-Ass* **imports** *Vars Com*
begin

7.3 Definite Assignment Analysis

inductive $D :: vname\ set \Rightarrow com \Rightarrow vname\ set \Rightarrow bool$ **where**
Skip: $D\ A\ SKIP\ A$ |
Assign: $vars\ a \subseteq A \Longrightarrow D\ A\ (x ::= a)\ (insert\ x\ A)$ |
Semi: $[\![\ D\ A_1\ c_1\ A_2;\ \ D\ A_2\ c_2\ A_3\ ]\!] \Longrightarrow D\ A_1\ (c_1;\ c_2)\ A_3$ |
If: $[\![\ vars\ b \subseteq A;\ \ D\ A\ c_1\ A_1;\ \ D\ A\ c_2\ A_2\ ]\!] \Longrightarrow$
$D\ A\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ (A_1\ Int\ A_2)$ |
While: $[\![\ vars\ b \subseteq A;\ \ D\ A\ c\ A'\ ]\!] \Longrightarrow D\ A\ (WHILE\ b\ DO\ c)\ A$

inductive-cases [*elim*!]:
$D\ A\ SKIP\ A'$
$D\ A\ (x ::= a)\ A'$
$D\ A\ (c1;c2)\ A'$
$D\ A\ (IF\ b\ THEN\ c1\ ELSE\ c2)\ A'$
$D\ A\ (WHILE\ b\ DO\ c)\ A'$

lemma *D-incr*:
$D\ A\ c\ A' \Longrightarrow A \subseteq A'$
by (*induct rule*: *D.induct*) *auto*

end

theory *Def-Ass-Exp* **imports** *Vars*
begin

7.4 Initialization-Sensitive Expressions Evaluation

type-synonym $state = vname \Rightarrow val\ option$

fun $aval :: aexp \Rightarrow state \Rightarrow val\ option$ **where**
$aval\ (N\ i)\ s = Some\ i$ |
$aval\ (V\ x)\ s = s\ x$ |
$aval\ (Plus\ a_1\ a_2)\ s =$
$(case\ (aval\ a_1\ s,\ aval\ a_2\ s)\ of$

(*Some* i_1,*Some* i_2) ⇒ *Some*(i_1+i_2) | - ⇒ *None*)

fun *bval* :: *bexp* ⇒ *state* ⇒ *bool option* **where**
bval (*Bc v*) *s* = *Some v* |
bval (*Not b*) *s* = (*case bval b s of None* ⇒ *None* | *Some bv* ⇒ *Some*(¬ *bv*))
|
bval (*And* b_1 b_2) *s* = (*case* (*bval* b_1 *s*, *bval* b_2 *s*) *of*
(*Some* bv_1, *Some* bv_2) ⇒ *Some*(bv_1 & bv_2) | - ⇒ *None*) |
bval (*Less* a_1 a_2) *s* = (*case* (*aval* a_1 *s*, *aval* a_2 *s*) *of*
(*Some* i_1, *Some* i_2) ⇒ *Some*(i_1 < i_2) | - ⇒ *None*)

lemma *aval-Some*: *vars a* ⊆ *dom s* ⟹ ∃ *i*. *aval a s* = *Some i*
by (*induct a*) *auto*

lemma *bval-Some*: *vars b* ⊆ *dom s* ⟹ ∃ *bv*. *bval b s* = *Some bv*
by (*induct b*) (*auto dest*!: *aval-Some*)

end

theory *Def-Ass-Big* **imports** *Com Def-Ass-Exp*
begin

7.5 Initialization-Sensitive Big Step Semantics

inductive
big-step :: (*com* × *state option*) ⇒ *state option* ⇒ *bool* (**infix** ⇒ *55*)
where
None: (*c*,*None*) ⇒ *None* |
Skip: (*SKIP*,*s*) ⇒ *s* |
AssignNone: *aval a s* = *None* ⟹ (*x* ::= *a*, *Some s*) ⇒ *None* |
Assign: *aval a s* = *Some i* ⟹ (*x* ::= *a*, *Some s*) ⇒ *Some*(*s*(*x* := *Some i*))
|
Semi: (*c*$_1$,*s*$_1$) ⇒ *s*$_2$ ⟹ (*c*$_2$,*s*$_2$) ⇒ *s*$_3$ ⟹ (*c*$_1$;*c*$_2$,*s*$_1$) ⇒ *s*$_3$ |

IfNone: *bval b s* = *None* ⟹ (*IF b THEN* c_1 *ELSE* c_2,*Some s*) ⇒ *None* |
IfTrue: ⟦ *bval b s* = *Some True*; (*c*$_1$,*Some s*) ⇒ *s*′ ⟧ ⟹
(*IF b THEN* c_1 *ELSE* c_2,*Some s*) ⇒ *s*′ |
IfFalse: ⟦ *bval b s* = *Some False*; (*c*$_2$,*Some s*) ⇒ *s*′ ⟧ ⟹
(*IF b THEN* c_1 *ELSE* c_2,*Some s*) ⇒ *s*′ |

WhileNone: *bval b s* = *None* $\Longrightarrow$ (*WHILE b DO c*,*Some s*) $\Rightarrow$ *None* |
WhileFalse: *bval b s* = *Some False* $\Longrightarrow$ (*WHILE b DO c*,*Some s*) $\Rightarrow$ *Some s* |
WhileTrue:
⟦ *bval b s* = *Some True*; (*c*,*Some s*) $\Rightarrow$ *s'*; (*WHILE b DO c*,*s'*) $\Rightarrow$ *s''* ⟧
$\Longrightarrow$
(*WHILE b DO c*,*Some s*) $\Rightarrow$ *s''*

lemmas *big-step-induct* = *big-step.induct*[*split-format*(*complete*)]

end

theory *Def-Ass-Sound-Big* **imports** *Def-Ass Def-Ass-Big*
begin

7.6 Soundness wrt Big Steps

Note the special form of the induction because one of the arguments of the inductive predicate is not a variable but the term *Some s*:

theorem *Sound*:
⟦ (*c*,*Some s*) $\Rightarrow$ *s'*; *D A c A'*; *A* $\subseteq$ *dom s* ⟧
$\Longrightarrow \exists$ *t*. *s'* = *Some t* $\wedge$ *A'* $\subseteq$ *dom t*
proof (*induction c Some s s' arbitrary*: *s A A' rule*:*big-step-induct*)
case *AssignNone* **thus** *?case*
by *auto* (*metis aval-Some option.simps*(*3*) *subset-trans*)
next
case *Semi* **thus** *?case* **by** *auto metis*
next
case *IfTrue* **thus** *?case* **by** *auto blast*
next
case *IfFalse* **thus** *?case* **by** *auto blast*
next
case *IfNone* **thus** *?case*
by *auto* (*metis bval-Some option.simps*(*3*) *order-trans*)
next
case *WhileNone* **thus** *?case*
by *auto* (*metis bval-Some option.simps*(*3*) *order-trans*)
next
case (*WhileTrue b s c s' s''*)
from ‹*D A* (*WHILE b DO c*) *A'*› **obtain** *A'* **where** *D A c A'* **by** *blast*
then obtain *t'* **where** *s'* = *Some t' A* $\subseteq$ *dom t'*
by (*metis D-incr WhileTrue*(*3*,*7*) *subset-trans*)

from *WhileTrue(5)*[*OF this(1) WhileTrue(6) this(2)*] **show** *?case* **.**
qed *auto*

corollary *sound*: ⟦ *D* (*dom s*) *c A'*; (*c*,*Some s*) ⇒ *s'* ⟧ ⟹ *s'* ≠ *None*
by (*metis Sound not-Some-eq subset-refl*)

end

8 Live Variable Analysis

theory *Live* **imports** *Vars Big-Step*
begin

8.1 Liveness Analysis

fun *L* :: *com* ⇒ *vname set* ⇒ *vname set* **where**
L SKIP X = *X* |
L (*x* ::= *a*) *X* = *X*−{*x*} ∪ *vars a* |
L (*c*$_1$; *c*$_2$) *X* = (*L c*$_1$ ∘ *L c*$_2$) *X* |
L (*IF b THEN c*$_1$ *ELSE c*$_2$) *X* = *vars b* ∪ *L c*$_1$ *X* ∪ *L c*$_2$ *X* |
L (*WHILE b DO c*) *X* = *vars b* ∪ *X* ∪ *L c X*

value *show* (*L* (''*y*'' ::= *V* ''*z*''; ''*x*'' ::= *Plus* (*V* ''*y*'') (*V* ''*z*'')) {''*x*''})

value *show* (*L* (*WHILE Less* (*V* ''*x*'') (*V* ''*x*'') *DO* ''*y*'' ::= *V* ''*z*'') {''*x*''})

fun *kill* :: *com* ⇒ *vname set* **where**
kill SKIP = {} |
kill (*x* ::= *a*) = {*x*} |
kill (*c*$_1$; *c*$_2$) = *kill c*$_1$ ∪ *kill c*$_2$ |
kill (*IF b THEN c*$_1$ *ELSE c*$_2$) = *kill c*$_1$ ∩ *kill c*$_2$ |
kill (*WHILE b DO c*) = {}

fun *gen* :: *com* ⇒ *vname set* **where**
gen SKIP = {} |
gen (*x* ::= *a*) = *vars a* |
gen (*c*$_1$; *c*$_2$) = *gen c*$_1$ ∪ (*gen c*$_2$ − *kill c*$_1$) |
gen (*IF b THEN c*$_1$ *ELSE c*$_2$) = *vars b* ∪ *gen c*$_1$ ∪ *gen c*$_2$ |
gen (*WHILE b DO c*) = *vars b* ∪ *gen c*

lemma *L-gen-kill*: *L c X* = (*X* − *kill c*) ∪ *gen c*
by(*induct c arbitrary*:*X*) *auto*

lemma *L-While-pfp*: *L c* (*L* (*WHILE b DO c*) *X*) ⊆ *L* (*WHILE b DO c*) *X*
by(*auto simp add*:*L-gen-kill*)

lemma *L-While-lpfp*:
vars b ∪ *X* ∪ *L c P* ⊆ *P* ⟹ *L* (*WHILE b DO c*) *X* ⊆ *P*
by(*simp add*: *L-gen-kill*)

8.2 Soundness

theorem *L-sound*:
(*c*,*s*) ⇒ *s'* ⟹ *s* = *t on L c X* ⟹
∃ *t'*. (*c*,*t*) ⇒ *t'* & *s'* = *t'* *on X*
proof (*induction arbitrary*: *X t rule*: *big-step-induct*)
case *Skip* **then show** *?case* **by** *auto*
next
case *Assign* **then show** *?case*
by (*auto simp*: *ball-Un*)
next
case (*Semi c1 s1 s2 c2 s3 X t1*)
from *Semi.IH*(*1*) *Semi.prems* **obtain** *t2* **where**
t12: (*c1*, *t1*) ⇒ *t2* **and** *s2t2*: *s2* = *t2 on L c2 X*
by *simp blast*
from *Semi.IH*(*2*)[*OF s2t2*] **obtain** *t3* **where**
t23: (*c2*, *t2*) ⇒ *t3* **and** *s3t3*: *s3* = *t3 on X*
by *auto*
show *?case* **using** *t12 t23 s3t3* **by** *auto*
next
case (*IfTrue b s c1 s' c2*)
hence *s* = *t on vars b s* = *t on L c1 X* **by** *auto*
from *bval-eq-if-eq-on-vars*[*OF this*(*1*)] *IfTrue*(*1*) **have** *bval b t* **by** *simp*
from *IfTrue*(*3*)[*OF* ‹*s* = *t on L c1 X*›] **obtain** *t'* **where**
(*c1*, *t*) ⇒ *t' s'* = *t'* *on X* **by** *auto*
thus *?case* **using** ‹*bval b t*› **by** *auto*
next
case (*IfFalse b s c2 s' c1*)
hence *s* = *t on vars b s* = *t on L c2 X* **by** *auto*
from *bval-eq-if-eq-on-vars*[*OF this*(*1*)] *IfFalse*(*1*) **have** ~*bval b t* **by** *simp*
from *IfFalse*(*3*)[*OF* ‹*s* = *t on L c2 X*›] **obtain** *t'* **where**
(*c2*, *t*) ⇒ *t' s'* = *t'* *on X* **by** *auto*
thus *?case* **using** ‹~*bval b t*› **by** *auto*
next
case (*WhileFalse b s c*)
hence ~ *bval b t* **by** (*auto simp*: *ball-Un*) (*metis bval-eq-if-eq-on-vars*)

thus *?case* **using** *WhileFalse.prems* **by** *auto*
next
case (*WhileTrue b s1 c s2 s3 X t1*)
let *?w* = *WHILE b DO c*
from ‹*bval b s1*› *WhileTrue.prems* **have** *bval b t1*
by (*auto simp*: *ball-Un*) (*metis bval-eq-if-eq-on-vars*)
have *s1* = *t1 on L c* (*L ?w X*) **using** *L-While-pfp WhileTrue.prems*
by (*blast*)
from *WhileTrue.IH*(*1*)[*OF this*] **obtain** *t2* **where**
(*c*, *t1*) ⇒ *t2 s2* = *t2 on L ?w X* **by** *auto*
from *WhileTrue.IH*(*2*)[*OF this*(*2*)] **obtain** *t3* **where** (*?w*,*t2*) ⇒ *t3 s3* = *t3 on X*
by *auto*
with ‹*bval b t1*› ‹(*c*, *t1*) ⇒ *t2*› **show** *?case* **by** *auto*
qed

8.3 Program Optimization

Burying assignments to dead variables:

fun *bury* :: *com* ⇒ *vname set* ⇒ *com* **where**
bury SKIP X = *SKIP* |
bury (*x* ::= *a*) *X* = (*if x*:*X then x*::= *a else SKIP*) |
bury (*c*$_1$; *c*$_2$) *X* = (*bury c*$_1$ (*L c*$_2$ *X*); *bury c*$_2$ *X*) |
bury (*IF b THEN c*$_1$ *ELSE c*$_2$) *X* = *IF b THEN bury c*$_1$ *X ELSE bury c*$_2$ *X* |
bury (*WHILE b DO c*) *X* = *WHILE b DO bury c* (*vars b* ∪ *X* ∪ *L c X*)

We could prove the analogous lemma to *L-sound*, and the proof would be very similar. However, we phrase it as a semantics preservation property:

theorem *bury-sound*:
(*c*,*s*) ⇒ *s*′ ⟹ *s* = *t on L c X* ⟹
∃ *t*′. (*bury c X*,*t*) ⇒ *t*′ & *s*′ = *t*′ *on X*
proof (*induction arbitrary*: *X t rule*: *big-step-induct*)
case *Skip* **then show** *?case* **by** *auto*
next
case *Assign* **then show** *?case*
by (*auto simp*: *ball-Un*)
next
case (*Semi c1 s1 s2 c2 s3 X t1*)
from *Semi.IH*(*1*) *Semi.prems* **obtain** *t2* **where**
t12: (*bury c1* (*L c2 X*), *t1*) ⇒ *t2* **and** *s2t2*: *s2* = *t2 on L c2 X*
by *simp blast*
from *Semi.IH*(*2*)[*OF s2t2*] **obtain** *t3* **where**
t23: (*bury c2 X*, *t2*) ⇒ *t3* **and** *s3t3*: *s3* = *t3 on X*

```
    by auto
  show ?case using t12 t23 s3t3 by auto
next
  case (IfTrue b s c1 s' c2)
  hence s = t on vars b s = t on L c1 X by auto
  from  bval-eq-if-eq-on-vars[OF this(1)] IfTrue(1) have bval b t by simp
  from IfTrue(3)[OF ‹s = t on L c1 X›] obtain t' where
    (bury c1 X, t) ⇒ t' s' =t' on X by auto
  thus ?case using ‹bval b t› by auto
next
  case (IfFalse b s c2 s' c1)
  hence s = t on vars b s = t on L c2 X by auto
 from  bval-eq-if-eq-on-vars[OF this(1)] IfFalse(1) have ~bval b t by simp
  from IfFalse(3)[OF ‹s = t on L c2 X›] obtain t' where
    (bury c2 X, t) ⇒ t' s' = t' on X by auto
  thus ?case using ‹~bval b t› by auto
next
  case (WhileFalse b s c)
  hence ~ bval b t by (auto simp: ball-Un) (metis bval-eq-if-eq-on-vars)
  thus ?case using WhileFalse.prems by auto
next
  case (WhileTrue b s1 c s2 s3 X t1)
  let ?w = WHILE b DO c
  from ‹bval b s1› WhileTrue.prems have bval b t1
    by (auto simp: ball-Un) (metis bval-eq-if-eq-on-vars)
  have s1 = t1 on L c (L ?w X)
    using L-While-pfp WhileTrue.prems by blast
  from WhileTrue.IH(1)[OF this] obtain t2 where
    (bury c (L ?w X), t1) ⇒ t2 s2 = t2 on L ?w X by auto
  from WhileTrue.IH(2)[OF this(2)] obtain t3
    where (bury ?w X,t2) ⇒ t3 s3 = t3 on X
    by auto
  with ‹bval b t1› ‹(bury c (L ?w X), t1) ⇒ t2› show ?case by auto
qed

corollary final-bury-sound: (c,s) ⇒ s' ⟹ (bury c UNIV,s) ⇒ s'
using bury-sound[of c s s' UNIV]
by (auto simp: fun-eq-iff[symmetric])
```

Now the opposite direction.

```
lemma SKIP-bury[simp]:
  SKIP = bury c X ⟷ c = SKIP | (EX x a. c = x::=a & x ∉ X)
by (cases c) auto
```

lemma *Assign-bury*[*simp*]: $x::=a = bury\ c\ X \longleftrightarrow c = x::=a\ \&\ x : X$
by (*cases c*) *auto*

lemma *Semi-bury*[*simp*]: $bc_1;bc_2 = bury\ c\ X \longleftrightarrow$
$(EX\ c_1\ c_2.\ c = c_1;c_2\ \&\ bc_2 = bury\ c_2\ X\ \&\ bc_1 = bury\ c_1\ (L\ c_2\ X))$
by (*cases c*) *auto*

lemma *If-bury*[*simp*]: *IF b THEN bc1 ELSE bc2* $= bury\ c\ X \longleftrightarrow$
(*EX c1 c2. c = IF b THEN c1 ELSE c2 &*
bc1 $= bury\ c1\ X$ *& bc2* $= bury\ c2\ X$)
by (*cases c*) *auto*

lemma *While-bury*[*simp*]: *WHILE b DO bc'* $= bury\ c\ X \longleftrightarrow$
$(EX\ c'.\ c = WHILE\ b\ DO\ c'\ \&\ bc' = bury\ c'\ (vars\ b \cup X \cup L\ c\ X))$
by (*cases c*) *auto*

theorem *bury-sound2*:
$(bury\ c\ X,s) \Rightarrow s' \Longrightarrow s = t\ on\ L\ c\ X \Longrightarrow$
$\exists\ t'.\ (c,t) \Rightarrow t'\ \&\ s' = t'\ on\ X$
proof (*induction bury c X s s' arbitrary*: *c X t rule*: *big-step-induct*)
case *Skip* **then show** *?case* **by** *auto*
next
case *Assign* **then show** *?case*
by (*auto simp*: *ball-Un*)
next
case (*Semi bc1 s1 s2 bc2 s3 c X t1*)
then obtain *c1 c2* **where** *c*: *c = c1;c2*
and *bc2*: *bc2 = bury c2 X* **and** *bc1*: *bc1 = bury c1* (*L c2 X*) **by** *auto*
note *IH* = *Semi.hyps*(*2*,*4*)
from *IH*(*1*)[*OF bc1*, *of t1*] *Semi.prems c* **obtain** *t2* **where**
t12: (*c1*, *t1*) ⇒ *t2* **and** *s2t2*: *s2 = t2 on L c2 X* **by** *auto*
from *IH*(*2*)[*OF bc2 s2t2*] **obtain** *t3* **where**
t23: (*c2*, *t2*) ⇒ *t3* **and** *s3t3*: *s3 = t3 on X*
by *auto*
show *?case* **using** *c t12 t23 s3t3* **by** *auto*
next
case (*IfTrue b s bc1 s' bc2*)
then obtain *c1 c2* **where** *c*: *c = IF b THEN c1 ELSE c2*
and *bc1*: *bc1 = bury c1 X* **and** *bc2*: *bc2 = bury c2 X* **by** *auto*
have *s = t on vars b s = t on L c1 X* **using** *IfTrue.prems c* **by** *auto*
from *bval-eq-if-eq-on-vars*[*OF this*(*1*)] *IfTrue*(*1*) **have** *bval b t* **by** *simp*
note *IH* = *IfTrue.hyps*(*3*)
from *IH*[*OF bc1* ‹*s = t on L c1 X*›] **obtain** *t'* **where**
(*c1*, *t*) ⇒ *t' s' =t' on X* **by** *auto*

thus *?case* **using** *c* ‹*bval b t*› **by** *auto*
next
case (*IfFalse b s bc2 s' bc1*)
then obtain *c1 c2* **where** *c*: *c = IF b THEN c1 ELSE c2*
and *bc1*: *bc1 = bury c1 X* **and** *bc2*: *bc2 = bury c2 X* **by** *auto*
have *s = t on vars b s = t on L c2 X* **using** *IfFalse.prems c* **by** *auto*
from *bval-eq-if-eq-on-vars*[*OF this*(*1*)] *IfFalse*(*1*) **have** ~*bval b t* **by** *simp*
note *IH = IfFalse.hyps*(*3*)
from *IH*[*OF bc2* ‹*s = t on L c2 X*›] **obtain** *t'* **where**
(*c2*, *t*) ⇒ *t' s' =t' on X* **by** *auto*
thus *?case* **using** *c* ‹~*bval b t*› **by** *auto*
next
case (*WhileFalse b s c*)
hence ~ *bval b t* **by** (*auto simp*: *ball-Un dest*: *bval-eq-if-eq-on-vars*)
thus *?case* **using** *WhileFalse* **by** *auto*
next
case (*WhileTrue b s1 bc' s2 s3 c X t1*)
then obtain *c'* **where** *c*: *c = WHILE b DO c'*
and *bc'*: *bc' = bury c'* (*vars b* ∪ *X* ∪ *L c' X*) **by** *auto*
let *?w = WHILE b DO c'*
from ‹*bval b s1*› *WhileTrue.prems c* **have** *bval b t1*
by (*auto simp*: *ball-Un*) (*metis bval-eq-if-eq-on-vars*)
note *IH = WhileTrue.hyps*(*3*,*5*)
have *s1 = t1 on L c'* (*L ?w X*)
using *L-While-pfp WhileTrue.prems c* **by** *blast*
with *IH*(*1*)[*OF bc', of t1*] **obtain** *t2* **where**
(*c'*, *t1*) ⇒ *t2 s2 = t2 on L ?w X* **by** *auto*
from *IH*(*2*)[*OF WhileTrue.hyps*(*6*), *of t2*] *c this*(*2*) **obtain** *t3*
where (*?w*,*t2*) ⇒ *t3 s3 = t3 on X*
by *auto*
with ‹*bval b t1*› ‹(*c'*, *t1*) ⇒ *t2*› *c* **show** *?case* **by** *auto*
qed

corollary *final-bury-sound2*: (*bury c UNIV*,*s*) ⇒ *s'* ⟹ (*c*,*s*) ⇒ *s'*
using *bury-sound2*[*of c UNIV*]
by (*auto simp*: *fun-eq-iff*[*symmetric*])

corollary *bury-iff*: (*bury c UNIV*,*s*) ⇒ *s'* ⟷ (*c*,*s*) ⇒ *s'*
by(*metis final-bury-sound final-bury-sound2*)

end

theory *Live-True*
imports ~~/*src*/*HOL*/*Library*/*While-Combinator Vars Big-Step*
begin

8.4 True Liveness Analysis

fun $L :: com \Rightarrow vname\ set \Rightarrow vname\ set$ **where**
$L\ SKIP\ X = X\ |$
$L\ (x ::= a)\ X = (if\ x{:}X\ then\ X{-}\{x\} \cup vars\ a\ else\ X)\ |$
$L\ (c_1;\ c_2)\ X = (L\ c_1 \circ L\ c_2)\ X\ |$
$L\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ X = vars\ b \cup L\ c_1\ X \cup L\ c_2\ X\ |$
$L\ (WHILE\ b\ DO\ c)\ X = lfp(\%\, Y.\ vars\ b \cup X \cup L\ c\ Y)$

lemma *L-mono*: $mono\ (L\ c)$
proof−
  { fix $X\ Y$ **have** $X \subseteq Y \Longrightarrow L\ c\ X \subseteq L\ c\ Y$
    proof(*induction c arbitrary*: $X\ Y$)
      case (*While b c*)
      show *?case*
      proof(*simp*, *rule lfp-mono*)
        fix Z **show** $vars\ b \cup X \cup L\ c\ Z \subseteq vars\ b \cup Y \cup L\ c\ Z$
          using *While* **by** *auto*
      qed
    next
      case *If* **thus** *?case* **by**(*auto simp*: *subset-iff*)
    qed *auto*
  } thus *?thesis* **by**(*rule monoI*)
qed

lemma *mono-union-L*:
  $mono\ (\%\, Y.\ X \cup L\ c\ Y)$
by (*metis* (*no-types*) *L-mono mono-def order-eq-iff set-eq-subset sup-mono*)

lemma *L-While-unfold*:
  $L\ (WHILE\ b\ DO\ c)\ X = vars\ b \cup X \cup L\ c\ (L\ (WHILE\ b\ DO\ c)\ X)$
by(*metis lfp-unfold*[*OF mono-union-L*] *L.simps*(*5*))

8.5 Soundness

theorem *L-sound*:
  $(c,s) \Rightarrow s' \Longrightarrow s = t\ on\ L\ c\ X \Longrightarrow$
  $\exists\ t'.\ (c,t) \Rightarrow t'\ \&\ s' = t'\ on\ X$
proof (*induction arbitrary*: $X\ t$ *rule*: *big-step-induct*)
  case *Skip* **then show** *?case* **by** *auto*

next
case *Assign* **then show** *?case*
by (*auto simp*: *ball-Un*)
next
case (*Semi c1 s1 s2 c2 s3 X t1*)
from *Semi.IH*(*1*) *Semi.prems* **obtain** *t2* **where**
t12: (*c1*, *t1*) ⇒ *t2* **and** *s2t2*: *s2* = *t2 on L c2 X*
by *simp blast*
from *Semi.IH*(*2*)[*OF s2t2*] **obtain** *t3* **where**
t23: (*c2*, *t2*) ⇒ *t3* **and** *s3t3*: *s3* = *t3 on X*
by *auto*
show *?case* **using** *t12 t23 s3t3* **by** *auto*
next
case (*IfTrue b s c1 s' c2*)
hence *s* = *t on vars b s* = *t on L c1 X* **by** *auto*
from *bval-eq-if-eq-on-vars*[*OF this*(*1*)] *IfTrue*(*1*) **have** *bval b t* **by** *simp*
from *IfTrue*(*3*)[*OF* ‹*s* = *t on L c1 X*›] **obtain** *t'* **where**
(*c1*, *t*) ⇒ *t' s'* = *t' on X* **by** *auto*
thus *?case* **using** ‹*bval b t*› **by** *auto*
next
case (*IfFalse b s c2 s' c1*)
hence *s* = *t on vars b s* = *t on L c2 X* **by** *auto*
from *bval-eq-if-eq-on-vars*[*OF this*(*1*)] *IfFalse*(*1*) **have** ~*bval b t* **by** *simp*
from *IfFalse*(*3*)[*OF* ‹*s* = *t on L c2 X*›] **obtain** *t'* **where**
(*c2*, *t*) ⇒ *t' s'* = *t' on X* **by** *auto*
thus *?case* **using** ‹~*bval b t*› **by** *auto*
next
case (*WhileFalse b s c*)
hence ~ *bval b t*
by (*metis L-While-unfold UnI1 bval-eq-if-eq-on-vars*)
thus *?case* **using** *WhileFalse.prems L-While-unfold*[*of b c X*] **by** *auto*
next
case (*WhileTrue b s1 c s2 s3 X t1*)
let *?w* = *WHILE b DO c*
from ‹*bval b s1*› *WhileTrue.prems* **have** *bval b t1*
by (*metis L-While-unfold UnI1 bval-eq-if-eq-on-vars*)
have *s1* = *t1 on L c* (*L ?w X*) **using** *L-While-unfold WhileTrue.prems*
by (*blast*)
from *WhileTrue.IH*(*1*)[*OF this*] **obtain** *t2* **where**
(*c*, *t1*) ⇒ *t2 s2* = *t2 on L ?w X* **by** *auto*
from *WhileTrue.IH*(*2*)[*OF this*(*2*)] **obtain** *t3* **where** (*?w*,*t2*) ⇒ *t3 s3*
= *t3 on X*
by *auto*
with ‹*bval b t1*› ‹(*c*, *t1*) ⇒ *t2*› **show** *?case* **by** *auto*

qed

instantiation *com* :: *vars*
begin

fun *vars-com* :: *com* ⇒ *vname set* **where**
vars SKIP = {} |
vars (*x*::=*e*) = *vars e* |
vars (*c*$_1$; *c*$_2$) = *vars* *c*$_1$ ∪ *vars* *c*$_2$ |
vars (*IF b THEN* *c*$_1$ *ELSE* *c*$_2$) = *vars b* ∪ *vars* *c*$_1$ ∪ *vars* *c*$_2$ |
vars (*WHILE b DO c*) = *vars b* ∪ *vars c*

instance ..

end

lemma *L-subset-vars*: *L c X* ⊆ *vars c* ∪ *X*
proof(*induction c arbitrary*: *X*)
case (*While b c*)
have *lfp*(%*Y*. *vars b* ∪ *X* ∪ *L c Y*) ⊆ *vars b* ∪ *vars c* ∪ *X*
using *While.IH*[*of vars b* ∪ *vars c* ∪ *X*]
by (*auto intro*!: *lfp-lowerbound*)
thus *?case* **by** *simp*
qed *auto*

lemma *afinite*[*simp*]: *finite*(*vars*(*a*::*aexp*))
by (*induction a*) *auto*

lemma *bfinite*[*simp*]: *finite*(*vars*(*b*::*bexp*))
by (*induction b*) *auto*

lemma *cfinite*[*simp*]: *finite*(*vars*(*c*::*com*))
by (*induction c*) *auto*

lemma *lfp-finite-iter*: **assumes** *mono f* **and** (*f*^^*Suc k*) *bot* = (*f*^^*k*) *bot*
shows *lfp f* = (*f*^^*k*) *bot*
proof(*rule antisym*)
show *lfp f* ≤ (*f*^^*k*) *bot*
proof(*rule lfp-lowerbound*)
show *f* ((*f*^^*k*) *bot*) ≤ (*f*^^*k*) *bot* **using** *assms*(*2*) **by** *simp*
qed
next

show $(f\hat{}\hat{}k)$ *bot* $\leq$ *lfp f*
proof(*induction k*)
case *0* **show** *?case* **by** *simp*
next
case *Suc*
from *monoD*[*OF assms*(*1*) *Suc*] *lfp-unfold*[*OF assms*(*1*)]
show *?case* **by** *simp*
qed
qed

lemma *while-option-stop2*:
while-option b c s = *Some t* $\Longrightarrow$ *EX k*. *t* = $(c\hat{}\hat{}k)$ *s* $\wedge$ $\neg$ *b t*
apply(*simp add*: *while-option-def split*: *if-splits*)
by (*metis* (*lifting*) *LeastI-ex*)

lemma *while-option-finite-subset-Some*: **fixes** *C* :: *'a set*
assumes *mono f* **and** !!*X*. *X* $\subseteq$ *C* $\Longrightarrow$ *f X* $\subseteq$ *C* **and** *finite C*
shows $\exists$ *P*. *while-option* $(\lambda A.\ f\ A \neq A)$ *f* {} = *Some P*
proof(*rule measure-while-option-Some*[**where**
f= %*A*::*'a set*. *card C* − *card A* **and** *P*= %*A*. *A* $\subseteq$ *C* $\wedge$ *A* $\subseteq$ *f A* **and** *s*= {}])
fix *A* **assume** *A*: *A* $\subseteq$ *C* $\wedge$ *A* $\subseteq$ *f A f A* $\neq$ *A*
show (*f A* $\subseteq$ *C* $\wedge$ *f A* $\subseteq$ *f* (*f A*)) $\wedge$ *card C* − *card* (*f A*) < *card C* − *card A*
(**is** *?L* $\wedge$ *?R*)
proof
show *?L* **by**(*metis A*(*1*) *assms*(*2*) *monoD*[*OF* ‹*mono f*›])
show *?R* **by** (*metis A assms*(*2*,*3*) *card-seteq diff-less-mono2 equalityI linorder-le-less-linear rev-finite-subset*)
qed
qed *simp*

lemma *lfp-eq-while-option*:
assumes *mono f* **and** !!*X*. *X* $\subseteq$ *C* $\Longrightarrow$ *f X* $\subseteq$ *C* **and** *finite C*
shows *lfp f* = *the*(*while-option* $(\lambda A.\ f\ A \neq A)$ *f* {})
proof−
obtain *P* **where** *while-option* $(\lambda A.\ f\ A \neq A)$ *f* {} = *Some P*
using *while-option-finite-subset-Some*[*OF assms*] **by** *blast*
with *while-option-stop2*[*OF this*] *lfp-finite-iter*[*OF assms*(*1*)]
show *?thesis* **by** *auto*
qed

For code generation:

lemma *L-While*: **fixes** *b c X*
assumes *finite X* **defines** $f == \lambda A.\ vars\ b \cup X \cup L\ c\ A$
shows $L\ (WHILE\ b\ DO\ c)\ X = the(while\text{-}option\ (\lambda A.\ f\ A \neq A)\ f\ \{\})$ (**is** $- = ?r$)
proof −
  let $?V = vars\ b \cup vars\ c \cup X$
  have $lfp\ f = ?r$
  proof(*rule lfp-eq-while-option*[**where** $C = ?V$])
    show *mono f* **by**(*simp add*: *f-def mono-union-L*)
  next
    fix *Y* **show** $Y \subseteq ?V \Longrightarrow f\ Y \subseteq ?V$
      unfolding *f-def* **using** *L-subset-vars*[*of c*] **by** *blast*
  next
    show *finite ?V* **using** ⟨*finite X*⟩ **by** *simp*
  qed
  thus *?thesis* **by** (*simp add*: *f-def*)
qed

An approximate computation of the WHILE-case:

fun *iter* :: $('a \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$
where
$iter\ f\ 0\ p\ d = d\ |$
$iter\ f\ (Suc\ n)\ p\ d = (if\ f\ p = p\ then\ p\ else\ iter\ f\ n\ (f\ p)\ d)$

lemma *iter-pfp*:
  $f\ d \leq d \Longrightarrow mono\ f \Longrightarrow x \leq f\ x \Longrightarrow f(iter\ f\ i\ x\ d) \leq iter\ f\ i\ x\ d$
apply(*induction i arbitrary*: *x*)
 apply *simp*
apply (*simp add*: *mono-def*)
done

lemma *iter-While-pfp*:
fixes *b c X W k f*
defines $f == \lambda A.\ vars\ b \cup X \cup L\ c\ A$ **and** $W == vars\ b \cup vars\ c \cup X$
and $P == iter\ f\ k\ \{\}\ W$
shows $f\ P \subseteq P$
proof−
  have $f\ W \subseteq W$ **unfolding** *f-def W-def* **using** *L-subset-vars*[*of c*] **by** *blast*
  have *mono f* **by**(*simp add*: *f-def mono-union-L*)
  from *iter-pfp*[*of f*, *OF* ⟨$f\ W \subseteq W$⟩ ⟨*mono f*⟩ *empty-subsetI*]
  show *?thesis* **by**(*simp add*: *P-def*)
qed

end

9 Security Type Systems

theory *Sec-Type-Expr* **imports** *Big-Step*
begin

9.1 Security Levels and Expressions

type-synonym *level* = *nat*

The security/confidentiality level of each variable is globally fixed for simplicity. For the sake of examples — the general theory does not rely on it! — a variable of length n has security level n:

definition *sec* :: *vname* ⇒ *level* **where**
sec n = *size n*

fun *sec-aexp* :: *aexp* ⇒ *level* **where**
sec-aexp (*N n*) = *0* |
sec-aexp (*V x*) = *sec x* |
sec-aexp (*Plus* a_1 a_2) = *max* (*sec-aexp* a_1) (*sec-aexp* a_2)

fun *sec-bexp* :: *bexp* ⇒ *level* **where**
sec-bexp (*Bc v*) = *0* |
sec-bexp (*Not b*) = *sec-bexp b* |
sec-bexp (*And* b_1 b_2) = *max* (*sec-bexp* b_1) (*sec-bexp* b_2) |
sec-bexp (*Less* a_1 a_2) = *max* (*sec-aexp* a_1) (*sec-aexp* a_2)

abbreviation *eq-le* :: *state* ⇒ *state* ⇒ *level* ⇒ *bool*
((- = - '(≤ -')) [*51*,*51*,*0*] *50*) **where**
s = *s'* (≤ *l*) == (∀ *x*. *sec x* ≤ *l* ⟶ *s x* = *s' x*)

abbreviation *eq-less* :: *state* ⇒ *state* ⇒ *level* ⇒ *bool*
((- = - '(< -')) [*51*,*51*,*0*] *50*) **where**
s = *s'* (< *l*) == (∀ *x*. *sec x* < *l* ⟶ *s x* = *s' x*)

lemma *aval-eq-if-eq-le*:
⟦ s_1 = s_2 (≤ *l*); *sec-aexp a* ≤ *l* ⟧ ⟹ *aval a* s_1 = *aval a* s_2
by (*induct a*) *auto*

lemma *bval-eq-if-eq-le*:
⟦ s_1 = s_2 (≤ *l*); *sec-bexp b* ≤ *l* ⟧ ⟹ *bval b* s_1 = *bval b* s_2
by (*induct b*) (*auto simp add*: *aval-eq-if-eq-le*)

end

theory *Sec-Typing* **imports** *Sec-Type-Expr*
begin

9.2 Syntax Directed Typing

inductive *sec-type* :: *nat* ⇒ *com* ⇒ *bool* ((-/ ⊢ -) [*0*,*0*] *50*) **where**
Skip:
  l ⊢ *SKIP* |
Assign:
  ⟦ *sec x* ≥ *sec-aexp a*; *sec x* ≥ *l* ⟧ ⟹ *l* ⊢ *x* ::= *a* |
Semi:
  ⟦ *l* ⊢ c_1; *l* ⊢ c_2 ⟧ ⟹ *l* ⊢ c_1;c_2 |
If:
  ⟦ *max* (*sec-bexp b*) *l* ⊢ c_1; *max* (*sec-bexp b*) *l* ⊢ c_2 ⟧ ⟹ *l* ⊢ *IF b THEN* c_1 *ELSE* c_2 |
While:
  max (*sec-bexp b*) *l* ⊢ *c* ⟹ *l* ⊢ *WHILE b DO c*

code-pred (*expected-modes*: *i* => *i* => *bool*) *sec-type* **.**

value *0* ⊢ *IF Less* (*V* ''*x1*'') (*V* ''*x*'') *THEN* ''*x1*'' ::= *N 0 ELSE SKIP*
value *1* ⊢ *IF Less* (*V* ''*x1*'') (*V* ''*x*'') *THEN* ''*x*'' ::= *N 0 ELSE SKIP*
value *2* ⊢ *IF Less* (*V* ''*x1*'') (*V* ''*x*'') *THEN* ''*x1*'' ::= *N 0 ELSE SKIP*

inductive-cases [*elim*!]:
  l ⊢ *x* ::= *a* *l* ⊢ c_1;c_2 *l* ⊢ *IF b THEN* c_1 *ELSE* c_2 *l* ⊢ *WHILE b DO c*

An important property: anti-monotonicity.

lemma *anti-mono*: ⟦ *l* ⊢ *c*; *l'* ≤ *l* ⟧ ⟹ *l'* ⊢ *c*
apply(*induction arbitrary*: *l'* *rule*: *sec-type.induct*)
apply (*metis sec-type.intros*(*1*))
apply (*metis le-trans sec-type.intros*(*2*))
apply (*metis sec-type.intros*(*3*))
apply (*metis If le-refl sup-mono sup-nat-def*)
apply (*metis While le-refl sup-mono sup-nat-def*)
done

lemma *confinement*: ⟦ (*c*,*s*) ⇒ *t*; *l* ⊢ *c* ⟧ ⟹ *s* = *t* (< *l*)
proof(*induction rule*: *big-step-induct*)

```
  case Skip thus ?case by simp
next
  case Assign thus ?case by auto
next
  case Semi thus ?case by auto
next
  case (IfTrue b s c1)
  hence max (sec-bexp b) l ⊢ c1 by auto
  hence l ⊢ c1 by (metis le-maxI2 anti-mono)
  thus ?case using IfTrue.IH by metis
next
  case (IfFalse b s c2)
  hence max (sec-bexp b) l ⊢ c2 by auto
  hence l ⊢ c2 by (metis le-maxI2 anti-mono)
  thus ?case using IfFalse.IH by metis
next
  case WhileFalse thus ?case by auto
next
  case (WhileTrue b s1 c)
  hence max (sec-bexp b) l ⊢ c by auto
  hence l ⊢ c by (metis le-maxI2 anti-mono)
  thus ?case using WhileTrue by metis
qed


theorem noninterference:
  ⟦ (c,s) ⇒ s'; (c,t) ⇒ t';  0 ⊢ c;  s = t (≤ l) ⟧
   ⟹ s' = t' (≤ l)
proof(induction arbitrary: t t' rule: big-step-induct)
  case Skip thus ?case by auto
next
  case (Assign x a s)
  have [simp]: t' = t(x := aval a t) using Assign by auto
  have sec x >= sec-aexp a using ‹0 ⊢ x ::= a› by auto
  show ?case
  proof auto
    assume sec x ≤ l
    with ‹sec x >= sec-aexp a› have sec-aexp a ≤ l by arith
    thus aval a s = aval a t
      by (rule aval-eq-if-eq-le[OF ‹s = t (≤ l)›])
  next
    fix y assume y ≠ x sec y ≤ l
    thus s y = t y using ‹s = t (≤ l)› by simp
  qed
```

```
next
  case Semi thus ?case by blast
next
  case (IfTrue b s c1 s' c2)
  have sec-bexp b ⊢ c1 sec-bexp b ⊢ c2 using IfTrue.prems(2) by auto
  show ?case
  proof cases
    assume sec-bexp b ≤ l
    hence s = t (≤ sec-bexp b) using ‹s = t (≤ l)› by auto
    hence bval b t using ‹bval b s› by(simp add: bval-eq-if-eq-le)
    with IfTrue.IH IfTrue.prems(1,3) ‹sec-bexp b ⊢ c1›  anti-mono
    show ?thesis by auto
  next
    assume ¬ sec-bexp b ≤ l
    have 1: sec-bexp b ⊢ IF b THEN c1 ELSE c2
      by(rule sec-type.intros)(simp-all add: ‹sec-bexp b ⊢ c1› ‹sec-bexp b ⊢
c2›)
    from confinement[OF IfTrue.hyps(2) ‹sec-bexp b ⊢ c1›] ‹¬ sec-bexp b ≤
l›
    have s = s' (≤ l) by auto
    moreover
    from confinement[OF IfTrue.prems(1) 1] ‹¬ sec-bexp b ≤ l›
    have t = t' (≤ l) by auto
    ultimately show s' = t' (≤ l) using ‹s = t (≤ l)› by auto
  qed
next
  case (IfFalse b s c2 s' c1)
  have sec-bexp b ⊢ c1 sec-bexp b ⊢ c2 using IfFalse.prems(2) by auto
  show ?case
  proof cases
    assume sec-bexp b ≤ l
    hence s = t (≤ sec-bexp b) using ‹s = t (≤ l)› by auto
    hence ¬ bval b t using ‹¬ bval b s› by(simp add: bval-eq-if-eq-le)
    with IfFalse.IH IfFalse.prems(1,3) ‹sec-bexp b ⊢ c2› anti-mono
    show ?thesis by auto
  next
    assume ¬ sec-bexp b ≤ l
    have 1: sec-bexp b ⊢ IF b THEN c1 ELSE c2
      by(rule sec-type.intros)(simp-all add: ‹sec-bexp b ⊢ c1› ‹sec-bexp b ⊢
c2›)
    from confinement[OF big-step.IfFalse[OF IfFalse(1,2)] 1] ‹¬ sec-bexp b
≤ l›
    have s = s' (≤ l) by auto
    moreover
```

```
    from confinement[OF IfFalse.prems(1) 1] ‹¬ sec-bexp b ≤ l›
    have t = t' (≤ l) by auto
    ultimately show s' = t' (≤ l) using ‹s = t (≤ l)› by auto
  qed
next
  case (WhileFalse b s c)
  have sec-bexp b ⊢ c using WhileFalse.prems(2) by auto
  show ?case
  proof cases
    assume sec-bexp b ≤ l
    hence s = t (≤ sec-bexp b) using ‹s = t (≤ l)› by auto
    hence ¬ bval b t using ‹¬ bval b s› by(simp add: bval-eq-if-eq-le)
    with WhileFalse.prems(1,3) show ?thesis by auto
  next
    assume ¬ sec-bexp b ≤ l
    have 1: sec-bexp b ⊢ WHILE b DO c
      by(rule sec-type.intros)(simp-all add: ‹sec-bexp b ⊢ c›)
    from confinement[OF WhileFalse.prems(1) 1] ‹¬ sec-bexp b ≤ l›
    have t = t' (≤ l) by auto
    thus s = t' (≤ l) using ‹s = t (≤ l)› by auto
  qed
next
  case (WhileTrue b s1 c s2 s3 t1 t3)
  let ?w = WHILE b DO c
  have sec-bexp b ⊢ c using WhileTrue.prems(2) by auto
  show ?case
  proof cases
    assume sec-bexp b ≤ l
    hence s1 = t1 (≤ sec-bexp b) using ‹s1 = t1 (≤ l)› by auto
    hence bval b t1
      using ‹bval b s1› by(simp add: bval-eq-if-eq-le)
    then obtain t2 where (c,t1) ⇒ t2 (?w,t2) ⇒ t3
      using ‹(?w,t1) ⇒ t3› by auto
    from WhileTrue.IH(2)[OF ‹(?w,t2) ⇒ t3› ‹0 ⊢ ?w›
      WhileTrue.IH(1)[OF ‹(c,t1) ⇒ t2› anti-mono[OF ‹sec-bexp b ⊢ c›]
        ‹s1 = t1 (≤ l)›]]
    show ?thesis by simp
  next
    assume ¬ sec-bexp b ≤ l
   have 1: sec-bexp b ⊢ ?w by(rule sec-type.intros)(simp-all add: ‹sec-bexp
b ⊢ c›)
     from confinement[OF big-step.WhileTrue[OF WhileTrue.hyps] 1] ‹¬
sec-bexp b ≤ l›
    have s1 = s3 (≤ l) by auto
```

moreover
from *confinement*[*OF WhileTrue.prems*(*1*) *1*] ‹¬ *sec-bexp b* ≤ *l*›
have *t1* = *t3* (≤ *l*) **by** *auto*
ultimately show *s3* = *t3* (≤ *l*) **using** ‹*s1* = *t1* (≤ *l*)› **by** *auto*
qed
qed

9.3 The Standard Typing System

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

inductive *sec-type′* :: *nat* ⇒ *com* ⇒ *bool* ((-/ ⊢′′ -) [*0*,*0*] *50*) **where**
Skip′:
$l \vdash' SKIP$ |
Assign′:
⟦ *sec x* ≥ *sec-aexp a*; *sec x* ≥ *l* ⟧ ⟹ $l \vdash' x ::= a$ |
Semi′:
⟦ $l \vdash' c_1$; $l \vdash' c_2$ ⟧ ⟹ $l \vdash' c_1;c_2$ |
If′:
⟦ *sec-bexp b* ≤ *l*; $l \vdash' c_1$; $l \vdash' c_2$ ⟧ ⟹ $l \vdash' IF\ b\ THEN\ c_1\ ELSE\ c_2$ |
While′:
⟦ *sec-bexp b* ≤ *l*; $l \vdash' c$ ⟧ ⟹ $l \vdash' WHILE\ b\ DO\ c$ |
anti-mono′:
⟦ $l \vdash' c$; $l' \le l$ ⟧ ⟹ $l' \vdash' c$

lemma *sec-type-sec-type′*: $l \vdash c \Longrightarrow l \vdash' c$
apply(*induction rule*: *sec-type.induct*)
apply (*metis Skip′*)
apply (*metis Assign′*)
apply (*metis Semi′*)
apply (*metis min-max.inf-sup-ord*(*3*) *min-max.sup-absorb2 nat-le-linear If′ anti-mono′*)
by (*metis less-or-eq-imp-le min-max.sup-absorb1 min-max.sup-absorb2 nat-le-linear While′ anti-mono′*)

lemma *sec-type′-sec-type*: $l \vdash' c \Longrightarrow l \vdash c$
apply(*induction rule*: *sec-type′.induct*)
apply (*metis Skip*)
apply (*metis Assign*)
apply (*metis Semi*)
apply (*metis min-max.sup-absorb2 If*)

apply (*metis min-max.sup-absorb2 While*)
by (*metis anti-mono*)

9.4 A Bottom-Up Typing System

inductive *sec-type2* :: *com* ⇒ *level* ⇒ *bool* ((⊢ - : -) [*0*,*0*] *50*) **where**
Skip2:
⊢ *SKIP* : *l* |
Assign2:
sec x ≥ *sec-aexp a* ⟹ ⊢ *x* ::= *a* : *sec x* |
Semi2:
⟦ ⊢ c_1 : l_1; ⊢ c_2 : l_2 ⟧ ⟹ ⊢ c_1;c_2 : *min* l_1 l_2 |
If2:
⟦ *sec-bexp b* ≤ *min* l_1 l_2; ⊢ c_1 : l_1; ⊢ c_2 : l_2 ⟧
⟹ ⊢ *IF b THEN* c_1 *ELSE* c_2 : *min* l_1 l_2 |
While2:
⟦ *sec-bexp b* ≤ *l*; ⊢ *c* : *l* ⟧ ⟹ ⊢ *WHILE b DO c* : *l*

lemma *sec-type2-sec-type'*: ⊢ *c* : *l* ⟹ *l* ⊢' *c*
apply(*induction rule*: *sec-type2.induct*)
apply (*metis Skip'*)
apply (*metis Assign' eq-imp-le*)
apply (*metis Semi' anti-mono' min-max.inf.commute min-max.inf-le2*)
apply (*metis If' anti-mono' min-max.inf-absorb2 min-max.le-iff-inf nat-le-linear*)
by (*metis While'*)

lemma *sec-type'-sec-type2*: *l* ⊢' *c* ⟹ ∃ *l'* ≥ *l*. ⊢ *c* : *l'*
apply(*induction rule*: *sec-type'.induct*)
apply (*metis Skip2 le-refl*)
apply (*metis Assign2*)
apply (*metis Semi2 min-max.inf-greatest*)
apply (*metis If2 inf-greatest inf-nat-def le-trans*)
apply (*metis While2 le-trans*)
by (*metis le-trans*)

end

theory *Sec-TypingT* **imports** *Sec-Type-Expr*
begin

9.5 A Termination-Sensitive Syntax Directed System

inductive *sec-type* :: *nat* ⇒ *com* ⇒ *bool* ((-/ ⊢ -) [*0*,*0*] *50*) **where**

Skip:
$l \vdash SKIP$ |
Assign:
$[\![sec\ x \geq sec\text{-}aexp\ a;\ \ sec\ x \geq l\ ]\!] \Longrightarrow l \vdash x ::= a$ |
Semi:
$l \vdash c_1 \Longrightarrow l \vdash c_2 \Longrightarrow l \vdash c_1;c_2$ |
If:
$[\![max\ (sec\text{-}bexp\ b)\ l \vdash c_1;\ \ max\ (sec\text{-}bexp\ b)\ l \vdash c_2\ ]\!]$
$\Longrightarrow l \vdash IF\ b\ THEN\ c_1\ ELSE\ c_2$ |
While:
$sec\text{-}bexp\ b = 0 \Longrightarrow 0 \vdash c \Longrightarrow 0 \vdash WHILE\ b\ DO\ c$

code-pred (*expected-modes*: $i \Rightarrow i \Rightarrow bool$) *sec-type* **.**

inductive-cases [*elim*!]:
$l \vdash x ::= a$ $l \vdash c_1;c_2$ $l \vdash IF\ b\ THEN\ c_1\ ELSE\ c_2$ $l \vdash WHILE\ b\ DO\ c$

lemma *anti-mono*: $l \vdash c \Longrightarrow l' \leq l \Longrightarrow l' \vdash c$
apply(*induction arbitrary*: l' *rule*: *sec-type.induct*)
apply (*metis sec-type.intros*(*1*))
apply (*metis le-trans sec-type.intros*(*2*))
apply (*metis sec-type.intros*(*3*))
by (*metis If le-refl sup-mono sup-nat-def*)

lemma *confinement*: $(c,s) \Rightarrow t \Longrightarrow l \vdash c \Longrightarrow s = t\ (< l)$
proof(*induction rule*: *big-step-induct*)
case *Skip* **thus** *?case* **by** *simp*
next
case *Assign* **thus** *?case* **by** *auto*
next
case *Semi* **thus** *?case* **by** *auto*
next
case (*IfTrue b s c1*)
hence *max* (*sec-bexp b*) $l \vdash$ *c1* **by** *auto*
hence $l \vdash$ *c1* **by** (*metis le-maxI2 anti-mono*)
thus *?case* **using** *IfTrue.IH* **by** *metis*
next
case (*IfFalse b s c2*)
hence *max* (*sec-bexp b*) $l \vdash$ *c2* **by** *auto*
hence $l \vdash$ *c2* **by** (*metis le-maxI2 anti-mono*)
thus *?case* **using** *IfFalse.IH* **by** *metis*

next
case *WhileFalse* **thus** *?case* **by** *auto*
next
case (*WhileTrue b s1 c*)
hence $l \vdash c$ **by** *auto*
thus *?case* **using** *WhileTrue* **by** *metis*
qed

lemma *termi-if-non0*: $l \vdash c \Longrightarrow l \neq 0 \Longrightarrow \exists\ t.\ (c,s) \Rightarrow t$
apply(*induction arbitrary*: *s rule*: *sec-type.induct*)
apply (*metis big-step.Skip*)
apply (*metis big-step.Assign*)
apply (*metis big-step.Semi*)
apply (*metis IfFalse IfTrue le0 le-antisym le-maxI2*)
apply *simp*
done

theorem *noninterference*: $(c,s) \Rightarrow s' \Longrightarrow 0 \vdash c \Longrightarrow\ s = t\ (\le l)$
$\Longrightarrow \exists\ t'.\ (c,t) \Rightarrow t' \wedge s' = t'\ (\le l)$
proof(*induction arbitrary*: *t rule*: *big-step-induct*)
case *Skip* **thus** *?case* **by** *auto*
next
case (*Assign x a s*)
have *sec x >= sec-aexp a* **using** ‹$0 \vdash x ::= a$› **by** *auto*
have $(x ::= a,t) \Rightarrow t(x := aval\ a\ t)$ **by** *auto*
moreover
have $s(x := aval\ a\ s) = t(x := aval\ a\ t)\ (\le l)$
proof *auto*
assume $sec\ x \le l$
with ‹*sec x* $\ge$ *sec-aexp a*› **have** *sec-aexp a* $\le l$ **by** *arith*
thus $aval\ a\ s = aval\ a\ t$
by (*rule aval-eq-if-eq-le*[*OF* ‹$s = t\ (\le l)$›])
next
fix *y* **assume** $y \neq x\ sec\ y \le l$
thus $s\ y = t\ y$ **using** ‹$s = t\ (\le l)$› **by** *simp*
qed
ultimately show *?case* **by** *blast*
next
case *Semi* **thus** *?case* **by** *blast*
next
case (*IfTrue b s c1 s' c2*)
have *sec-bexp b* $\vdash$ *c1 sec-bexp b* $\vdash$ *c2* **using** *IfTrue.prems* **by** *auto*
obtain t' **where** t': $(c1,\ t) \Rightarrow t'\ s' = t'\ (\le l)$
using *IfTrue*(*3*)[*OF anti-mono*[*OF* ‹*sec-bexp b* $\vdash$ *c1*›] *IfTrue.prems*(*2*)]

by *blast*
show *?case*
proof *cases*
assume *sec-bexp b* $\leq$ *l*
hence *s* = *t* (*sec-bexp b*) **using** ‹*s* = *t* (*l*)› **by** *auto*
hence *bval b t* **using** ‹*bval b s*› **by**(*simp add*: *bval-eq-if-eq-le*)
thus *?thesis* **by** (*metis t' big-step.IfTrue*)
next
assume ¬ *sec-bexp b* $\leq$ *l*
hence *0*: *sec-bexp b* $\neq$ *0* **by** *arith*
have *1*: *sec-bexp b* ⊢ *IF b THEN c1 ELSE c2*
by(*rule sec-type.intros*)(*simp-all add*: ‹*sec-bexp b* ⊢ *c1*› ‹*sec-bexp b* ⊢ *c2*›)
from *confinement*[*OF big-step.IfTrue*[*OF IfTrue*(*1*,*2*)] *1*] ‹¬ *sec-bexp b* $\leq$ *l*›
have *s* = *s'* (*l*) **by** *auto*
moreover
from *termi-if-non0*[*OF 1 0*, *of t*] **obtain** *t'* **where**
(*IF b THEN c1 ELSE c2*,*t*) ⇒ *t'* **..**
moreover
from *confinement*[*OF this 1*] ‹¬ *sec-bexp b* $\leq$ *l*›
have *t* = *t'* (*l*) **by** *auto*
ultimately
show *?case* **using** ‹*s* = *t* (*l*)› **by** *auto*
qed
next
case (*IfFalse b s c2 s' c1*)
have *sec-bexp b* ⊢ *c1 sec-bexp b* ⊢ *c2* **using** *IfFalse.prems* **by** *auto*
obtain *t'* **where** *t'*: (*c2*, *t*) ⇒ *t' s'* = *t'* (*l*)
using *IfFalse*(*3*)[*OF anti-mono*[*OF* ‹*sec-bexp b* ⊢ *c2*›] *IfFalse.prems*(*2*)]
by *blast*
show *?case*
proof *cases*
assume *sec-bexp b* $\leq$ *l*
hence *s* = *t* (*sec-bexp b*) **using** ‹*s* = *t* (*l*)› **by** *auto*
hence ¬ *bval b t* **using** ‹¬ *bval b s*› **by**(*simp add*: *bval-eq-if-eq-le*)
thus *?thesis* **by** (*metis t' big-step.IfFalse*)
next
assume ¬ *sec-bexp b* $\leq$ *l*
hence *0*: *sec-bexp b* $\neq$ *0* **by** *arith*
have *1*: *sec-bexp b* ⊢ *IF b THEN c1 ELSE c2*
by(*rule sec-type.intros*)(*simp-all add*: ‹*sec-bexp b* ⊢ *c1*› ‹*sec-bexp b* ⊢ *c2*›)
from *confinement*[*OF big-step.IfFalse*[*OF IfFalse*(*1*,*2*)] *1*] ‹¬ *sec-bexp b*

```
≤ l⟩
    have s = s' (≤ l) by auto
    moreover
    from termi-if-non0[OF 1 0, of t] obtain t' where
      (IF b THEN c1 ELSE c2,t) ⇒ t' ..
    moreover
    from confinement[OF this 1] ⟨¬ sec-bexp b ≤ l⟩
    have t = t' (≤ l) by auto
    ultimately
    show ?case using ⟨s = t (≤ l)⟩ by auto
  qed
next
  case (WhileFalse b s c)
  hence [simp]: sec-bexp b = 0 by auto
  have s = t (≤ sec-bexp b) using ⟨s = t (≤ l)⟩ by auto
  hence ¬ bval b t using ⟨¬ bval b s⟩ by (metis bval-eq-if-eq-le le-refl)
  with WhileFalse.prems(2) show ?case by auto
next
  case (WhileTrue b s c s'' s')
  let ?w = WHILE b DO c
  from ⟨0 ⊢ ?w⟩ have [simp]: sec-bexp b = 0 by auto
  have 0 ⊢ c using WhileTrue.prems(1) by auto
  from WhileTrue.IH(1)[OF this WhileTrue.prems(2)]
  obtain t'' where (c,t) ⇒ t'' and s'' = t'' (≤l) by blast
  from WhileTrue.IH(2)[OF ⟨0 ⊢ ?w⟩ this(2)]
  obtain t' where (?w,t'') ⇒ t' and s' = t' (≤l) by blast
  from ⟨bval b s⟩ have bval b t
    using bval-eq-if-eq-le[OF ⟨s = t (≤l)⟩] by auto
  show ?case
    using big-step.WhileTrue[OF ⟨bval b t⟩ ⟨(c,t) ⇒ t''⟩ ⟨(?w,t'') ⇒ t'⟩]
    by (metis ⟨s' = t' (≤ l)⟩)
qed
```

9.6 The Standard Termination-Sensitive System

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

```
inductive sec-type' :: nat ⇒ com ⇒ bool ((-/ ⊢'' -) [0,0] 50) where
Skip':
  l ⊢' SKIP  |
Assign':
  ⟦ sec x ≥ sec-aexp a;  sec x ≥ l ⟧ ⟹ l ⊢' x ::= a  |
```

Semi':
$l \vdash' c_1 \Longrightarrow l \vdash' c_2 \Longrightarrow l \vdash' c_1;c_2$ |
If':
$[\![\textit{sec-bexp } b \leq l;\ l \vdash' c_1;\ l \vdash' c_2]\!] \Longrightarrow l \vdash' \textit{IF } b \textit{ THEN } c_1 \textit{ ELSE } c_2$ |
While':
$[\![\textit{sec-bexp } b = 0;\ 0 \vdash' c]\!] \Longrightarrow 0 \vdash' \textit{WHILE } b \textit{ DO } c$ |
anti-mono':
$[\![l \vdash' c;\ l' \leq l]\!] \Longrightarrow l' \vdash' c$

lemma $l \vdash c \Longrightarrow l \vdash' c$
apply(*induction rule*: *sec-type.induct*)
apply (*metis Skip'*)
apply (*metis Assign'*)
apply (*metis Semi'*)
apply (*metis min-max.inf-sup-ord*(*3*) *min-max.sup-absorb2 nat-le-linear If' anti-mono'*)
by (*metis While'*)

lemma $l \vdash' c \Longrightarrow l \vdash c$
apply(*induction rule*: *sec-type'.induct*)
apply (*metis Skip*)
apply (*metis Assign*)
apply (*metis Semi*)
apply (*metis min-max.sup-absorb2 If*)
apply (*metis While*)
by (*metis anti-mono*)

end

10 Hoare Logic

theory *Hoare* **imports** *Big-Step* **begin**

10.1 Hoare Logic for Partial Correctness

type-synonym *assn* = *state* ⇒ *bool*

abbreviation *state-subst* :: *state* ⇒ *aexp* ⇒ *vname* ⇒ *state*
(-[-'/-] [*1000*,*0*,*0*] *999*)
where $s[a/x] == s(x := \textit{aval } a\ s)$

inductive

$hoare :: assn \Rightarrow com \Rightarrow assn \Rightarrow bool\ (\vdash (\{(1\text{-})\}/\ (\text{-})/\ \{(1\text{-})\})\ 50)$
where
Skip: $\vdash \{P\}\ SKIP\ \{P\}$ |

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

Semi: $[\![\vdash \{P\}\ c_1\ \{Q\};\ \vdash \{Q\}\ c_2\ \{R\}]\!]$
$\Longrightarrow \vdash \{P\}\ c_1;c_2\ \{R\}$ |

If: $[\![\vdash \{\lambda s.\ P\ s \wedge bval\ b\ s\}\ c_1\ \{Q\};\ \vdash \{\lambda s.\ P\ s \wedge \neg\ bval\ b\ s\}\ c_2\ \{Q\}]\!]$
$\Longrightarrow \vdash \{P\}\ IF\ b\ THEN\ c_1\ ELSE\ c_2\ \{Q\}$ |

While: $\vdash \{\lambda s.\ P\ s \wedge bval\ b\ s\}\ c\ \{P\} \Longrightarrow$
$\vdash \{P\}\ WHILE\ b\ DO\ c\ \{\lambda s.\ P\ s \wedge \neg\ bval\ b\ s\}$ |

conseq: $[\![\forall s.\ P'\ s \longrightarrow P\ s;\ \vdash \{P\}\ c\ \{Q\};\ \forall s.\ Q\ s \longrightarrow Q'\ s]\!]$
$\Longrightarrow \vdash \{P'\}\ c\ \{Q'\}$

lemmas [*simp*] = *hoare.Skip hoare.Assign hoare.Semi If*

lemmas [*intro*!] = *hoare.Skip hoare.Assign hoare.Semi hoare.If*

lemma *strengthen-pre*:
$[\![\forall s.\ P'\ s \longrightarrow P\ s;\ \vdash \{P\}\ c\ \{Q\}]\!] \Longrightarrow \vdash \{P'\}\ c\ \{Q\}$
by (*blast intro*: *conseq*)

lemma *weaken-post*:
$[\![\vdash \{P\}\ c\ \{Q\};\ \forall s.\ Q\ s \longrightarrow Q'\ s]\!] \Longrightarrow\ \vdash \{P\}\ c\ \{Q'\}$
by (*blast intro*: *conseq*)

The assignment and While rule are awkward to use in actual proofs because their pre and postcondition are of a very special form and the actual goal would have to match this form exactly. Therefore we derive two variants with arbitrary pre and postconditions.

lemma *Assign'*: $\forall s.\ P\ s \longrightarrow Q(s[a/x]) \Longrightarrow \vdash \{P\}\ x ::= a\ \{Q\}$
by (*simp add*: *strengthen-pre*[*OF - Assign*])

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

end

theory *Hoare-Examples* **imports** *Hoare* **begin**

10.2 Example: Sums

Summing up the first n natural numbers. The sum is accumulated in variable *0*, the loop counter is variable *1*.

abbreviation *w n* ==
  WHILE Less (*V* ''*y*'') (*N n*)
  DO (''*y*'' ::= *Plus* (*V* ''*y*'') (*N 1*); ''*x*'' ::= *Plus* (*V* ''*x*'') (*V* ''*y*''))

For this example we make use of some predefined functions. Function *Setsum*, also written $\sum$, sums up the elements of a set. The set of numbers from m to n is written $\{m..n\}$.

10.2.1 Proof by Operational Semantics

The behaviour of the loop is proved by induction:

lemma *setsum-head-plus-1*:
  $m \leq n \Longrightarrow$ *setsum f* $\{m..n\}$ = *f m* + *setsum f* $\{m+1..n::int\}$
by (*subst simp-from-to*) *simp*

lemma *while-sum*:
  (*w n*, *s*) $\Rightarrow$ *t* $\Longrightarrow$ *t* ''*x*'' = *s* ''*x*'' + $\sum$ {*s* ''*y*'' + *1* .. *n*}
apply(*induction w n s t rule*: *big-step-induct*)
apply(*auto simp add*: *setsum-head-plus-1*)
done

We were lucky that the proof was practically automatic, except for the induction. In general, such proofs will not be so easy. The automation is partly due to the right inversion rules that we set up as automatic elimination rules that decompose big-step premises.

Now we prefix the loop with the necessary initialization:

lemma *sum-via-bigstep*:
assumes (''*x*'' ::= *N 0*; ''*y*'' ::= *N 0*; *w n*, *s*) $\Rightarrow$ *t*
shows *t* ''*x*'' = $\sum$ {*1* .. *n*}
proof −
  from *assms* **have** (*w n*,*s*(''*x*'':=*0*,''*y*'':=*0*)) $\Rightarrow$ *t* **by** *auto*
  from *while-sum*[*OF this*] **show** *?thesis* **by** *simp*
qed

10.2.2 Proof by Hoare Logic

Note that we deal with sequences of commands from right to left, pulling back the postcondition towards the precondition.

lemma $\vdash \{\lambda s.\ 0 <= n\}\ ''x'' ::= N\ 0;\ ''y'' ::= N\ 0;\ w\ n\ \{\lambda s.\ s\ ''x'' = \sum \{1\ ..\ n\}\}$
apply(*rule hoare.Semi*)
prefer *2*
apply(*rule While'*
[**where** $P = \lambda s.\ s\ ''x'' = \sum\ \{1..s\ ''y''\} \wedge 0 \leq s\ ''y'' \wedge s\ ''y'' \leq n$])
apply(*rule Semi*)
prefer *2*
apply(*rule Assign*)
apply(*rule Assign'*)
apply(*fastforce simp*: *atLeastAtMostPlus1-int-conv algebra-simps*)
apply(*fastforce*)
apply(*rule Semi*)
prefer *2*
apply(*rule Assign*)
apply(*rule Assign'*)
apply *simp*
done

The proof is intentionally an apply skript because it merely composes the rules of Hoare logic. Of course, in a few places side conditions have to be proved. But since those proofs are 1-liners, a structured proof is overkill. In fact, we shall learn later that the application of the Hoare rules can be automated completely and all that is left for the user is to provide the loop invariants and prove the side-conditions.

end

theory *Hoare-Sound-Complete* **imports** *Hoare* **begin**

10.3 Soundness

definition
hoare-valid :: *assn* ⇒ *com* ⇒ *assn* ⇒ *bool* (⊨ {(*1-*)}/ (-)/ {(*1-*)} *50*) **where**
$\models \{P\}c\{Q\} = (\forall s\ t.\ (c,s) \Rightarrow t\ \longrightarrow\ P\ s \longrightarrow\ Q\ t)$

lemma *hoare-sound*: $\vdash \{P\}c\{Q\}\ \Longrightarrow\ \models \{P\}c\{Q\}$
proof(*induction rule*: *hoare.induct*)
case (*While P b c*)

{ fix *s t*
have (*WHILE b DO c*,*s*) ⇒ *t* ⟹ *P s* ⟶ *P t* ∧ ¬ *bval b t*
proof(*induction WHILE b DO c s t rule*: *big-step-induct*)
case *WhileFalse* **thus** *?case* **by** *blast*
next
case *WhileTrue* **thus** *?case*
using *While*(*2*) **unfolding** *hoare-valid-def* **by** *blast*
qed
}
thus *?case* **unfolding** *hoare-valid-def* **by** *blast*
qed (*auto simp*: *hoare-valid-def*)

10.4 Weakest Precondition

definition *wp* :: *com* ⇒ *assn* ⇒ *assn* **where**
wp c Q = (λ*s*. ∀ *t*. (*c*,*s*) ⇒ *t* ⟶ *Q t*)

lemma *wp-SKIP*[*simp*]: *wp SKIP Q* = *Q*
by (*rule ext*) (*auto simp*: *wp-def*)

lemma *wp-Ass*[*simp*]: *wp* (*x*::=*a*) *Q* = (λ*s*. *Q*(*s*[*a*/*x*]))
by (*rule ext*) (*auto simp*: *wp-def*)

lemma *wp-Semi*[*simp*]: *wp* (*c*$_1$;*c*$_2$) *Q* = *wp c*$_1$ (*wp c*$_2$ *Q*)
by (*rule ext*) (*auto simp*: *wp-def*)

lemma *wp-If*[*simp*]:
wp (*IF b THEN c*$_1$ *ELSE c*$_2$) *Q* =
(λ*s*. (*bval b s* ⟶ *wp c*$_1$ *Q s*) ∧ (¬ *bval b s* ⟶ *wp c*$_2$ *Q s*))
by (*rule ext*) (*auto simp*: *wp-def*)

lemma *wp-While-If*:
wp (*WHILE b DO c*) *Q s* =
wp (*IF b THEN c*;*WHILE b DO c ELSE SKIP*) *Q s*
unfolding *wp-def* **by** (*metis unfold-while*)

lemma *wp-While-True*[*simp*]: *bval b s* ⟹
wp (*WHILE b DO c*) *Q s* = *wp* (*c*; *WHILE b DO c*) *Q s*
by(*simp add*: *wp-While-If*)

lemma *wp-While-False*[*simp*]: ¬ *bval b s* ⟹ *wp* (*WHILE b DO c*) *Q s* = *Q s*
by(*simp add*: *wp-While-If*)

10.5 Completeness

lemma *wp-is-pre*: $\vdash \{wp\ c\ Q\}\ c\ \{Q\}$
proof(*induction c arbitrary*: *Q*)
  case *Semi* **thus** *?case* **by**(*auto intro*: *Semi*)
next
  case (*If b c1 c2*)
  let *?If* = *IF b THEN c1 ELSE c2*
  show *?case*
  proof(*rule hoare.If*)
    show $\vdash \{\lambda s.\ wp\ ?If\ Q\ s \wedge bval\ b\ s\}\ c1\ \{Q\}$
    proof(*rule strengthen-pre*[*OF - If*(*1*)])
      show $\forall s.\ wp\ ?If\ Q\ s \wedge bval\ b\ s \longrightarrow wp\ c1\ Q\ s$ **by** *auto*
    qed
    show $\vdash \{\lambda s.\ wp\ ?If\ Q\ s \wedge \neg\ bval\ b\ s\}\ c2\ \{Q\}$
    proof(*rule strengthen-pre*[*OF - If*(*2*)])
      show $\forall s.\ wp\ ?If\ Q\ s \wedge \neg\ bval\ b\ s \longrightarrow wp\ c2\ Q\ s$ **by** *auto*
    qed
  qed
next
  case (*While b c*)
  let *?w* = *WHILE b DO c*
  have $\vdash \{wp\ ?w\ Q\}\ ?w\ \{\lambda s.\ wp\ ?w\ Q\ s \wedge \neg\ bval\ b\ s\}$
  proof(*rule hoare.While*)
    show $\vdash \{\lambda s.\ wp\ ?w\ Q\ s \wedge bval\ b\ s\}\ c\ \{wp\ ?w\ Q\}$
    proof(*rule strengthen-pre*[*OF - While*(*1*)])
      show $\forall s.\ wp\ ?w\ Q\ s \wedge bval\ b\ s \longrightarrow wp\ c\ (wp\ ?w\ Q)\ s$ **by** *auto*
    qed
  qed
  thus *?case*
  proof(*rule weaken-post*)
    show $\forall s.\ wp\ ?w\ Q\ s \wedge \neg\ bval\ b\ s \longrightarrow Q\ s$ **by** *auto*
  qed
qed *auto*

lemma *hoare-relative-complete*: **assumes** $\models \{P\}c\{Q\}$ **shows** $\vdash \{P\}c\{Q\}$
proof(*rule strengthen-pre*)
  show $\forall s.\ P\ s \longrightarrow wp\ c\ Q\ s$ **using** *assms*
    by (*auto simp*: *hoare-valid-def wp-def*)
  show $\vdash \{wp\ c\ Q\}\ c\ \{Q\}$ **by**(*rule wp-is-pre*)
qed

end

11 Verification Conditions

theory *VC* **imports** *Hoare* **begin**

11.1 VCG via Weakest Preconditions

Annotated commands: commands where loops are annotated with invariants.

datatype *acom* =
  ASKIP |
  Aassign vname aexp ((- ::= -) [*1000*, *61*] *61*) |
  Asemi acom acom (-;/ - [*60*, *61*] *60*) |
  Aif bexp acom acom ((*IF* -/ *THEN* -/ *ELSE* -) [*0*, *0*, *61*] *61*) |
  Awhile assn bexp acom (({-}/ *WHILE* -/ *DO* -) [*0*, *0*, *61*] *61*)

Weakest precondition from annotated commands:

fun *pre* :: *acom* ⇒ *assn* ⇒ *assn* **where**
pre ASKIP Q = *Q* |
pre (*Aassign x a*) *Q* = (λ*s*. *Q*(*s*(*x* := *aval a s*))) |
pre (*Asemi* c_1 c_2) *Q* = *pre* c_1 (*pre* c_2 *Q*) |
pre (*Aif b* c_1 c_2) *Q* =
  (λ*s*. (*bval b s* ⟶ *pre* c_1 *Q s*) ∧
    (¬ *bval b s* ⟶ *pre* c_2 *Q s*)) |
pre (*Awhile I b c*) *Q* = *I*

Verification condition:

fun *vc* :: *acom* ⇒ *assn* ⇒ *assn* **where**
vc ASKIP Q = (λ*s*. *True*) |
vc (*Aassign x a*) *Q* = (λ*s*. *True*) |
vc (*Asemi* c_1 c_2) *Q* = (λ*s*. *vc* c_1 (*pre* c_2 *Q*) *s* ∧ *vc* c_2 *Q s*) |
vc (*Aif b* c_1 c_2) *Q* = (λ*s*. *vc* c_1 *Q s* ∧ *vc* c_2 *Q s*) |
vc (*Awhile I b c*) *Q* =
  (λ*s*. (*I s* ∧ ¬ *bval b s* ⟶ *Q s*) ∧
    (*I s* ∧ *bval b s* ⟶ *pre c I s*) ∧
    vc c I s)

Strip annotations:

fun *strip* :: *acom* ⇒ *com* **where**
strip ASKIP = *SKIP* |
strip (*Aassign x a*) = (*x*::=*a*) |
strip (*Asemi* c_1 c_2) = (*strip* c_1; *strip* c_2) |
strip (*Aif b* c_1 c_2) = (*IF b THEN strip* c_1 *ELSE strip* c_2) |
strip (*Awhile I b c*) = (*WHILE b DO strip c*)

11.2 Soundness

lemma *vc-sound*: $\forall s.\ vc\ c\ Q\ s \Longrightarrow \vdash \{pre\ c\ Q\}\ strip\ c\ \{Q\}$
proof(*induction c arbitrary*: *Q*)
case (*Awhile I b c*)
show *?case*
proof(*simp*, *rule While'*)
from ‹$\forall s.\ vc\ (Awhile\ I\ b\ c)\ Q\ s$›
have *vc*: $\forall s.\ vc\ c\ I\ s$ **and** *IQ*: $\forall s.\ I\ s \wedge \neg\ bval\ b\ s \longrightarrow Q\ s$ **and**
pre: $\forall s.\ I\ s \wedge bval\ b\ s \longrightarrow pre\ c\ I\ s$ **by** *simp-all*
have $\vdash \{pre\ c\ I\}\ strip\ c\ \{I\}$ **by**(*rule Awhile.IH*[*OF vc*])
with *pre* **show** $\vdash \{\lambda s.\ I\ s \wedge bval\ b\ s\}\ strip\ c\ \{I\}$
by(*rule strengthen-pre*)
show $\forall s.\ I\ s \wedge \neg bval\ b\ s \longrightarrow Q\ s$ **by**(*rule IQ*)
qed
qed (*auto intro*: *hoare.conseq*)

corollary *vc-sound'*:
$(\forall s.\ vc\ c\ Q\ s) \wedge (\forall s.\ P\ s \longrightarrow pre\ c\ Q\ s) \Longrightarrow \vdash \{P\}\ strip\ c\ \{Q\}$
by (*metis strengthen-pre vc-sound*)

11.3 Completeness

lemma *pre-mono*:
$\forall s.\ P\ s \longrightarrow P'\ s \Longrightarrow pre\ c\ P\ s \Longrightarrow pre\ c\ P'\ s$
proof (*induction c arbitrary*: *P P' s*)
case *Asemi* **thus** *?case* **by** *simp metis*
qed *simp-all*

lemma *vc-mono*:
$\forall s.\ P\ s \longrightarrow P'\ s \Longrightarrow vc\ c\ P\ s \Longrightarrow vc\ c\ P'\ s$
proof(*induction c arbitrary*: *P P'*)
case *Asemi* **thus** *?case* **by** *simp* (*metis pre-mono*)
qed *simp-all*

lemma *vc-complete*:
$\vdash \{P\}c\{Q\} \Longrightarrow \exists c'.\ strip\ c' = c \wedge (\forall s.\ vc\ c'\ Q\ s) \wedge (\forall s.\ P\ s \longrightarrow pre\ c'\ Q\ s)$
(**is** - $\Longrightarrow \exists c'.\ ?G\ P\ c\ Q\ c'$)
proof (*induction rule*: *hoare.induct*)
case *Skip*
show *?case* (**is** $\exists ac.\ ?C\ ac$)
proof show *?C ASKIP* **by** *simp* **qed**
next

case (*Assign P a x*)
show *?case* (**is** ∃ *ac*. *?C ac*)
proof show *?C*(*Aassign x a*) **by** *simp* **qed**
next
case (*Semi P c1 Q c2 R*)
from *Semi.IH* **obtain** *ac1* **where** *ih1*: *?G P c1 Q ac1* **by** *blast*
from *Semi.IH* **obtain** *ac2* **where** *ih2*: *?G Q c2 R ac2* **by** *blast*
show *?case* (**is** ∃ *ac*. *?C ac*)
proof
show *?C*(*Asemi ac1 ac2*)
using *ih1 ih2* **by** (*fastforce elim*!: *pre-mono vc-mono*)
qed
next
case (*If P b c1 Q c2*)
from *If.IH* **obtain** *ac1* **where** *ih1*: *?G* (λ*s*. *P s* ∧ *bval b s*) *c1 Q ac1*
by *blast*
from *If.IH* **obtain** *ac2* **where** *ih2*: *?G* (λ*s*. *P s* ∧ ¬*bval b s*) *c2 Q ac2*
by *blast*
show *?case* (**is** ∃ *ac*. *?C ac*)
proof
show *?C*(*Aif b ac1 ac2*) **using** *ih1 ih2* **by** *simp*
qed
next
case (*While P b c*)
from *While.IH* **obtain** *ac* **where** *ih*: *?G* (λ*s*. *P s* ∧ *bval b s*) *c P ac* **by** *blast*
show *?case* (**is** ∃ *ac*. *?C ac*)
proof show *?C*(*Awhile P b ac*) **using** *ih* **by** *simp* **qed**
next
case *conseq* **thus** *?case* **by**(*fast elim*!: *pre-mono vc-mono*)
qed

11.4 An Optimization

fun *vcpre* :: *acom* ⇒ *assn* ⇒ *assn* × *assn* **where**
vcpre ASKIP Q = (λ*s*. *True*, *Q*) |
vcpre (*Aassign x a*) *Q* = (λ*s*. *True*, λ*s*. *Q*(*s*[*a*/*x*])) |
vcpre (*Asemi* c_1 c_2) *Q* =
(*let* (vc_2,wp_2) = *vcpre* c_2 *Q*;
(vc_1,wp_1) = *vcpre* c_1 wp_2
in (λ*s*. vc_1 *s* ∧ vc_2 *s*, wp_1)) |
vcpre (*Aif b* c_1 c_2) *Q* =
(*let* (vc_2,wp_2) = *vcpre* c_2 *Q*;
(vc_1,wp_1) = *vcpre* c_1 *Q*

in ($\lambda s.\ vc_1\ s \wedge vc_2\ s,\ \lambda s.\ (bval\ b\ s \longrightarrow wp_1\ s) \wedge (\neg bval\ b\ s \longrightarrow wp_2\ s)$))
|
vcpre (*Awhile I b c*) *Q* =
(*let* (*vcc*,*wpc*) = *vcpre c I*
in ($\lambda s.\ (I\ s \wedge \neg\ bval\ b\ s \longrightarrow Q\ s) \wedge$
($I\ s \wedge bval\ b\ s \longrightarrow wpc\ s) \wedge vcc\ s,\ I$))

lemma *vcpre-vc-pre*: *vcpre c Q* = (*vc c Q*, *pre c Q*)
by (*induct c arbitrary*: *Q*) (*simp-all add*: *Let-def*)

end

12 Hoare Logic for Total Correctness

theory *HoareT* **imports** *Hoare-Sound-Complete* **begin**

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

definition *hoare-tvalid* :: *assn* ⇒ *com* ⇒ *assn* ⇒ *bool*
($\models_t$ {(*1*-)}/ (-)/ {(*1*-)} *50*) **where**
$\models_t \{P\}c\{Q\} \equiv \forall s.\ P\ s \longrightarrow (\exists t.\ (c,s) \Rightarrow t \wedge Q\ t)$

Provability of Hoare triples in the proof system for total correctness is written $\vdash_t \{P\}c\{Q\}$ and defined inductively. The rules for $\vdash_t$ differ from those for $\vdash$ only in the one place where nontermination can arise: the *While*-rule.

inductive
hoaret :: *assn* ⇒ *com* ⇒ *assn* ⇒ *bool* ($\vdash_t$ ({(*1*-)}/ (-)/ {(*1*-)}) *50*)
where
Skip: $\vdash_t \{P\}\ SKIP\ \{P\}$ |
Assign: $\vdash_t \{\lambda s.\ P(s[a/x])\}\ x::=a\ \{P\}$ |
Semi: $[\![\vdash_t \{P_1\}\ c_1\ \{P_2\}; \vdash_t \{P_2\}\ c_2\ \{P_3\}]\!] \Longrightarrow \vdash_t \{P_1\}\ c_1;c_2\ \{P_3\}$ |
If: $[\![\vdash_t \{\lambda s.\ P\ s \wedge bval\ b\ s\}\ c_1\ \{Q\}; \vdash_t \{\lambda s.\ P\ s \wedge \neg\ bval\ b\ s\}\ c_2\ \{Q\}]\!]$
$\Longrightarrow \vdash_t \{P\}\ IF\ b\ THEN\ c_1\ ELSE\ c_2\ \{Q\}$ |
While:
$[\![\bigwedge n::nat.\ \vdash_t \{\lambda s.\ P\ s \wedge bval\ b\ s \wedge f\ s = n\}\ c\ \{\lambda s.\ P\ s \wedge f\ s < n\}]\!]$
$\Longrightarrow \vdash_t \{P\}\ WHILE\ b\ DO\ c\ \{\lambda s.\ P\ s \wedge \neg bval\ b\ s\}$ |
conseq: $[\![\forall s.\ P'\ s \longrightarrow P\ s; \vdash_t \{P\}c\{Q\}; \forall s.\ Q\ s \longrightarrow Q'\ s\ ]\!] \Longrightarrow$
$\vdash_t \{P'\}c\{Q'\}$

The *While*-rule is like the one for partial correctness but it requires additionally that with every execution of the loop body some measure function f :: *state* ⇒ *nat* decreases.

lemma *strengthen-pre*:
$[\![\forall s.\ P'\ s \longrightarrow P\ s;\ \vdash_t \{P\}\ c\ \{Q\}\ ]\!] \Longrightarrow \vdash_t \{P'\}\ c\ \{Q\}$
by (*metis conseq*)

lemma *weaken-post*:
$[\![\vdash_t \{P\}\ c\ \{Q\};\ \forall s.\ Q\ s \longrightarrow Q'\ s\ ]\!] \Longrightarrow\ \vdash_t \{P\}\ c\ \{Q'\}$
by (*metis conseq*)

lemma *Assign'*: $\forall s.\ P\ s \longrightarrow Q(s[a/x]) \Longrightarrow \vdash_t \{P\}\ x ::= a\ \{Q\}$
by (*simp add*: *strengthen-pre*[*OF* - *Assign*])

lemma *While'*:
assumes $\bigwedge n{::}nat.\ \vdash_t \{\lambda s.\ P\ s \wedge bval\ b\ s \wedge f\ s = n\}\ c\ \{\lambda s.\ P\ s \wedge f\ s < n\}$
and $\forall s.\ P\ s \wedge \neg\ bval\ b\ s \longrightarrow Q\ s$
shows $\vdash_t \{P\}\ WHILE\ b\ DO\ c\ \{Q\}$
by(*blast intro*: *assms*(*1*) *weaken-post*[*OF While assms*(*2*)])

Our standard example:

abbreviation $w\ n ==$
WHILE Less (*V* $''y''$) (*N n*)
DO ($''y''$ ::= *Plus* (*V* $''y''$) (*N 1*); $''x''$ ::= *Plus* (*V* $''x''$) (*V* $''y''$))

lemma $\vdash_t \{\lambda s.\ 0 \le n\}\ ''x'' ::= N\ 0;\ ''y'' ::= N\ 0;\ w\ n\ \{\lambda s.\ s\ ''x'' = \sum\{1..n\}\}$
apply(*rule Semi*)
prefer *2*
apply(*rule While'*
[**where** $P = \lambda s.\ s\ ''x'' = \sum\ \{1..s\ ''y''\} \wedge 0 \le s\ ''y'' \wedge s\ ''y'' \le n$
and $f = \lambda s.\ nat\ (n - s\ ''y'')$])
apply(*rule Semi*)
prefer *2*
apply(*rule Assign*)
apply(*rule Assign'*)
apply (*simp add*: *atLeastAtMostPlus1-int-conv algebra-simps*)
apply *clarsimp*
apply *fastforce*
apply(*rule Semi*)
prefer *2*
apply(*rule Assign*)
apply(*rule Assign'*)
apply *simp*
done

The soundness theorem:

theorem *hoaret-sound*: $\vdash_t \{P\}c\{Q\} \implies \models_t \{P\}c\{Q\}$
proof(*unfold hoare-tvalid-def*, *induct rule*: *hoaret.induct*)
case (*While P b f c*)
show *?case*
proof
fix *s*
show $P\ s \longrightarrow (\exists\, t.\ (WHILE\ b\ DO\ c,\ s) \Rightarrow t \wedge P\ t \wedge \neg\ bval\ b\ t)$
proof(*induction f s arbitrary*: *s rule*: *less-induct*)
case (*less n*)
thus *?case* **by** (*metis While*(*2*) *WhileFalse WhileTrue*)
qed
qed
next
case *If* **thus** *?case* **by** *auto blast*
qed *fastforce*+

The completeness proof proceeds along the same lines as the one for partial correctness. First we have to strengthen our notion of weakest precondition to take termination into account:

definition *wpt* :: *com* ⇒ *assn* ⇒ *assn* (wp_t) **where**
$wp_t\ c\ Q \equiv \lambda s.\ \exists\, t.\ (c,s) \Rightarrow t \wedge Q\ t$

lemma [*simp*]: $wp_t\ SKIP\ Q = Q$
by(*auto intro*!: *ext simp*: *wpt-def*)

lemma [*simp*]: $wp_t\ (x ::= e)\ Q = (\lambda s.\ Q(s(x := aval\ e\ s)))$
by(*auto intro*!: *ext simp*: *wpt-def*)

lemma [*simp*]: $wp_t\ (c_1;c_2)\ Q = wp_t\ c_1\ (wp_t\ c_2\ Q)$
unfolding *wpt-def*
apply(*rule ext*)
apply *auto*
done

lemma [*simp*]:
$wp_t\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ Q = (\lambda s.\ wp_t\ (if\ bval\ b\ s\ then\ c_1\ else\ c_2)\ Q\ s)$
apply(*unfold wpt-def*)
apply(*rule ext*)
apply *auto*
done

Now we define the number of iterations *WHILE b DO c* needs to terminate when started in state *s*. Because this is a truly partial function, we define it as an (inductive) relation first:

inductive *Its* :: *bexp* ⇒ *com* ⇒ *state* ⇒ *nat* ⇒ *bool* **where**
Its-0: ¬ *bval b s* ⟹ *Its b c s 0* |
Its-Suc: ⟦ *bval b s*; (*c*,*s*) ⇒ *s*′; *Its b c s*′ *n* ⟧ ⟹ *Its b c s* (*Suc n*)

The relation is in fact a function:

lemma *Its-fun*: *Its b c s n* ⟹ *Its b c s n*′ ⟹ *n*=*n*′
proof(*induction arbitrary*: *n*′ *rule*:*Its.induct*)

case *Its-0*
from *this*(*1*) *Its.cases*[*OF this*(*2*)] **show** *?case* **by** *metis*
next
case (*Its-Suc b s c s*′ *n n*′)
note *C* = *this*
from *this*(*5*) **show** *?case*
proof *cases*
case *Its-0* **with** *Its-Suc*(*1*) **show** *?thesis* **by** *blast*
next
case *Its-Suc* **with** *C* **show** *?thesis* **by**(*metis big-step-determ*)
qed
qed

For all terminating loops, *Its* yields a result:

lemma *WHILE-Its*: (*WHILE b DO c*,*s*) ⇒ *t* ⟹ ∃ *n*. *Its b c s n*
proof(*induction WHILE b DO c s t rule*: *big-step-induct*)
case *WhileFalse* **thus** *?case* **by** (*metis Its-0*)
next
case *WhileTrue* **thus** *?case* **by** (*metis Its-Suc*)
qed

Now the relation is turned into a function with the help of the description operator *THE*:

definition *its* :: *bexp* ⇒ *com* ⇒ *state* ⇒ *nat* **where**
its b c s = (*THE n*. *Its b c s n*)

The key property: every loop iteration increases *its* by 1.

lemma *its-Suc*: ⟦ *bval b s*; (*c*, *s*) ⇒ *s*′; (*WHILE b DO c*, *s*′) ⇒ *t*⟧
⟹ *its b c s* = *Suc*(*its b c s*′)
by (*metis its-def WHILE-Its Its.intros*(*2*) *Its-fun the-equality*)

lemma *wpt-is-pre*: ⊢$_t$ {*wp*$_t$ *c Q*} *c* {*Q*}
proof (*induction c arbitrary*: *Q*)
case *SKIP* **show** *?case* **by** *simp* (*blast intro*:*hoaret.Skip*)
next
case *Assign* **show** *?case* **by** *simp* (*blast intro*:*hoaret.Assign*)

next
case *Semi* **thus** *?case* **by** *simp* (*blast intro*:*hoaret.Semi*)
next
case *If* **thus** *?case* **by** *simp* (*blast intro*:*hoaret.If hoaret.conseq*)
next
case (*While b c*)
let *?w* = *WHILE b DO c*
{ fix *n*
have ∀ *s*. wp_t *?w Q s* ∧ *bval b s* ∧ *its b c s* = *n* ⟶
wp_t *c* (λ*s*′. wp_t *?w Q s*′ ∧ *its b c s*′ < *n*) *s*
unfolding *wpt-def* **by** (*metis WhileE its-Suc lessI*)
note *strengthen-pre*[*OF this While*]
} note *hoaret.While*[*OF this*]
moreover have ∀ *s*. wp_t *?w Q s* ∧ ¬ *bval b s* ⟶ *Q s* **by** (*auto simp add*:*wpt-def*)
ultimately show *?case* **by**(*rule weaken-post*)
qed

In the *While*-case, *its* provides the obvious termination argument.

The actual completeness theorem follows directly, in the same manner as for partial correctness:

theorem *hoaret-complete*: ⊨$_t$ {*P*}*c*{*Q*} ⟹ ⊢$_t$ {*P*}*c*{*Q*}
apply(*rule strengthen-pre*[*OF - wpt-is-pre*])
apply(*auto simp*: *hoare-tvalid-def hoare-valid-def wpt-def*)
done

end

13 Abstract Interpretation

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

13.1 Complete Lattice (indexed)

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

theory *ACom*
imports *Com*
begin

definition *show-state xs s* = [(*x*,*s x*). *x* ← *xs*]

13.2 Annotated Commands

datatype *'a acom* =
SKIP 'a (*SKIP* {-} *61*) |
Assign vname aexp 'a ((- ::= -/ {-}) [*1000*, *61*, *0*] *61*) |
Semi (*'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 *map-acom* :: $('a \Rightarrow 'b) \Rightarrow 'a\ acom \Rightarrow 'b\ acom$ **where**
map-acom f (*SKIP* {*P*}) = *SKIP* {*f P*} |
map-acom f (*x* ::= *e* {*P*}) = (*x* ::= *e* {*f P*}) |
map-acom f (*c1*;*c2*) = (*map-acom f c1*; *map-acom f c2*) |
map-acom f (*IF b THEN c1 ELSE c2* {*P*}) =
(*IF b THEN map-acom f c1 ELSE map-acom f c2* {*f P*}) |
map-acom f ({*Inv*} *WHILE b DO c* {*P*}) =
({*f Inv*} *WHILE b DO map-acom f c* {*f P*})

lemma *post-map-acom*[*simp*]: *post*(*map-acom f c*) = *f*(*post c*)
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'*} ⟷ (∃ *S*. *c* = *x*::=*e* {*S*} ∧ *S'* = *f S*)
by (*cases c*) *auto*

lemma *map-acom-Semi*:
map-acom f c = *c1'*;*c2'* ⟷
(∃ *c1 c2*. *c* = *c1*;*c2* ∧ *map-acom f c1* = *c1'* ∧ *map-acom f c2* = *c2'*)
by (*cases c*) *auto*

lemma *map-acom-If*:
map-acom f c = *IF b THEN c1' ELSE c2'* {*S'*} ⟷
(∃ *S c1 c2*. *c* = *IF b THEN c1 ELSE c2* {*S*} ∧ *map-acom f c1* = *c1'* ∧
map-acom f c2 = *c2'* ∧ *S'* = *f S*)
by (*cases c*) *auto*

lemma *map-acom-While*:
map-acom f w = {*I'*} *WHILE b DO c'* {*P'*} ⟷
(∃ *I P c*. *w* = {*I*} *WHILE b DO c* {*P*} ∧ *map-acom f c* = *c'* ∧ *I'* = *f I* ∧
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 ⟷ (EX P. c = SKIP {P})
by (*cases c*) *simp-all*

lemma *strip-eq-Assign*:
strip c = x::=e ⟷ (EX P. c = x::=e {P})
by (*cases c*) *simp-all*

lemma *strip-eq-Semi*:
strip c = c1;c2 ⟷ (EX 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 ⟷
(EX 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 ⟷
(EX I d1 P. c = {I} WHILE b DO d1 {P} & strip d1 = c1)
by (*cases c*) *simp-all*

end

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

13.3 Collecting Semantics of Commands

13.3.1 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') |

$(\{Inv\}\ WHILE\ b\ DO\ c\ \{P\}) \leq (\{Inv'\}\ WHILE\ b'\ DO\ c'\ \{P'\}) =$
$(b{=}b' \wedge c \leq c' \wedge Inv \leq Inv' \wedge P \leq P')\ |$
less-eq-acom - - = False

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

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

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

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

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

definition *less-acom* :: $'a\ acom \Rightarrow 'a\ acom \Rightarrow bool$ **where**
$less\text{-}acom\ x\ y = (x \leq y \wedge \neg\ y \leq x)$

instance
proof
case *goal1* **show** *?case* **by**(*simp add*: *less-acom-def*)
next
case *goal2* **thus** *?case* **by** (*induct x*) *auto*
next
case *goal3* **thus** *?case*
apply(*induct x y arbitrary*: *z rule*: *less-eq-acom.induct*)
apply (*auto intro*: *le-trans simp*: *SKIP-le Assign-le Semi-le If-le While-le*)
done
next
case *goal4* **thus** *?case*
apply(*induct x y rule*: *less-eq-acom.induct*)
apply (*auto intro*: *le-antisym*)
done
qed

end

fun sub_1 :: $'a$ *acom* $\Rightarrow$ $'a$ *acom* **where**
$sub_1(c1;c2) = c1$ |
$sub_1(IF\ b\ THEN\ c1\ ELSE\ c2\ \{S\}) = c1$ |
$sub_1(\{I\}\ WHILE\ b\ DO\ c\ \{P\}) = c$

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

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

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

interpretation *Complete-Lattice-ix* $\%c.\ \{c'.\ strip\ c' = c\}$ *lift Inter*
proof
  case *goal1*
  have $a{:}A \Longrightarrow lift\ Inter\ (strip\ a)\ A \leq a$
  proof(*induction a arbitrary*: *A*)
    case *Semi* **from** *Semi.prems* **show** *?case* **by**(*force intro*!: *Semi.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 *goal1* **show** *?case* **by** *auto*
next
  case *goal2*
  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 Semi from Semi.prems show ?case
      by (force intro!: Semi.IH simp:Semi-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 goal3
  have strip(lift Inter i A) = i
  proof(induction i arbitrary: A)
    case Semi from Semi.prems show ?case
      by (fastforce simp: strip-eq-Semi subset-iff intro!: Semi.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 ≤ d ⟹ post c ≤ post d
by(induction c d rule: less-eq-acom.induct) auto
```

13.3.2 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'. EX 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 set-mp*)+
next
case *5* **thus** *?case* **by**(*simp add*: *subset-iff*) (*metis le-post set-mp*)
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

theory *Collecting-list*
imports *ACom*
begin

13.4 Executable Collecting Semantics on lists

fun *step* :: *state list* ⇒ *state list acom* ⇒ *state list acom* **where**
step S (*SKIP* {*P*}) = (*SKIP* {*S*}) |
step S (*x* ::= *e* {*P*}) =
(*x* ::= *e* {[*s*(*x* := *aval e s*). *s* ← *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]}

Examples:

definition *c = WHILE Less (V ''x'') (N 3)*
DO ''x'' ::= Plus (V ''x'') (N 2)
definition *c0 = anno [] c*

definition *show-acom xs = map-acom (map (show-state xs))*

Collecting semantics:

value *show-acom ['x'] (((step [<>]) ^^ 6) c0)*

Small step semantics:

value *show-acom ['x'] (((step []) ^^ 5) (step [<>] c0))*

end

theory *Abs-Int0-fun*
imports ~~/*src/HOL/ex/Interpretation-with-Defs*
~~/*src/HOL/Library/While-Combinator*
Collecting
begin

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

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

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

end

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

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

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

instance
proof
  case *goal2* **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
case *goal1* **thus** *?case* **by** (*induct x*) *auto*
next
case *goal2* **thus** *?case*
apply(*induct x y arbitrary*: *z rule*: *le-acom.induct*)
apply (*auto intro*: *le-trans*)
done
qed

end

13.5.1 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
  case *goal1* **show** *?case* **by**(*cases x*, *simp-all*)
next
  case *goal2* **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
  case *goal1* **thus** *?case* **by**(*cases x*, *simp*, *cases y*, *simp-all*)
next
  case *goal2* **thus** *?case* **by**(*cases y*, *simp*, *cases x*, *simp-all*)

next
case *goal3* **thus** *?case* **by**(*cases z*, *simp*, *cases y*, *simp*, *cases x*, *simp-all*)
next
case *goal4* **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

13.5.2 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* ‹$f\ p \sqsubseteq p$› **have** $f\ x \sqsubseteq p$ **by**(*rule le-trans*)
}
thus $x \sqsubseteq p$ **using** *assms*(2−) *while-option-rule*[**where** $P = \%x.\ x \sqsubseteq p$]
unfolding *pfp-def* **by** *blast*
qed

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

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

lemma *strip-pfp*:
assumes $\bigwedge x.\ g(f\ x) = g\ x$ **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** $\bigwedge c.\ strip(f\ c) = strip\ c$ **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*: $\bigwedge x\ y.\ x \sqsubseteq y \Longrightarrow f\ x \sqsubseteq f\ y$
and $strip\ p = c0$ **and** $f\ p \sqsubseteq p$ **and** *lp*: $lpfp_c\ f\ c0 = Some\ c$ **shows** $c \sqsubseteq p$
using *pfp-least*[*OF* - - *bot-acom*[*OF* ‹$strip\ p = c0$›] *lp*[*simplified* $lpfp_c$-*def*]]
mono ‹$f\ p \sqsubseteq p$›
by *blast*

13.6 Abstract Interpretation

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

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

The interface for abstract values:

locale *Val-abs* =
fixes γ :: '*av*::*SL-top* ⇒ *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-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*]: $strip(step'\ S\ c) = strip\ c$
by(*induct c arbitrary*: *S*) (*simp-all add*: *Let-def*)

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

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

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

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

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

lemma *mono-gamma-f*: $f \sqsubseteq g \Longrightarrow \gamma_f\ f \subseteq \gamma_f\ g$
by(*auto simp*: *le-fun-def* 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 *Semi* **thus** *?case* **apply** (*auto simp*: *Semi-le map-acom-Semi*)
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* (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

13.6.1 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

theory *Abs-State*
imports *Abs-Int0-fun*
~~/*src*/*HOL*/*Library*/*Char-ord* ~~/*src*/*HOL*/*Library*/*List-lexord*

begin

13.7 Abstract State with Computable Ordering

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* (*Option.map show-st*)
definition *show-acom-opt* = *Option.map 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* = (*ALL 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
case *goal2* **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* ⊑ *F'* ⟹ *lookup F x* ⊑ *lookup F' x*
by(*auto simp add*: *lookup-def le-st-def*)

lemma *mono-update*: *a* ⊑ *a'* ⟹ *S* ⊑ *S'* ⟹ *update S x a* ⊑ *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* ⇒ *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

theory *Abs-Int0*
imports *Abs-State*
begin

13.8 Computable Abstract Interpretation

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* : γ_f *S* ⟹ *aval a s* : γ(*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* ⇒ *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′ 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′* ⊤)

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* : γ_f *S*; *i* : γ *a* ⟧ ⟹ *s*(*x* := *i*) : γ_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*:
⟦ $S \subseteq \gamma_o\ S'$; $c \le \gamma_c\ c'$ ⟧ ⟹ $step\ S\ c \le \gamma_c\ (step'\ S'\ c')$
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 *Semi* **thus** *?case* **apply** (*auto simp*: *Semi-le map-acom-Semi*)
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 γ_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

13.8.1 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

13.8.2 Ascending Chain Condition

hide-const (**open**) *acc*

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

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

by(*auto simp*: *inv-image-def*)

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

ACC for option type:

lemma *acc-option*: **assumes** *acc* {(*x*,*y*::′*a*::*preord*). $x \sqsubseteq y$}
shows *acc* {(*x*,*y*::′*a*::*preord option*). $x \sqsubseteq y$}
proof(*auto simp*: *wf-eq-minimal*)
fix *xo* :: ′*a option* **and** *Qo* **assume** *xo* : *Qo*
let *?Q* = {*x*. *Some x* ∈ *Qo*}
show ∃ *yo*∈*Qo*. ∀ *zo*. *yo* $\sqsubseteq$ *zo* ∧ ~ *zo* $\sqsubseteq$ *yo* ⟶ *zo* ∉ *Qo* (**is** ∃ *zo*∈*Qo*. *?P zo*)
proof *cases*
assume *?Q* = {}
hence *?P None* **by** *auto*
moreover have *None* ∈ *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 *setsum-strict-mono1*:
fixes *f* :: ′*a* ⇒ ′*b*::{*comm-monoid-add*, *ordered-cancel-ab-semigroup-add*}
assumes *finite A* **and** *ALL x*:*A*. *f x* ≤ *g x* **and** *EX a*:*A*. *f a* < *g a*
shows *setsum f A* < *setsum g A*
proof−
from *assms*(*3*) **obtain** *a* **where** *a*: *a*:*A f a* < *g a* **by** *blast*
have *setsum f A* = *setsum f* ((*A*−{*a*}) ∪ {*a*})
by(*simp add*:*insert-absorb*[*OF* ‹*a*:*A*›])
also have ... = *setsum f* (*A*−{*a*}) + *setsum f* {*a*}
using ‹*finite A*› **by**(*subst setsum-Un-disjoint*) *auto*
also have *setsum f* (*A*−{*a*}) ≤ *setsum g* (*A*−{*a*})
by(*rule setsum-mono*)(*simp add*: *assms*(*2*))

also have *setsum f* $\{a\}$ $<$ *setsum g* $\{a\}$ **using** *a* **by** *simp*
also have *setsum g* $(A - \{a\})$ $+$ *setsum g* $\{a\}$ $=$ *setsum g*$((A-\{a\}) \cup \{a\})$
using ‹*finite A*› **by**(*subst setsum-Un-disjoint*[*symmetric*]) *auto*
also have $\ldots$ $=$ *setsum g A* **by**(*simp add*:*insert-absorb*[*OF* ‹*a*:*A*›])
finally show *?thesis* **by** (*metis add-right-mono add-strict-left-mono*)
qed

lemma *measure-st*: **assumes** (*strict*$\{(x,y::'a::SL\text{-}top).\ x \sqsubseteq y\})\hat{}-1$ $<=$ *measure m*
and $\forall\, x\ y::'a::SL\text{-}top.\ x \sqsubseteq y \wedge y \sqsubseteq x \longrightarrow m\ x = m\ y$
shows (*strict*$\{(S,S'::'a::SL\text{-}top\ st).\ S \sqsubseteq S'\})\hat{}-1 \subseteq$
measure$(\%fd.\ \sum x|\ x \in set(dom\ fd) \wedge\ ^\sim\ \top \sqsubseteq fun\ fd\ x.\ m(fun\ fd\ x)+1)$
proof−
{ fix $S\ S' ::\ 'a\ st$ **assume** $S \sqsubseteq S'\ ^\sim\ S' \sqsubseteq S$
let *?X* $=$ *set*(*dom S*) **let** *?Y* $=$ *set*(*dom S'*)
let *?f* $=$ *fun S* **let** *?g* $=$ *fun S'*
let *?X'* $= \{x{:}?X.\ ^\sim\ \top \sqsubseteq ?f\ x\}$ **let** *?Y'* $= \{y{:}?Y.\ ^\sim\ \top \sqsubseteq ?g\ y\}$
from ‹$S \sqsubseteq S'$› **have** *ALL* $y{:}?Y' \cap ?X.\ ?f\ y \sqsubseteq ?g\ y$
by(*auto simp*: *le-st-def lookup-def*)
hence *1*: *ALL* $y{:}?Y' \cap ?X.\ m(?g\ y)+1 \leq m(?f\ y)+1$
using *assms*(*1*,*2*) **by**(*fastforce*)
from ‹$^\sim\ S' \sqsubseteq S$› **obtain** *u* **where** *u*: $u : ?X\ ^\sim$ *lookup S' u* $\sqsubseteq ?f\ u$
by(*auto simp*: *le-st-def*)
hence $u : ?X'$ **by** *simp* (*metis preord-class.le-trans top*)
have $?Y' - ?X = \{\}$ **using** ‹$S \sqsubseteq S'$› **by**(*fastforce simp*: *le-st-def lookup-def*)
have $?Y' \cap ?X <= ?X'$ **apply** *auto*
apply (*metis* ‹$S \sqsubseteq S'$› *le-st-def lookup-def preord-class.le-trans*)
done
have $(\sum y \in ?Y'.\ m(?g\ y)+1) = (\sum y \in (?Y' - ?X) \cup (?Y' \cap ?X).\ m(?g\ y)+1)$
by (*metis Un-Diff-Int*)
also have $\ldots = (\sum y \in ?Y' \cap ?X.\ m(?g\ y)+1)$
using ‹$?Y' - ?X = \{\}$› **by** (*metis Un-empty-left*)
also have $\ldots < (\sum x \in ?X'.\ m(?f\ x)+1)$
proof *cases*
assume $u \in ?Y'$
hence $m(?g\ u) < m(?f\ u)$ **using** *assms*(*1*) ‹$S \sqsubseteq S'$› *u*
by (*fastforce simp*: *le-st-def lookup-def*)
have $(\sum y \in ?Y' \cap ?X.\ m(?g\ y)+1) < (\sum y \in ?Y' \cap ?X.\ m(?f\ y)+1)$
using ‹*u*:*?X*› ‹*u*:*?Y'*› ‹$m(?g\ u) < m(?f\ u)$›
by(*fastforce intro*!: *setsum-strict-mono1*[*OF - 1*])
also have $\ldots \leq (\sum y \in ?X'.\ m(?f\ y)+1)$
by(*simp add*: *setsum-mono3*[*OF -* ‹$?Y' \cap ?X <= ?X'$›])

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*: *setsum-mono*)
also have $\ldots \leq (\sum y \in ?X' - \{u\}.\ m(?f\ y)+1)$
by(*simp add*: *setsum-mono3*[*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 setsum-Un-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*: $\mathit{size}\ xs1 = \mathit{size}\ ys1 \Longrightarrow \mathit{size}\ xs2 = \mathit{size}\ ys2 \Longrightarrow$
$(xs1@xs2,\ ys1@ys2) : \mathit{listrel}\ r \longleftrightarrow$
$(xs1,ys1) : \mathit{listrel}\ r \wedge (xs2,ys2) : \mathit{listrel}\ r$
by(*auto simp add*: *listrel-iff-zip*)

lemma *listrel-converse*: *listrel*($r\hat{}-1$) = (*listrel r*)$\hat{}-1$
proof−
{ fix *xs ys*
have (*xs*,*ys*) : *listrel*($r\hat{}-1$) ⟷ (*ys*,*xs*) : *listrel r*
apply(*induct xs arbitrary*: *ys*)
apply (*fastforce simp*: *listrel.Nil*)
apply (*fastforce simp*: *listrel-Cons-iff*)
done
} thus *?thesis* **by** *auto*
qed

lemma *acc-listrel*: **fixes** r :: ($'a$*$'a$)*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* ⟹ *EX 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


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

lemma size-annos-same: strip c1 = strip c2 ⟹ size(annos c1) = size(annos
c2)
apply(induct c2 arbitrary: c1)
apply (auto simp: strip-eq-SKIP strip-eq-Assign strip-eq-Semi strip-eq-If
strip-eq-While)
done

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

lemma set-annos-anno: set (annos (anno a c)) = {a}
by(induction c)(auto)

lemma le-iff-le-annos: c1 ⊑ c2 ⟷
 (annos c1, annos c2) : listrel{(x,y). x ⊑ y} ∧ 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* ⊑ *c'*}) ^ −*1* ⊆
(*strict*(*inv-image* (*listrel*{(*a*,*a'*::'*a*). *a* ⊑ *a'*} − {([],[])}) *annos*)) ^ −*1*
by(*auto simp*: *le-iff-le-annos*)

lemma *acc-acom*: *acc* {(*a*,*a'*::'*a*::*preord*). *a* ⊑ *a'*} ⟹
acc {(*c*,*c'*::'*a acom*). *c* ⊑ *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*: ⋀*x y*. *x* ⊑ *y* ⟹ *f x* ⊑ *f y* **and** *acc* {(*x*::'*a*,*y*). *x* ⊑ *y*}
and *x0* ⊑ *f x0* **shows** *EX x*. *pfp f x0* = *Some x*
proof(*simp add*: *pfp-def*, *rule wf-while-option-Some*[**where** *P* = %*x*. *x* ⊑ *f x*])
show *wf* {(*x*, *s*). (*s* ⊑ *f s* ∧ ¬ *f s* ⊑ *s*) ∧ *x* = *f s*}
by(*rule wf-subset*[*OF assms*(*2*)]) *auto*
next
show *x0* ⊑ *f x0* **by**(*rule assms*)
next
fix *x* **assume** *x* ⊑ *f x* **thus** *f x* ⊑ *f*(*f x*) **by**(*rule mono*)
qed

lemma *lpfpc-termination*:
fixes *f* :: ((*'a*::*SL-top*)*option acom* ⇒ *'a option acom*)
assumes *acc* {(*x*::'*a*,*y*). *x* ⊑ *y*} **and** ⋀*x y*. *x* ⊑ *y* ⟹ *f x* ⊑ *f y*
and ⋀*c*. *strip*(*f c*) = *strip c*
shows ∃ *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 $\forall\, x\ y{::}'a.\ x \sqsubseteq y \wedge y \sqsubseteq x \longrightarrow m\ x = m\ y$
shows $\exists\, c'.\ AI\ c = Some\ c'$
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

theory *Abs-Int0-parity*
imports *Abs-Int0*
begin

13.9 Parity Analysis

datatype *parity* = *Even* | *Odd* | *Either*

Instantiation of class *preord* with type *parity*:

instantiation *parity* :: *preord*
begin

First the definition of the interface function $\sqsubseteq$. Note that the header of the definition must refer to the ascii name *op* $\sqsubseteq$ of the constants as *le-parity* and the definition is named *le-parity-def*. Inside the definition the symbolic names can be used.

definition *le-parity* **where**
$x \sqsubseteq y = (y = Either \vee x{=}y)$

Now the instance proof, i.e. the proof that the definition fulfills the axioms (assumptions) of the class. The initial proof-step generates the necessary proof obligations.

instance
proof
  fix *x*::*parity* **show** $x \sqsubseteq x$ **by**(*auto simp*: *le-parity-def*)
next
  fix *x y z* :: *parity* **assume** $x \sqsubseteq y\ y \sqsubseteq z$ **thus** $x \sqsubseteq z$
    by(*auto simp*: *le-parity-def*)
qed

end

Instantiation of class *SL-top* with type *parity*:

instantiation *parity* :: *SL-top*
begin

definition *join-parity* **where**
$x \sqcup y = (\mathit{if}\ x \sqsubseteq y\ \mathit{then}\ y\ \mathit{else}\ \mathit{if}\ y \sqsubseteq x\ \mathit{then}\ x\ \mathit{else}\ \mathit{Either})$

definition *Top-parity* **where**
$\top = \mathit{Either}$

Now the instance proof. This time we take a lazy shortcut: we do not write out the proof obligations but use the *goali* primitive to refer to the assumptions of subgoal i and *case?* to refer to the conclusion of subgoal i. The class axioms are presented in the same order as in the class definition.

instance
proof
  case *goal1* **show** *?case* **by**(*auto simp*: *le-parity-def join-parity-def*)
next
  case *goal2* **show** *?case* **by**(*auto simp*: *le-parity-def join-parity-def*)
next
  case *goal3* **thus** *?case* **by**(*auto simp*: *le-parity-def join-parity-def*)
next
  case *goal4* **show** *?case* **by**(*auto simp*: *le-parity-def Top-parity-def*)
qed

end

Now we define the functions used for instantiating the abstract interpretation locales. Note that the Isabelle terminology is *interpretation*, not *instantiation* of locales, but we use instantiation to avoid confusion with abstract interpretation.

fun γ-*parity* :: *parity* $\Rightarrow$ *val set* **where**
γ-*parity Even* $= \{i.\ i\ \mathit{mod}\ 2 = 0\}$ |
γ-*parity Odd* $= \{i.\ i\ \mathit{mod}\ 2 = 1\}$ |
γ-*parity Either* $= \mathit{UNIV}$

fun *num-parity* :: *val* $\Rightarrow$ *parity* **where**
num-parity i $= (\mathit{if}\ i\ \mathit{mod}\ 2 = 0\ \mathit{then}\ \mathit{Even}\ \mathit{else}\ \mathit{Odd})$

fun *plus-parity* :: *parity* $\Rightarrow$ *parity* $\Rightarrow$ *parity* **where**
plus-parity Even Even $=$ *Even* |
plus-parity Odd Odd $=$ *Even* |
plus-parity Even Odd $=$ *Odd* |

plus-parity Odd Even = *Odd* |
plus-parity Either y = *Either* |
plus-parity x Either = *Either*

First we instantiate the abstract value interface and prove that the functions on type *parity* have all the necessary properties:

interpretation *Val-abs*
where γ = γ*-parity* **and** *num′* = *num-parity* **and** *plus′* = *plus-parity*
proof

of the locale axioms

fix *a b* :: *parity*
assume *a* $\sqsubseteq$ *b* **thus** γ*-parity a* $\subseteq$ γ*-parity b*
by(*auto simp*: *le-parity-def*)
next

The rest in the lazy, implicit way

case *goal2* **show** *?case* **by**(*auto simp*: *Top-parity-def*)
next
case *goal3* **show** *?case* **by** *auto*
next

Warning: this subproof refers to the names *a1* and *a2* from the statement of the axiom.

case *goal4* **thus** *?case*
proof(*cases a1 a2 rule*: *parity.exhaust*[*case-product parity.exhaust*])
qed (*auto simp add*:*mod-add-eq*)
qed

Instantiating the abstract interpretation locale requires no more proofs (they happened in the instatiation above) but delivers the instantiated abstract interpreter which we call AI:

interpretation *Abs-Int*
where γ = γ*-parity* **and** *num′* = *num-parity* **and** *plus′* = *plus-parity*
defines *aval-parity* **is** *aval′* **and** *step-parity* **is** *step′* **and** *AI-parity* **is** *AI*
..

13.9.1 Tests

definition *test1-parity* =
''x'' ::= *N 1*;
WHILE Less (*V ''x''*) (*N 100*) *DO ''x''* ::= *Plus* (*V ''x''*) (*N 2*)

value *show-acom-opt* (*AI-parity test1-parity*)

definition *test2-parity* =
  ″x″ ::= *N 1*;
  WHILE Less (*V* ″x″) (*N 100*) *DO* ″x″ ::= *Plus* (*V* ″x″) (*N 3*)

value *show-acom* ((*step-parity* ⊤ ^^*1*) (*anno None test2-parity*))
value *show-acom* ((*step-parity* ⊤ ^^*2*) (*anno None test2-parity*))
value *show-acom* ((*step-parity* ⊤ ^^*3*) (*anno None test2-parity*))
value *show-acom* ((*step-parity* ⊤ ^^*4*) (*anno None test2-parity*))
value *show-acom* ((*step-parity* ⊤ ^^*5*) (*anno None test2-parity*))
value *show-acom-opt* (*AI-parity test2-parity*)

13.9.2 Termination

interpretation *Abs-Int-mono*
where $\gamma = \gamma$*-parity* **and** *num′* = *num-parity* **and** *plus′* = *plus-parity*
proof
  case *goal1* **thus** *?case*
  proof(*cases a1 a2 b1 b2*
  rule: *parity.exhaust*[*case-product parity.exhaust*[*case-product parity.exhaust*[*case-product parity.exhaust*]]])
  qed (*auto simp add*:*le-parity-def*)
qed

definition *m-parity* :: *parity* ⇒ *nat* **where**
m-parity x = (*if x*=*Either then 0 else 1*)

lemma *measure-parity*:
  (*strict*{(*x*::*parity*,*y*). *x* ⊑ *y*}) ^ −*1* ⊆ *measure m-parity*
by(*auto simp add*: *m-parity-def le-parity-def*)

lemma *measure-parity-eq*:
  ∀ *x y*::*parity*. *x* ⊑ *y* ∧ *y* ⊑ *x* ⟶ *m-parity x* = *m-parity y*
by(*auto simp add*: *m-parity-def le-parity-def*)

lemma *AI-parity-Some*: ∃ *c′*. *AI-parity c* = *Some c′*
by(*rule AI-Some-measure*[*OF measure-parity measure-parity-eq*])

end

theory *Abs-Int-Tests*
imports *ACom*
begin

13.10 Test Programs

For constant propagation:

Straight line code:

definition *test1-const* =
''y'' ::= N 7;
''z'' ::= Plus (V ''y'') (N 2);
''y'' ::= Plus (V ''x'') (N 0)

Conditional:

definition *test2-const* =
IF Less (N 41) (V ''x'') THEN ''x'' ::= N 5 ELSE ''x'' ::= N 5

Conditional, test is relevant:

definition *test3-const* =
''x'' ::= N 42;
IF Less (N 41) (V ''x'') THEN ''x'' ::= N 5 ELSE ''x'' ::= N 6

While:

definition *test4-const* =
''x'' ::= N 0; WHILE Bc True DO ''x'' ::= N 0

While, test is relevant:

definition *test5-const* =
''x'' ::= N 0; WHILE Less (V ''x'') (N 1) DO ''x'' ::= N 1

Iteration is needed:

definition *test6-const* =
''x'' ::= N 0; ''y'' ::= N 0; ''z'' ::= N 2;
WHILE Less (V ''x'') (N 1) DO (''x'' ::= V ''y''; ''y'' ::= V ''z'')

For intervals:

definition *test1-ivl* =
''y'' ::= N 7;
IF Less (V ''x'') (V ''y'')
THEN ''y'' ::= Plus (V ''y'') (V ''x'')
ELSE ''x'' ::= Plus (V ''x'') (V ''y'')

definition *test2-ivl* =
WHILE Less (V ''x'') (N 100)
DO ''x'' ::= Plus (V ''x'') (N 1)

definition *test3-ivl* =
''x'' ::= N 7;

WHILE Less (V ''x'') (N 100)
DO ''x'' ::= Plus (V ''x'') (N 1)

definition *test4-ivl* =
''x'' ::= N 0; ''y'' ::= N 0;
WHILE Less (V ''x'') (N 11)
DO (''x'' ::= Plus (V ''x'') (N 1); ''y'' ::= Plus (V ''y'') (N 1))

definition *test5-ivl* =
''x'' ::= N 0; ''y'' ::= N 0;
WHILE Less (V ''x'') (N 1000)
DO (''y'' ::= V ''x''; ''x'' ::= Plus (V ''x'') (N 1))

definition *test6-ivl* =
''x'' ::= N 0;
WHILE Less (V ''x'') (N 1) DO ''x'' ::= Plus (V ''x'') (N −1)

end

theory *Abs-Int0-const*
imports *Abs-Int0 Abs-Int-Tests*
begin

13.11 Constant Propagation

datatype *const* = *Const val* | *Any*

fun *γ-const* **where**
γ-const (Const n) = {*n*} |
γ-const (Any) = *UNIV*

fun *plus-const* **where**
plus-const (Const m) (Const n) = *Const*(*m*+*n*) |
plus-const - - = *Any*

lemma *plus-const-cases*: *plus-const a1 a2* =
(*case* (*a1*,*a2*) *of* (*Const m*, *Const n*) ⇒ *Const*(*m*+*n*) | - ⇒ *Any*)
by(*auto split*: *prod.split const.split*)

instantiation *const* :: *SL-top*
begin

fun *le-const* **where**
- $\sqsubseteq$ *Any* = *True* |
Const n $\sqsubseteq$ *Const m* = (*n*=*m*) |
Any $\sqsubseteq$ *Const* - = *False*

fun *join-const* **where**
Const m $\sqcup$ *Const n* = (*if n*=*m then Const m else Any*) |
- $\sqcup$ - = *Any*

definition $\top$ = *Any*

instance
proof
  case *goal1* **thus** *?case* **by** (*cases x*) *simp-all*
next
  case *goal2* **thus** *?case* **by**(*cases z*, *cases y*, *cases x*, *simp-all*)
next
  case *goal3* **thus** *?case* **by**(*cases x*, *cases y*, *simp-all*)
next
  case *goal4* **thus** *?case* **by**(*cases y*, *cases x*, *simp-all*)
next
  case *goal5* **thus** *?case* **by**(*cases z*, *cases y*, *cases x*, *simp-all*)
next
  case *goal6* **thus** *?case* **by**(*simp add*: *Top-const-def*)
qed

end

interpretation *Val-abs*
where γ = γ-*const* **and** *num*′ = *Const* **and** *plus*′ = *plus-const*
proof
  case *goal1* **thus** *?case*
    by(*cases a*, *cases b*, *simp*, *simp*, *cases b*, *simp*, *simp*)
next
  case *goal2* **show** *?case* **by**(*simp add*: *Top-const-def*)
next
  case *goal3* **show** *?case* **by** *simp*
next
  case *goal4* **thus** *?case*
    by(*auto simp*: *plus-const-cases split*: *const.split*)
qed

interpretation *Abs-Int*

where $\gamma = \gamma$*-const* **and** *num*$' =$ *Const* **and** *plus*$' =$ *plus-const*
defines *AI-const* **is** *AI* **and** *step-const* **is** *step*$'$ **and** *aval*$'$*-const* **is** *aval*$'$
..

13.11.1 Tests

value *show-acom* (((*step-const* $\top$) ^^*0*) ($\bot_c$ *test1-const*))
value *show-acom* (((*step-const* $\top$) ^^*1*) ($\bot_c$ *test1-const*))
value *show-acom* (((*step-const* $\top$) ^^*2*) ($\bot_c$ *test1-const*))
value *show-acom* (((*step-const* $\top$) ^^*3*) ($\bot_c$ *test1-const*))
value *show-acom-opt* (*AI-const test1-const*)

value *show-acom-opt* (*AI-const test2-const*)
value *show-acom-opt* (*AI-const test3-const*)

value *show-acom* (((*step-const* $\top$) ^^*0*) ($\bot_c$ *test4-const*))
value *show-acom* (((*step-const* $\top$) ^^*1*) ($\bot_c$ *test4-const*))
value *show-acom* (((*step-const* $\top$) ^^*2*) ($\bot_c$ *test4-const*))
value *show-acom* (((*step-const* $\top$) ^^*3*) ($\bot_c$ *test4-const*))
value *show-acom-opt* (*AI-const test4-const*)

value *show-acom* (((*step-const* $\top$) ^^*0*) ($\bot_c$ *test5-const*))
value *show-acom* (((*step-const* $\top$) ^^*1*) ($\bot_c$ *test5-const*))
value *show-acom* (((*step-const* $\top$) ^^*2*) ($\bot_c$ *test5-const*))
value *show-acom* (((*step-const* $\top$) ^^*3*) ($\bot_c$ *test5-const*))
value *show-acom* (((*step-const* $\top$) ^^*4*) ($\bot_c$ *test5-const*))
value *show-acom* (((*step-const* $\top$) ^^*5*) ($\bot_c$ *test5-const*))
value *show-acom-opt* (*AI-const test5-const*)

value *show-acom* (((*step-const* $\top$) ^^*0*) ($\bot_c$ *test6-const*))
value *show-acom* (((*step-const* $\top$) ^^*1*) ($\bot_c$ *test6-const*))
value *show-acom* (((*step-const* $\top$) ^^*2*) ($\bot_c$ *test6-const*))
value *show-acom* (((*step-const* $\top$) ^^*3*) ($\bot_c$ *test6-const*))
value *show-acom* (((*step-const* $\top$) ^^*4*) ($\bot_c$ *test6-const*))
value *show-acom* (((*step-const* $\top$) ^^*5*) ($\bot_c$ *test6-const*))
value *show-acom* (((*step-const* $\top$) ^^*6*) ($\bot_c$ *test6-const*))
value *show-acom* (((*step-const* $\top$) ^^*7*) ($\bot_c$ *test6-const*))
value *show-acom* (((*step-const* $\top$) ^^*8*) ($\bot_c$ *test6-const*))
value *show-acom* (((*step-const* $\top$) ^^*9*) ($\bot_c$ *test6-const*))
value *show-acom* (((*step-const* $\top$) ^^*10*) ($\bot_c$ *test6-const*))
value *show-acom* (((*step-const* $\top$) ^^*11*) ($\bot_c$ *test6-const*))
value *show-acom-opt* (*AI-const test6-const*)

Monotonicity:

interpretation *Abs-Int-mono*
where $\gamma = \gamma$-*const* **and** *num*$' =$ *Const* **and** *plus*$' =$ *plus-const*
proof
case *goal1* **thus** *?case*
by(*auto simp*: *plus-const-cases split*: *const.split*)
qed

Termination:

definition *m-const x* = (*case x of Const - ⇒ 1 | Any ⇒ 0*)

lemma *measure-const*:
(*strict*{(*x*::*const*,*y*). *x* ⊑ *y*}) ^ −1 ⊆ *measure m-const*
by(*auto simp*: *m-const-def split*: *const.splits*)

lemma *measure-const-eq*:
∀ *x y*::*const*. *x* ⊑ *y* ∧ *y* ⊑ *x* ⟶ *m-const x* = *m-const y*
by(*auto simp*: *m-const-def split*: *const.splits*)

lemma *EX c'*. *AI-const c* = *Some c'*
by(*rule AI-Some-measure*[*OF measure-const measure-const-eq*])

end

theory *Abs-Int1*
imports *Abs-Int0 Vars*
begin

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

definition *le-prod p1 p2* = (*fst p1* ⊑ *fst p2* ∧ *snd p1* ⊑ *snd p2*)

instance
proof
case *goal1* **show** *?case* **by**(*simp add*: *le-prod-def*)
next
case *goal2* **thus** *?case* **unfolding** *le-prod-def* **by**(*metis le-trans*)
qed

end

instantiation *com* :: *vars*

begin

fun *vars-com* :: *com* ⇒ *vname set* **where**
vars com.SKIP = {} |
vars (*x*::=*e*) = {*x*} ∪ *vars e* |
vars (*c1*;*c2*) = *vars c1* ∪ *vars c2* |
vars (*IF b THEN c1 ELSE c2*) = *vars b* ∪ *vars c1* ∪ *vars c2* |
vars (*WHILE b DO c*) = *vars b* ∪ *vars c*

instance ..

end

lemma *finite-avars*: *finite*(*vars*(*a*::*aexp*))
by(*induction a*) *simp-all*

lemma *finite-bvars*: *finite*(*vars*(*b*::*bexp*))
by(*induction b*) (*simp-all add*: *finite-avars*)

lemma *finite-cvars*: *finite*(*vars*(*c*::*com*))
by(*induction c*) (*simp-all add*: *finite-avars finite-bvars*)

13.12 Backward Analysis of Expressions

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** γ = γ **for** γ :: *'av*::*L-top-bot* ⇒ *val set* +
assumes *inter-gamma-subset-gamma-meet*:
γ *a1* ∩ γ *a2* ⊆ γ(*a1* ⊓ *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 set-mp*)

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 set-rev-mp*)

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

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

13.12.1 Backward analysis

fun *afilter* :: *aexp* $\Rightarrow$ *'av* $\Rightarrow$ *'av st option* $\Rightarrow$ *'av st option* **where**
afilter (*N n*) *a S* = (*if test-num' n a then S else None*) |

```
afilter (V x) a S = (case S of None ⇒ 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* **by**(*fastforce simp*: *in-gamma-join-UpI*)
next
case (*Less e1 e2*) **thus** *?case*
by (*auto split*: *prod.split*)
(*metis afilter-sound filter-less' aval''-sound Less*)
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*)

13.12.2 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 *Semi* **thus** *?case* **apply** (*auto simp*: *Semi-le map-acom-Semi*)
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*}
$P \subseteq \gamma_o\ Pa\ cs1 \le \gamma_c\ ca1\ cs2 \le \gamma_c\ ca2$
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* $\le \gamma_c$ *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* $\le \gamma_c$ *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*}
$I \subseteq \gamma_o\ Ia\ P \subseteq \gamma_o\ Pa\ cs1 \le \gamma_c\ ca1$
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* $\le \gamma_c$ *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* $\le \gamma_c$ *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
from *this 2* **show** *lfp* (*step UNIV*) $c \leq \gamma_c\ c'$
by (*blast intro*: *mono-gamma-c order-trans*)
qed

13.12.3 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* $\top$ = {}
by(*simp add*: *domo-def Top-st-def Top-option-def*)

lemma *bot-acom-Com*[*simp*]: $\bot_c\ c \in$ *Com* X
by(*simp add*: *bot-acom-def Com-def domo-def set-annos-anno*)

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

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

lemma *domo-afilter*: *vars* $a \subseteq X \Longrightarrow$ *domo* $S \subseteq X \Longrightarrow$ *domo*(*afilter* a i S) $\subseteq X$
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* $b \subseteq X \Longrightarrow$ *domo* $S \subseteq X \Longrightarrow$ *domo*(*bfilter* b bv S) $\subseteq X$
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

13.12.4 Monotonicity

locale *Abs-Int1-mono* = *Abs-Int1* +
assumes *mono-plus′*: *a1* ⊑ *b1* ⟹ *a2* ⊑ *b2* ⟹ *plus′ a1 a2* ⊑ *plus′ b1 b2*
and *mono-filter-plus′*: *a1* ⊑ *b1* ⟹ *a2* ⊑ *b2* ⟹ *r* ⊑ *r′* ⟹
filter-plus′ r a1 a2 ⊑ *filter-plus′ r′ b1 b2*
and *mono-filter-less′*: *a1* ⊑ *b1* ⟹ *a2* ⊑ *b2* ⟹
filter-less′ bv a1 a2 ⊑ *filter-less′ bv b1 b2*
begin

lemma *mono-aval′*: *S* ⊑ *S′* ⟹ *aval′ e S* ⊑ *aval′ e S′*
by(*induction e*) (*auto simp*: *le-st-def lookup-def mono-plus′*)

lemma *mono-aval″*: *S* ⊑ *S′* ⟹ *aval″ e S* ⊑ *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(*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

theory *Abs-Int1-ivl*
imports *Abs-Int1 Abs-Int-Tests*
begin

13.13 Interval Analysis

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
case *goal1* **thus** *?case*
by(*cases x*, *simp add*: *le-ivl-def le-option-def split*: *option.split*)
next
case *goal2* **thus** *?case*
by(*cases x*, *cases y*, *cases z*, *auto simp*: *le-ivl-def le-option-def split*: *option.splits if-splits*)
next

case *goal3* **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 *goal4* **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 *goal5* **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 *goal6* **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
case *goal1* **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 *goal2* **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 *goal3* **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 *goal4* **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 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*)))

interpretation *Val-abs*
where *γ* = *γ-ivl* **and** *num′* = *num-ivl* **and** *plus′* = *plus-ivl*
proof
case *goal1* **thus** *?case*
by(*auto simp*: *γ-ivl-def le-ivl-def le-option-def split*: *ivl.split option.split if-splits*)
next
case *goal2* **show** *?case* **by**(*simp add*: *γ-ivl-def Top-ivl-def*)
next
case *goal3* **thus** *?case* **by**(*simp add*: *γ-ivl-def num-ivl-def*)
next
case *goal4* **thus** *?case*

by(*auto simp add*: γ*-ivl-def plus-ivl-def split*: *ivl.split option.splits*)
qed

interpretation *Val-abs1-gamma*
where $\gamma = \gamma$*-ivl* **and** *num′* = *num-ivl* **and** *plus′* = *plus-ivl*
defines *aval-ivl* **is** *aval′*
proof
case *goal1* **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 *goal2* **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

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
case *goal1* **thus** *?case*
by (*simp add*: γ*-ivl-def split*: *ivl.split option.split*)
next
case *goal2* **thus** *?case*
by(*auto simp add*: *filter-plus-ivl-def*)
(*metis gamma-minus-ivl add-diff-cancel add-commute*)+
next
case *goal3* **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

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* **is** *afilter*
and *bfilter-ivl* **is** *bfilter*
and *step-ivl* **is** *step′*
and *AI-ivl* **is** *AI*
and *aval-ivl′* **is** *aval″*
..

Monotonicity:

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
case *goal1* **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 *goal2* **thus** *?case*
by(*auto simp*: *filter-plus-ivl-def le-prod-def mono-meet mono-minus-ivl*)
next
case *goal3* **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

13.13.1 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* ⊤) ^^*50*) (⊥$_c$ *test6-ivl*))

end

theory *Abs-Int2*
imports *Abs-Int1-ivl*
begin

13.14 Widening and Narrowing

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*

13.14.1 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

13.14.2 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
case *goal1* **thus** *?case*
by(*simp add*: *widen-st-def le-st-def lookup-def widen1*)
next
case *goal2* **thus** *?case*
by(*simp add*: *widen-st-def le-st-def lookup-def widen2*)
next
case *goal3* **thus** *?case*
by(*auto simp*: *narrow-st-def le-st-def lookup-def narrow1*)
next
case *goal4* **thus** *?case*
by(*auto simp*: *narrow-st-def le-st-def lookup-def narrow2*)
qed

end

13.14.3 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
  case *goal1* **show** *?case*
    by(*induct x y rule*: *widen-option.induct*) (*simp-all add*: *widen1*)
next
  case *goal2* **show** *?case*
    by(*induct x y rule*: *widen-option.induct*) (*simp-all add*: *widen2*)
next
  case *goal3* **thus** *?case*
    by(*induct x y rule*: *narrow-option.induct*) (*simp-all add*: *narrow1*)
next
  case *goal4* **thus** *?case*
    by(*induct x y rule*: *narrow-option.induct*) (*simp-all add*: *narrow2*)
qed

end

13.14.4 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*}) =

$(\{f\ a1\ a3\}\ WHILE\ b\ DO\ map2\text{-}acom\ f\ c\ c'\ \{f\ a2\ a4\})$

abbreviation *widen-acom* :: $('a::WN)acom \Rightarrow 'a\ acom \Rightarrow 'a\ acom$ (**infix** ∇_c *65*)
where *widen-acom* == *map2-acom* (*op* ∇)

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

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*)

13.14.5 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* $\Longrightarrow$ *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'* $\Longrightarrow f\ c' \sqsubseteq c'$
by(*auto simp add*: *iter-widen-def dest*: *while-option-stop*)

lemma *strip-while*: **fixes** $f :: 'a\ acom \Rightarrow 'a\ acom$
assumes $\forall c.\ strip\ (f\ c) = strip\ c$ **and** *while-option* $P\ f\ c = Some\ c'$
shows $strip\ c' = strip\ c$
using *while-option-rule*[**where** $P = \lambda c'.\ strip\ c' = strip\ c$, *OF - assms*(*2*)]
by (*metis assms*(*1*))

lemma *strip-iter-widen*: **fixes** $f :: 'a{::}WN\ acom \Rightarrow 'a\ acom$
assumes $\forall c.\ strip\ (f\ c) = strip\ c$ **and** *iter-widen* $f\ c = Some\ c'$
shows $strip\ c' = strip\ c$
proof−
have $\forall c.\ strip(c \bigtriangledown_c f\ c) = strip\ c$ **by** (*metis assms*(*1*) *strip-map2-acom*)
from *strip-while*[*OF this*] *assms*(*2*) **show** *?thesis* **by**(*simp add*: *iter-widen-def*)
qed

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

lemma *pfp-wn-pfp*:
$[\![\ mono\ f;\ \ pfp\text{-}wn\ f\ c = Some\ c'\ ]\!] \Longrightarrow f\ c' \sqsubseteq c'$
unfolding *pfp-wn-def*
by (*auto dest*: *iter-widen-pfp iter-narrow-pfp split*: *option.splits*)

lemma *strip-pfp-wn*:
$[\![\ \forall c.\ strip(f\ c) = strip\ c;\ pfp\text{-}wn\ f\ c = Some\ c'\ ]\!] \Longrightarrow strip\ c' = c$
apply(*auto simp add*: *pfp-wn-def iter-narrow-def split*: *option.splits*)

by (*metis* (*no-types*) *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*} $\Rightarrow$ *val set*
begin

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

lemma *AI-wn-sound*: *AI-wn c* = *Some c'* $\Longrightarrow$ *CS c* $\le \gamma_c$ *c'*
proof(*simp add*: *CS-def AI-wn-def*)
assume *1*: *pfp-wn* (*step'* $\top$) *c* = *Some c'*
from *pfp-wn-pfp*[*OF mono-step'2 1*]
have *2*: *step'* $\top$ *c'* $\sqsubseteq$ *c'* **.**
have *3*: *strip* (γ_c (*step'* $\top$ *c'*)) = *c* **by**(*simp add*: *strip-pfp-wn*[*OF - 1*])
have *lfp* (*step UNIV*) *c* $\le \gamma_c$ (*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
from *this 2* **show** *lfp* (*step UNIV*) *c* $\le \gamma_c$ *c'*
by (*blast intro*: *mono-gamma-c order-trans*)
qed

end

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'* **is** *AI-wn*
..

13.14.6 Tests

definition *step-up-ivl n* = ((λ*c*. *c* ∇_c *step-ivl* $\top$ *c*) ^^*n*)
definition *step-down-ivl n* = ((λ*c*. *c* $\triangle_c$ *step-ivl* $\top$ *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*)

13.14.7 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*:

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

13.14.8 Termination: Abstract State

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

lemma *m-st-height*: **assumes** *finite X* **and** *set* (*dom S*) $\subseteq X$
shows *m-st m-ivl S* $\leq$ *2* $*$ *card X*
proof(*auto simp*: *m-st-def*)
have $(\sum x{\in}$*set*(*dom S*)*.* *m-ivl* (*fun S x*)) $\leq (\sum x{\in}$*set*(*dom S*)*.* *2*) (**is** *?L* $\leq$ -)
by(*rule setsum-mono*)(*simp add*:*m-ivl-height*)
also have ... $\leq (\sum x{\in}X.\ 2)$
by(*rule setsum-mono3*[*OF assms*]) *simp*
also have ... $=$ *2* $*$ *card X* **by**(*simp add*: *setsum-constant*)
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-ivl*(*?g y*)$) = (\sum y{\in}$(*?Y* $-$ *?X*) $\cup$ (*?Y* $\cap$ *?X*)*.* *m-ivl*(*?g y*))
by (*metis Un-Diff-Int*)
also have ... $= (\sum y{\in}$*?Y* $-$ *?X*. *m-ivl*(*?g y*)$) + (\sum y{\in}$*?Y* $\cap$ *?X*. *m-ivl*(*?g y*))
by(*subst setsum-Un-disjoint*) *auto*
also have $(\sum y{\in}$*?Y* $-$ *?X*. *m-ivl*(*?g y*)$) =$ *0* **using** *0* **by** *simp*
also have *0* $+ (\sum y{\in}$*?Y* $\cap$ *?X*. *m-ivl*(*?g y*)$) = (\sum y{\in}$*?Y* $\cap$ *?X*. *m-ivl*(*?g y*)) **by** *simp*
also have ... $\leq (\sum y{\in}$*?Y* $\cap$ *?X*. *m-ivl*(*?f y*))
by(*rule setsum-mono*)(*simp add*: *1*)
also have ... $\leq (\sum y{\in}$*?X*. *m-ivl*(*?f y*))
by(*simp add*: *setsum-mono3*[*of ?X ?Y Int ?X*, *OF - Int-lower2*])
finally show $(\sum y{\in}$*?Y*. *m-ivl*(*?g y*)$) \leq (\sum x{\in}$*?X*. *m-ivl*(*?f x*))
by (*metis add-less-cancel-left*)

qed

lemma *m-st-widen*:
assumes ¬ *S2* ⊑ *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* ∈ *?X* ¬ *lookup S2 x* ⊑ *?f x*
have (∑ *x*∈*?X*∩*?Y*. *m-ivl*(*?f x* ▽ *?g x*)) < (∑ *x*∈*?X*. *m-ivl*(*?f x*)) (**is** *?L* < *?R*)
proof *cases*
assume *x* : *?Y*
have *?L* < (∑ *x*∈*?X*∩*?Y*. *m-ivl*(*?f x*))
proof(*rule setsum-strict-mono1*, *simp*)
show ∀ *x*∈*?X*∩*?Y*. *m-ivl*(*?f x* ▽ *?g x*) ≤ *m-ivl* (*?f x*)
by (*metis m-ivl-anti-mono widen1*)
next
show ∃ *x*∈*?X*∩*?Y*. *m-ivl*(*?f x* ▽ *?g x*) < *m-ivl*(*?f x*)
using ‹*x*:*?X*› ‹*x*:*?Y*› ‹¬ *lookup S2 x* ⊑ *?f x*›
by (*metis IntI m-ivl-widen lookup-def*)
qed
also have ... ≤ *?R* **by**(*simp add*: *setsum-mono3*[*OF - Int-lower1*])
finally show *?thesis* **.**
next
assume *x* ~: *?Y*
have *?L* ≤ (∑ *x*∈*?X*∩*?Y*. *m-ivl*(*?f x*))
proof(*rule setsum-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 setsum-mono3*)(*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)* $\subseteq$ *X set(dom S2)* $\subseteq$ *X S1* $\sqsubseteq$ *S2*
shows *n-st n-ivl X S1* $\leq$ *n-st n-ivl X S2*
proof−
  have $(\sum x{\in}X.\ n\text{-}ivl(lookup\ S1\ x)) \leq (\sum x{\in}X.\ n\text{-}ivl(lookup\ S2\ x))$
    apply(*rule setsum-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)* $\subseteq$ *X set(dom S2)* $\subseteq$ *X*
and *S2* $\sqsubseteq$ *S1* $\neg$ *S1* $\sqsubseteq$ *S1* $\triangle$ *S2*
shows *n-st n-ivl X* (*S1* $\triangle$ *S2*) $<$ *n-st n-ivl X S1*
proof−
  have *1*: $\forall x{\in}X.$ *n-ivl* (*lookup* (*S1* $\triangle$ *S2*) *x*) $\leq$ *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*: $\exists x{\in}X.$ *n-ivl* (*lookup* (*S1* $\triangle$ *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 $(\sum x{\in}X.\ n\text{-}ivl(lookup\ (S1\ \triangle\ S2)\ x)) < (\sum x{\in}X.\ n\text{-}ivl(lookup\ S1\ x))$
    apply(*rule setsum-strict-mono1*[*OF* ‹*finite X*›]) **using** *1 2* **by** *blast*+
  thus *?thesis* **by**(*simp add*: *n-st-def*)
qed

13.14.9 Termination: Option

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

lemma *m-o-anti-mono*: *finite X* $\Longrightarrow$ *domo S2* $\subseteq$ *X* $\Longrightarrow$ *S1* $\sqsubseteq$ *S2* $\Longrightarrow$
  m-o (*m-st m-ivl*) (*2* $*$ *card X*) *S2* $\leq$ *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* $\subseteq$ *X*; $\neg$ *S2* $\sqsubseteq$ *S1* $]\!] \Longrightarrow$
  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

13.14.10 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*}$^{-1}$
⊆ *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 setsum-strict-mono1*)
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* $\Longrightarrow$ $\{(c,\ c \bigtriangleup_c c')\ |c\ c'.$
strip c = *strip c'* $\wedge$ *c* $\in$ *Com X* $\wedge$ *c'* $\in$ *Com X* $\wedge$ *c'* $\sqsubseteq$ *c* $\wedge$ $\neg$ *c* $\sqsubseteq$ *c* $\bigtriangleup_c$ *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 setsum-strict-mono1*)
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 strip-narrow-acom*)
apply(*rule-tac x*=*n* **in** *bexI*)
prefer *2* **apply** *simp*
apply(*erule n-o-narrow*)
apply (*simp*)+
done

13.14.11 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*: $\bigwedge c\ c'.\ P\ c \Longrightarrow P\ c' \Longrightarrow P(c\ \nabla_c\ c')$

and $wf(\{(c::'a\ acom, c\ \nabla_c\ c') | c\ c'.\ P\ c \wedge P\ c' \wedge \sim\ c' \sqsubseteq c\}\ \hat{}\ -1)$
and $P\ c0$ **and** $c0 \sqsubseteq f\ c0$ **shows** $EX\ c.\ iter\text{-}widen\ f\ c0 = Some\ c$
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*: $wf(\{(c, c\ \triangle_c\ f\ c) | c\ c'.\ P\ c \wedge \sim\ c \sqsubseteq c\ \triangle_c\ f\ c\}\ \hat{}\ -1)$
and $P\ c0$ **shows** $EX\ c.\ iter\text{-}narrow\ f\ c0 = Some\ c$
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*:
$EX\ c.\ iter\text{-}widen\ (step\text{-}ivl\ \top)\ (\bot_c\ c0) = Some\ c$
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*:
$c0 \in Com\ (vars(strip\ c0)) \Longrightarrow step\text{-}ivl\ \top\ c0 \sqsubseteq c0 \Longrightarrow$
$EX\ c.\ iter\text{-}narrow\ (step\text{-}ivl\ \top)\ c0 = Some\ c$
apply(*rule iter-narrow-termination*[**where**
$P = \%c.\ strip\ c = strip\ c0 \wedge c : Com(vars(strip\ c0)) \wedge step\text{-}ivl\ \top\ c \sqsubseteq c$])

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 ∀ *c*::*′a st option acom*. c : *Com*(X) ⟶ *vars*(*strip c*) ⊆ X ⟶ *f c* : *Com*(X)
and c : *Com*(X) **and** *vars*(*strip c*) ⊆ X **shows** *c′* : *Com*(X)
using *while-option-rule*[**where** P = λ*c′*. *c′* : *Com*(X) ∧ *vars*(*strip c′*) ⊆ X, *OF - assms*(*1*)]
by(*simp add*: *assms*(*2*−))

lemma *iter-widen-Com*: **fixes** *f* :: *′a*::*WN st option acom* ⇒ *′a st option acom*
assumes *iter-widen f c* = *Some c′*
and ∀ *c*. c : *Com*(X) ⟶ *vars*(*strip c*) ⊆ X ⟶ *f c* : *Com*(X)
and !!c. *strip*(*f c*) = *strip c*
and c : *Com*(X) **and** *vars* (*strip c*) ⊆ X **shows** *c′* : *Com*(X)
proof−
have ∀ *c*. c : *Com*(X) ⟶ *vars*(*strip c*) ⊆ X ⟶ 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′* ⊤) c = *Some c′* ⟹ *vars*(*strip c*) ⊆ X ⟹ 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*:
EX 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. Winskel is (almost) right: Towards a mechanized semantics textbook. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *Lect. Notes in Comp. Sci.*, pages 180–192. Springer-Verlag, 1996.