Concrete Semantics

TN & GK

February 12, 2013

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)) (λx. if x = ''x'' then 7 else 0)*

The same state more concisely:

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

A little syntax magic to write larger states compactly:

definition *null_state* (<>) **where**
null_state ≡ λx. 0
syntax
State :: updbinds => 'a (<>)
translations
_State ms => _Update <> ms

We can now write a series of updates to the function *λx. 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*>

In the <*a := b*> syntax, variables that are not mentioned are 0 by default:

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*)) (*Bc 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: $r\ x\ y \Longrightarrow star\ r\ y\ z \Longrightarrow star\ r\ x\ z$

hide_fact (**open**) *refl step* — names too generic

lemma *star_trans*:
  $star\ r\ x\ y \Longrightarrow star\ r\ y\ z \Longrightarrow star\ r\ x\ z$
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*:: $'a{*}'b \Rightarrow 'a{*}'b \Rightarrow bool$, *split_format*(*complete*)]

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

lemma *star_step1*[*simp*, *intro*]: $r\ x\ y \Longrightarrow star\ r\ x\ y$
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*)
    | *Seq 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
  $big\_step :: com \times state \Rightarrow state \Rightarrow bool$ (**infix** $\Rightarrow$ *55*)
where
Skip: $(SKIP,s) \Rightarrow s$ |
Assign: $(x ::= a,s) \Rightarrow s(x := aval\ a\ s)$ |
Seq: $[\![(c_1,s_1) \Rightarrow s_2;\ (c_2,s_2) \Rightarrow s_3]\!] \Longrightarrow$
  $(c_1;c_2,\ s_1) \Rightarrow s_3$ |

IfTrue: $[\![bval\ b\ s;\ (c_1,s) \Rightarrow t]\!] \Longrightarrow$
  $(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \Rightarrow t$ |
IfFalse: $[\![\neg bval\ b\ s;\ (c_2,s) \Rightarrow t]\!] \Longrightarrow$
  $(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \Rightarrow t$ |

WhileFalse: $\neg bval\ b\ s \Longrightarrow (WHILE\ b\ DO\ c,s) \Rightarrow s$ |
WhileTrue: $[\![bval\ b\ s_1;\ (c,s_1) \Rightarrow s_2;\ (WHILE\ b\ DO\ c,\ s_2) \Rightarrow s_3]\!] \Longrightarrow$
  $(WHILE\ b\ DO\ c,\ s_1) \Rightarrow s_3$

schematic_lemma *ex*: $(''x'' ::= N\ 5;\ ''y'' ::= V\ ''x'',\ s) \Rightarrow ?t$
apply(*rule Seq*)
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 $\{t.\ (SKIP,\ \lambda\_.\ 0) \Rightarrow t\}$

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

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

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

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

Proof automation:

declare *big_step.intros* [*intro*]

The standard induction rule

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

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*

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

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*) ⇒ *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*) ⇒ *t*
thm *AssignE*
inductive_cases *SeqE*[*elim*!]: (*c1*;*c2*,*s1*) ⇒ *s3*
thm *SeqE*
inductive_cases *IfE*[*elim*!]: (*IF b THEN c1 ELSE c2*,*s*) ⇒ *t*
thm *IfE*

inductive_cases *WhileE*[*elim*]: (*WHILE b DO c*,*s*) ⇒ *t*
thm *WhileE*

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

An automatic example:

lemma (*IF b THEN SKIP ELSE SKIP*, *s*) ⇒ *t* ⟹ *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 Seq*)
— 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** ‹$(?iw,\ s) \Rightarrow t$› **by** *blast*
hence $(?w,\ s) \Rightarrow t$ **using** ‹$\neg$*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 ‹$(?iw,\ s) \Rightarrow t$› **have** $(c;\ ?w,\ s) \Rightarrow t$ **by** *auto*
— and for this, only the Seq-rule is applicable:
then obtain s' **where**
$(c,\ s) \Rightarrow s'$ **and** $(?w,\ s') \Rightarrow t$ **by** *auto*
— with this information, we can build a derivation tree for the *WHILE*

with ‹*bval b s*›
have $(?w,\ s) \Rightarrow t$ **by** (*rule WhileTrue*)
}
ultimately
— both cases together again give us what we want:
have $(?w,\ s) \Rightarrow t$ **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*) $\sim$ (*IF b THEN c*; *WHILE b DO c ELSE SKIP*)
by *blast*

lemma *triv_if*:
(*IF b THEN c ELSE c*) $\sim$ *c*
by *blast*

lemma *commute_if*:
(*IF b1 THEN* (*IF b2 THEN c11 ELSE c12*) *ELSE c2*)
$\sim$
(*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: $(x ::= a,\ s) \rightarrow (SKIP,\ s(x := aval\ a\ s))$ |

Seq1: $(SKIP;c_2,s) \rightarrow (c_2,s)$ |
Seq2: $(c_1,s) \rightarrow (c_1',s') \Longrightarrow (c_1;c_2,s) \rightarrow (c_1';c_2,s')$ |

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*) $\to$ (c_2,*s*) |

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

abbreviation
small_steps :: *com* $*$ *state* $\Rightarrow$ *com* $*$ *state* $\Rightarrow$ *bool* (**infix** $\to*$ *55*)
where *x* $\to*$ *y* == *star small_step x y*

4.2 Executability

code_pred *small_step* **.**

values {(*c'*,*map t* [''*x*'',''*y*'',''*z*'']) |*c' t*.
(''*x*'' ::= *V* ''*z*''; ''*y*'' ::= *V* ''*x*'',
<''*x*'' := *3*, ''*y*'' := *7*, ''*z*'' := *5*>) $\to*$ (*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 $a \to b \Longrightarrow \ldots$ 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 $\to$ 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*) $\to$ *ct*
thm *SkipE*
inductive_cases *AssignE*[*elim*!]: (*x*::=*a*,*s*) $\to$ *ct*
thm *AssignE*
inductive_cases *SeqE*[*elim*]: (*c1*;*c2*,*s*) $\to$ *ct*
thm *SeqE*
inductive_cases *IfE*[*elim*!]: (*IF b THEN c1 ELSE c2*,*s*) $\to$ *ct*
inductive_cases *WhileE*[*elim*]: (*WHILE b DO c*, *s*) $\to$ *ct*

A simple property:

lemma *deterministic*:
$cs \to cs' \Longrightarrow cs \to cs'' \Longrightarrow cs'' = cs'$
apply(*induction arbitrary*: *cs''* *rule*: *small_step.induct*)

```
apply blast+
done
```

4.4 Equivalence with big-step semantics

```
lemma star_seq2: (c1,s) →* (c1',s') ⟹ (c1;c2,s) →* (c1';c2,s')
proof(induction rule: star_induct)
  case refl thus ?case by simp
next
  case step
  thus ?case by (metis Seq2 star.step)
qed

lemma seq_comp:
  ⟦ (c1,s1) →* (SKIP,s2); (c2,s2) →* (SKIP,s3) ⟧
   ⟹ (c1;c2, s1) →* (SKIP,s3)
by(blast intro: star.step star_seq2 star_trans)
```

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

```
lemma big_to_small:
  cs ⇒ t ⟹ cs →* (SKIP,t)
proof (induction rule: big_step.induct)
  fix s show (SKIP,s) →* (SKIP,s) by simp
next
  fix x a s show (x ::= a,s) →* (SKIP, s(x := aval a s)) by auto
next
  fix c1 c2 s1 s2 s3
  assume (c1,s1) →* (SKIP,s2) and (c2,s2) →* (SKIP,s3)
  thus (c1;c2, s1) →* (SKIP,s3) by (rule seq_comp)
next
  fix s::state and b c0 c1 t
  assume bval b s
  hence (IF b THEN c0 ELSE c1,s) → (c0,s) by simp
  moreover assume (c0,s) →* (SKIP,t)
  ultimately
  show (IF b THEN c0 ELSE c1,s) →* (SKIP,t) by (metis star.simps)
next
  fix s::state and b c0 c1 t
  assume ¬bval b s
  hence (IF b THEN c0 ELSE c1,s) → (c1,s) by simp
  moreover assume (c1,s) →* (SKIP,t)
  ultimately
  show (IF b THEN c0 ELSE c1,s) →* (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*) →∗ (*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′*) →∗ (*SKIP*,*t*)
  assume *c*: (*c*,*s*) →∗ (*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*) →∗ (*SKIP*,*t*) **by**(*rule seq_comp*[*OF c w*])
  ultimately show (*WHILE b DO c*,*s*) →∗ (*SKIP*,*t*) **by** (*metis star.simps*)
qed

Each case of the induction can be proved automatically:

lemma *cs* ⇒ *t* ⟹ *cs* →∗ (*SKIP*,*t*)
proof (*induction rule*: *big_step.induct*)
  case *Skip* **show** *?case* **by** *blast*
next
  case *Assign* **show** *?case* **by** *blast*
next
  case *Seq* **thus** *?case* **by** (*blast intro*: *seq_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 seq_comp small_step.IfTrue star.step*[*of small_step*])
qed

lemma *small1_big_continue*:
  cs → *cs′* ⟹ *cs′* ⇒ *t* ⟹ *cs* ⇒ *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 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*)

fun *iexec* :: *instr* ⇒ *config* ⇒ *config* **where**
iexec instr (*i*,*s*,*stk*) = (*case instr of*
  LOADI n ⇒ (*i*+*1*,*s*, *n*#*stk*) |
  LOAD x ⇒ (*i*+*1*,*s*, *s x* # *stk*) |
  ADD ⇒ (*i*+*1*,*s*, (*hd2 stk* + *hd stk*) # *tl2 stk*) |

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

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

declare *exec1_def* [*simp*]

lemma *exec1I* [*intro*, *code_pred_intro*]:
$c' = iexec\ (P!!i)\ (i,s,stk) \Longrightarrow 0 \le i \Longrightarrow i < isize\ P$
$\Longrightarrow P \vdash (i,s,stk) \to c'$
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** *fastforce*

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*+

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 *iexec_shift* [*simp*]:
$((n+i',s',stk') = iexec\ x\ (n+i,s,stk)) = ((i',s',stk') = iexec\ x\ (i,s,stk))$
by(*auto split*:*instr.split*)

lemma *exec1_appendR*: $P \vdash c \rightarrow c' \Longrightarrow P@P' \vdash c \rightarrow c'$
by *auto*

lemma *exec_appendR*: $P \vdash c \rightarrow* c' \Longrightarrow P@P' \vdash c \rightarrow* c'$
by (*induction rule*: *exec.induct*) (*fastforce intro*: *exec1_appendR*)+

lemma *exec1_appendL*:
$P \vdash (i,s,stk) \rightarrow (i',s',stk') \Longrightarrow$
$P' \mathbin{@} P \vdash (isize(P')+i,s,stk) \rightarrow (isize(P')+i',s',stk')$
by (*auto split*: *instr.split*)

lemma *exec_appendL*:
$P \vdash (i,s,stk) \rightarrow* (i',s',stk') \Longrightarrow$
$P' \mathbin{@} P \vdash (isize(P')+i,s,stk) \rightarrow* (isize(P')+i',s',stk')$
by (*induction rule*: *exec_induct*) (*blast intro*!: *exec1_appendL*)+

Now we specialise the above lemmas to enable automatic proofs of $P \vdash c \rightarrow* c'$ 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*]:
$P \vdash (0,s,stk) \rightarrow* (j,t,stk') \Longrightarrow$
$instr\#P \vdash (1,s,stk) \rightarrow* (1+j,t,stk')$
by (*drule exec_appendL*[**where** P'=[*instr*]]) *simp*

lemma *exec_appendL_if*[*intro*]:
$isize\ P' <= i$
$\Longrightarrow P \vdash (i - isize\ P',s,stk) \rightarrow* (i',s',stk')$
$\Longrightarrow P' \mathbin{@} P \vdash (i,s,stk) \rightarrow* (isize\ P' + i',s',stk')$
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*]:
$P \vdash (0,s,stk) \rightarrow* (i',s',stk') \Longrightarrow$
$isize\ P \leq i' \Longrightarrow$
$P' \vdash (i' - isize\ P,s',stk') \rightarrow* (i'',s'',stk'') \Longrightarrow$
$j'' = isize\ P + i''$

$\Longrightarrow$
P @ $P' \vdash (0,s,stk) \to* (j'',s'',stk'')$
by(*metis exec_trans*[*OF exec_appendR exec_appendL_if*])

declare *Let_def*[*simp*]

5.4 Compilation

fun *acomp* :: *aexp* ⇒ *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 ⊢ (*0*,*s*,*stk*) →∗ (*isize*(*acomp a*),*s*,*aval a s*#*stk*)
by (*induction a arbitrary*: *stk*) *fastforce+*

fun *bcomp* :: *bexp* ⇒ *bool* ⇒ *int* ⇒ *instr list* **where**
bcomp (*Bc v*) *c n* = (*if v*=*c then* [*JMP n*] *else* []) |
bcomp (*Not b*) *c n* = *bcomp b* (¬*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*]:
$0 \le n \Longrightarrow$
bcomp b c n ⊢
(*0*,*s*,*stk*) →∗ (*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*=~*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 (Seq c1 s1 s2 c2 s3)
  let ?cc1 = ccomp c1  let ?cc2 = ccomp c2
  have ?cc1 @ ?cc2 ⊢ (0,s1,stk) →* (isize ?cc1,s2,stk)
    using Seq.IH(1) by fastforce
  moreover
  have ?cc1 @ ?cc2 ⊢ (isize ?cc1,s2,stk) →* (isize(?cc1 @ ?cc2),s3,stk)
    using Seq.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*,*s1*,*stk*)
using ⟨*bval b s1*⟩ **by** *fastforce*
moreover
have *?cw* ⊢ (*isize ?cb*,*s1*,*stk*) →∗ (*isize ?cb* + *isize ?cc*,*s2*,*stk*)
using *WhileTrue.IH*(*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

theory *Comp_Rev*
imports *Compiler*
begin

6 Compiler Correctness, Reverse Direction

6.1 Definitions

Execution in n steps for simpler induction

primrec
exec_n :: *instr list* ⇒ *config* ⇒ *nat* ⇒ *config* ⇒ *bool*
(_/ ⊢ (_ →^_/ _) [*65*,*0*,*1000*,*55*] *55*)
where
$P \vdash c \rightarrow \hat{}0\ c' = (c'{=}c)$ |
$P \vdash c \rightarrow \hat{}(Suc\ n)\ c'' = (\exists\, c'.\ (P \vdash c \rightarrow c') \wedge P \vdash c' \rightarrow \hat{}n\ c'')$

The possible successor pc's of an instruction at position n

definition
isuccs i n ≡ *case i of*
JMP j ⇒ {*n* + *1* + *j*}
| *JMPLESS j* ⇒ {*n* + *1* + *j*, *n* + *1*}
| *JMPGE j* ⇒ {*n* + *1* + *j*, *n* + *1*}
| _ ⇒ {*n* +*1*}

The possible successors pc's of an instruction list

definition
$succs\ P\ n = \{s.\ \exists\, i.\ 0 \le i \wedge i < isize\ P \wedge s \in isuccs\ (P!!i)\ (n{+}i)\}$

Possible exit pc's of a program

definition
$exits\ P = succs\ P\ 0 - \{0..< isize\ P\}$

6.2 Basic properties of *exec_n*

lemma *exec_n_exec*:
$P \vdash c \rightarrow\hat{}n\ c' \Longrightarrow P \vdash c \rightarrow* c'$
by (*induct n arbitrary*: *c*) (*auto simp del*: *exec1_def*)

lemma *exec_0* [*intro*!]: $P \vdash c \rightarrow\hat{}0\ c$ **by** *simp*

lemma *exec_Suc*:
$[\![P \vdash c \rightarrow c'; P \vdash c' \rightarrow\hat{}n\ c'']\!] \Longrightarrow P \vdash c \rightarrow\hat{}(Suc\ n)\ c''$
by (*fastforce simp del*: *split_paired_Ex*)

lemma *exec_exec_n*:
$P \vdash c \rightarrow* c' \Longrightarrow \exists n.\ P \vdash c \rightarrow\hat{}n\ c'$
by (*induct rule*: *exec.induct*) (*auto simp del*: *exec1_def intro*: *exec_Suc*)

lemma *exec_eq_exec_n*:
$(P \vdash c \rightarrow* c') = (\exists n.\ P \vdash c \rightarrow\hat{}n\ c')$
by (*blast intro*: *exec_exec_n exec_n_exec*)

lemma *exec_n_Nil* [*simp*]:
$[] \vdash c \rightarrow\hat{}k\ c' = (c' = c \wedge k = 0)$
by (*induct k*) *auto*

lemma *exec1_exec_n* [*intro*!]:
$P \vdash c \rightarrow c' \Longrightarrow P \vdash c \rightarrow\hat{}1\ c'$
by (*cases c'*) *simp*

6.3 Concrete symbolic execution steps

lemma *exec_n_step*:
$n \neq n' \Longrightarrow$
$P \vdash (n,stk,s) \rightarrow\hat{}k\ (n',stk',s') =$
$(\exists c.\ P \vdash (n,stk,s) \rightarrow c \wedge P \vdash c \rightarrow\hat{}(k - 1)\ (n',stk',s') \wedge 0 < k)$
by (*cases k*) *auto*

lemma *exec1_end*:
$isize\ P <= fst\ c \Longrightarrow \neg\ P \vdash c \rightarrow c'$
by *auto*

lemma *exec_n_end*:

```
  isize P <= n ⟹
  P ⊢ (n,s,stk) → ^k (n',s',stk') = (n' = n ∧ stk'=stk ∧ s'=s ∧ k =0)
  by (cases k) (auto simp: exec1_end)

lemmas exec_n_simps = exec_n_step exec_n_end
```

6.4 Basic properties of *succs*

```
lemma succs_simps [simp]:
  succs [ADD] n = {n + 1}
  succs [LOADI v] n = {n + 1}
  succs [LOAD x] n = {n + 1}
  succs [STORE x] n = {n + 1}
  succs [JMP i] n = {n + 1 + i}
  succs [JMPGE i] n = {n + 1 + i, n + 1}
  succs [JMPLESS i] n = {n + 1 + i, n + 1}
  by (auto simp: succs_def isuccs_def)

lemma succs_empty [iff]: succs [] n = {}
  by (simp add: succs_def)

lemma succs_Cons:
  succs (x#xs) n = isuccs x n ∪ succs xs (1+n) (is _ = ?x ∪ ?xs)
proof
  let ?isuccs = λp P n i. 0 ≤ i ∧ i < isize P ∧ p ∈ isuccs (P!!i) (n+i)
  { fix p assume p ∈ succs (x#xs) n
    then obtain i where isuccs: ?isuccs p (x#xs) n i
      unfolding succs_def by auto
    have p ∈ ?x ∪ ?xs
    proof cases
      assume i = 0 with isuccs show ?thesis by simp
    next
      assume i ≠ 0
      with isuccs
      have ?isuccs p xs (1+n) (i − 1) by auto
      hence p ∈ ?xs unfolding succs_def by blast
      thus ?thesis ..
    qed
  }
  thus succs (x#xs) n ⊆ ?x ∪ ?xs ..

  { fix p assume p ∈ ?x ∨ p ∈ ?xs
    hence p ∈ succs (x#xs) n
    proof
```

```
    assume p ∈ ?x thus ?thesis by (fastforce simp: succs_def)
  next
    assume p ∈ ?xs
    then obtain i where ?isuccs p xs (1+n) i
      unfolding succs_def by auto
    hence ?isuccs p (x#xs) n (1+i)
      by (simp add: algebra_simps)
    thus ?thesis unfolding succs_def by blast
  qed
 }
 thus ?x ∪ ?xs ⊆ succs (x#xs) n by blast
qed

lemma succs_iexec1:
 assumes c' = iexec (P!!i) (i,s,stk) 0 ≤ i i < isize P
 shows fst c' ∈ succs P 0
 using assms by (auto simp: succs_def isuccs_def split: instr.split)

lemma succs_shift:
 (p − n ∈ succs P 0) = (p ∈ succs P n)
 by (fastforce simp: succs_def isuccs_def split: instr.split)

lemma inj_op_plus [simp]:
 inj (op + (i::int))
 by (metis add_minus_cancel inj_on_inverseI)

lemma succs_set_shift [simp]:
 op + i ` succs xs 0 = succs xs i
 by (force simp: succs_shift [where n=i, symmetric] intro: set_eqI)

lemma succs_append [simp]:
 succs (xs @ ys) n = succs xs n ∪ succs ys (n + isize xs)
 by (induct xs arbitrary: n) (auto simp: succs_Cons algebra_simps)


lemma exits_append [simp]:
 exits (xs @ ys) = exits xs ∪ (op + (isize xs)) ` exits ys −
                  {0..<isize xs + isize ys}
 by (auto simp: exits_def image_set_diff)

lemma exits_single:
 exits [x] = isuccs x 0 − {0}
 by (auto simp: exits_def succs_def)
```

lemma *exits_Cons*:
  exits (*x* # *xs*) = (*isuccs x 0* − {*0*}) ∪ (*op* + *1*) ‘ *exits xs* −
                    {*0*..<*1* + *isize xs*}
  using *exits_append* [*of* [*x*] *xs*]
  by (*simp add*: *exits_single*)

lemma *exits_empty* [*iff*]: *exits* [] = {} **by** (*simp add*: *exits_def*)

lemma *exits_simps* [*simp*]:
  exits [*ADD*] = {*1*}
  exits [*LOADI v*] = {*1*}
  exits [*LOAD x*] = {*1*}
  exits [*STORE x*] = {*1*}
  i ≠ −*1* ⟹ *exits* [*JMP i*] = {*1* + *i*}
  i ≠ −*1* ⟹ *exits* [*JMPGE i*] = {*1* + *i*, *1*}
  i ≠ −*1* ⟹ *exits* [*JMPLESS i*] = {*1* + *i*, *1*}
  by (*auto simp*: *exits_def*)

lemma *acomp_succs* [*simp*]:
  succs (*acomp a*) *n* = {*n* + *1* .. *n* + *isize* (*acomp a*)}
  by (*induct a arbitrary*: *n*) *auto*

lemma *acomp_size*:
  1 ≤ *isize* (*acomp a*)
  by (*induct a*) *auto*

lemma *acomp_exits* [*simp*]:
  exits (*acomp a*) = {*isize* (*acomp a*)}
  by (*auto simp*: *exits_def acomp_size*)

lemma *bcomp_succs*:
  0 ≤ *i* ⟹
  succs (*bcomp b c i*) *n* ⊆ {*n* .. *n* + *isize* (*bcomp b c i*)}
                          ∪ {*n* + *i* + *isize* (*bcomp b c i*)}
proof (*induction b arbitrary*: *c i n*)
  case (*And b1 b2*)
  from *And.prems*
  show *?case*
    by (*cases c*)
       (*auto dest*: *And.IH*(*1*) [*THEN subsetD*, *rotated*]
                And.IH(*2*) [*THEN subsetD*, *rotated*])
qed *auto*

lemmas *bcomp_succsD* [*dest!*] = *bcomp_succs* [*THEN subsetD*, *rotated*]

lemma *bcomp_exits*:
  $0 \leq i \Longrightarrow$
  exits (*bcomp b c i*) $\subseteq$ {*isize* (*bcomp b c i*), *i* + *isize* (*bcomp b c i*)}
  by (*auto simp*: *exits_def*)

lemma *bcomp_exitsD* [*dest*!]:
  $p \in$ *exits* (*bcomp b c i*) $\Longrightarrow 0 \leq i \Longrightarrow$
  p = *isize* (*bcomp b c i*) $\vee$ *p* = *i* + *isize* (*bcomp b c i*)
  using *bcomp_exits* **by** *auto*

lemma *ccomp_succs*:
  succs (*ccomp c*) *n* $\subseteq$ {*n..n* + *isize* (*ccomp c*)}
proof (*induction c arbitrary*: *n*)
  case *SKIP* **thus** *?case* **by** *simp*
next
  case *Assign* **thus** *?case* **by** *simp*
next
  case (*Seq c1 c2*)
  from *Seq.prems*
  show *?case*
    by (*fastforce dest*: *Seq.IH* [*THEN subsetD*])
next
  case (*If b c1 c2*)
  from *If.prems*
  show *?case*
    by (*auto dest*!: *If.IH* [*THEN subsetD*] *simp*: *isuccs_def succs_Cons*)
next
  case (*While b c*)
  from *While.prems*
  show *?case* **by** (*auto dest*!: *While.IH* [*THEN subsetD*])
qed

lemma *ccomp_exits*:
  exits (*ccomp c*) $\subseteq$ {*isize* (*ccomp c*)}
  using *ccomp_succs* [*of c 0*] **by** (*auto simp*: *exits_def*)

lemma *ccomp_exitsD* [*dest*!]:
  $p \in$ *exits* (*ccomp c*) $\Longrightarrow$ *p* = *isize* (*ccomp c*)
  using *ccomp_exits* **by** *auto*

6.5 Splitting up machine executions

lemma *exec1_split*:

```
  P @ c @ P' ⊢ (isize P + i, s) → (j,s') ⟹ 0 ≤ i ⟹ i < isize c ⟹
  c ⊢ (i,s) → (j − isize P, s')
  by (auto split: instr.splits)

lemma exec_n_split:
  assumes P @ c @ P' ⊢ (isize P + i, s) →^n (j, s')
          0 ≤ i i < isize c
          j ∉ {isize P ..< isize P + isize c}
  shows ∃ s'' i' k m.
                  c ⊢ (i, s) →^k (i', s'') ∧
                  i' ∈ exits c ∧
                  P @ c @ P' ⊢ (isize P + i', s'') →^m (j, s') ∧
                  n = k + m
using assms proof (induction n arbitrary: i j s)
  case 0
  thus ?case by simp
next
  case (Suc n)
  have i: 0 ≤ i i < isize c by fact+
  from Suc.prems
  have j: ¬ (isize P ≤ j ∧ j < isize P + isize c) by simp
  from Suc.prems
  obtain i0 s0 where
    step: P @ c @ P' ⊢ (isize P + i, s) → (i0,s0) and
    rest: P @ c @ P' ⊢ (i0,s0) →^n (j, s')
    by clarsimp

  from step i
  have c: c ⊢ (i,s) → (i0 − isize P, s0) by (rule exec1_split)

  have i0 = isize P + (i0 − isize P) by simp
  then obtain j0 where j0: i0 = isize P + j0 ..

  note split_paired_Ex [simp del]

  { assume j0 ∈ {0 ..< isize c}
    with j0 j rest c
    have ?case
      by (fastforce dest!: Suc.IH intro!: exec_Suc)
  } moreover {
    assume j0 ∉ {0 ..< isize c}
    moreover
    from c j0 have j0 ∈ succs c 0
      by (auto dest: succs_iexec1 simp del: iexec.simps)
```

```
    ultimately
    have j0 ∈ exits c by (simp add: exits_def)
    with c j0 rest
    have ?case by fastforce
  }
  ultimately
  show ?case by cases
qed

lemma exec_n_drop_right:
  assumes c @ P' ⊢ (0, s) →^n (j, s') j ∉ {0..<isize c}
  shows ∃ s'' i' k m.
          (if c = [] then s'' = s ∧ i' = 0 ∧ k = 0
           else c ⊢ (0, s) →^k (i', s'') ∧
           i' ∈ exits c) ∧
           c @ P' ⊢ (i', s'') →^m (j, s') ∧
           n = k + m
  using assms
  by (cases c = [])
     (auto dest: exec_n_split [where P=[], simplified])
```

Dropping the left context of a potentially incomplete execution of c.

```
lemma exec1_drop_left:
  assumes P1 @ P2 ⊢ (i, s, stk) → (n, s', stk') and isize P1 ≤ i
  shows P2 ⊢ (i − isize P1, s, stk) → (n − isize P1, s', stk')
proof −
  have i = isize P1 + (i − isize P1) by simp
  then obtain i' where i = isize P1 + i' ..
  moreover
  have n = isize P1 + (n − isize P1) by simp
  then obtain n' where n = isize P1 + n' ..
  ultimately
  show ?thesis using assms by (clarsimp simp del: iexec.simps)
qed

lemma exec_n_drop_left:
  assumes P @ P' ⊢ (i, s, stk) →^k (n, s', stk')
          isize P ≤ i exits P' ⊆ {0..}
  shows P' ⊢ (i − isize P, s, stk) →^k (n − isize P, s', stk')
using assms proof (induction k arbitrary: i s stk)
  case 0 thus ?case by simp
next
  case (Suc k)
  from Suc.prems
```

```
  obtain i' s'' stk'' where
    step: P @ P' ⊢ (i, s, stk) → (i', s'', stk'') and
    rest: P @ P' ⊢ (i', s'', stk'') →^k (n, s', stk')
    by (auto simp del: exec1_def)
  from step ‹isize P ≤ i›
  have P' ⊢ (i - isize P, s, stk) → (i' - isize P, s'', stk'')
    by (rule exec1_drop_left)
  moreover
  then have i' - isize P ∈ succs P' 0
    by (fastforce dest!: succs_iexec1 simp del: iexec.simps)
  with ‹exits P' ⊆ {0..}›
  have isize P ≤ i' by (auto simp: exits_def)
  from rest this ‹exits P' ⊆ {0..}›
  have P' ⊢ (i' - isize P, s'', stk'') →^k (n - isize P, s', stk')
    by (rule Suc.IH)
  ultimately
  show ?case by auto
qed

lemmas exec_n_drop_Cons =
  exec_n_drop_left [where P=[instr], simplified] for instr

definition
  closed P ⟷ exits P ⊆ {isize P}

lemma ccomp_closed [simp, intro!]: closed (ccomp c)
  using ccomp_exits by (auto simp: closed_def)

lemma acomp_closed [simp, intro!]: closed (acomp c)
  by (simp add: closed_def)

lemma exec_n_split_full:
  assumes exec: P @ P' ⊢ (0,s,stk) →^k (j, s', stk')
  assumes P: isize P ≤ j
  assumes closed: closed P
  assumes exits: exits P' ⊆ {0..}
  shows ∃ k1 k2 s'' stk''. P ⊢ (0,s,stk) →^k1 (isize P, s'', stk'') ∧
                           P' ⊢ (0,s'',stk'') →^k2 (j - isize P, s', stk')
proof (cases P)
  case Nil with exec
  show ?thesis by fastforce
next
  case Cons
  hence 0 < isize P by simp
```

with *exec P closed*
obtain *k1 k2 s'' stk''* **where**
1: $P \vdash (0,s,stk) \rightarrow\hat{}k1$ (*isize P*, s'', stk'') **and**
2: P @ $P' \vdash$ (*isize P*,s'',stk'') $\rightarrow\hat{}k2$ (j, s', stk')
by (*auto dest*!: *exec_n_split* [**where** *P*=[] **and** *i*=*0*, *simplified*]
simp: *closed_def*)
moreover
have j = *isize P* + (j − *isize P*) **by** *simp*
then obtain *j0* **where** j = *isize P* + *j0* **..**
ultimately
show *?thesis* **using** *exits*
by (*fastforce dest*: *exec_n_drop_left*)
qed

6.6 Correctness theorem

lemma *acomp_neq_Nil* [*simp*]:
acomp a ≠ []
by (*induct a*) *auto*

lemma *acomp_exec_n* [*dest*!]:
acomp a ⊢ (*0*,*s*,*stk*) →ˆ*n* (*isize* (*acomp a*),s',stk') ⟹
$s' = s \wedge stk' =$ *aval a s*#*stk*
proof (*induction a arbitrary*: *n* s' *stk* stk')
case (*Plus a1 a2*)
let *?sz* = *isize* (*acomp a1*) + (*isize* (*acomp a2*) + *1*)
from *Plus.prems*
have *acomp a1* @ *acomp a2* @ [*ADD*] ⊢ (*0*,*s*,*stk*) →ˆ*n* (*?sz*, s', stk')
by (*simp add*: *algebra_simps*)

then obtain *n1 s1 stk1 n2 s2 stk2 n3* **where**
acomp a1 ⊢ (*0*,*s*,*stk*) →ˆ*n1* (*isize* (*acomp a1*), *s1*, *stk1*)
acomp a2 ⊢ (*0*,*s1*,*stk1*) →ˆ*n2* (*isize* (*acomp a2*), *s2*, *stk2*)
[*ADD*] ⊢ (*0*,*s2*,*stk2*) →ˆ*n3* (*1*, s', stk')
by (*auto dest*!: *exec_n_split_full*)

thus *?case* **by** (*fastforce dest*: *Plus.IH simp*: *exec_n_simps*)
qed (*auto simp*: *exec_n_simps*)

lemma *bcomp_split*:
assumes *bcomp b c i* @ P' ⊢ (*0*, *s*, *stk*) →ˆ*n* (*j*, s', stk')
$j \notin$ {*0*..<*isize* (*bcomp b c i*)} $0 \le i$
shows ∃ s'' stk'' i' *k m*.
bcomp b c i ⊢ (*0*, *s*, *stk*) →ˆ*k* (i', s'', stk'') ∧

$(i' = isize\ (bcomp\ b\ c\ i) \vee i' = i + isize\ (bcomp\ b\ c\ i)) \wedge$
$bcomp\ b\ c\ i\ @\ P' \vdash (i', s'', stk'') \rightarrow\hat{}m\ (j, s', stk') \wedge$
$n = k + m$
using *assms* **by** (*cases bcomp b c i* = []) (*fastforce dest*!: *exec_n_drop_right*)+

lemma *bcomp_exec_n* [*dest*]:
assumes $bcomp\ b\ c\ j \vdash (0, s, stk) \rightarrow\hat{}n\ (i, s', stk')$
$isize\ (bcomp\ b\ c\ j) \leq i\ 0 \leq j$
shows $i = isize(bcomp\ b\ c\ j) + (if\ c = bval\ b\ s\ then\ j\ else\ 0) \wedge$
$s' = s \wedge stk' = stk$
using *assms* **proof** (*induction b arbitrary*: $c\ j\ i\ n\ s'\ stk'$)
case *Bc* **thus** *?case*
by (*simp split*: *split_if_asm add*: *exec_n_simps*)
next
case (*Not b*)
from *Not.prems* **show** *?case*
by (*fastforce dest*!: *Not.IH*)
next
case (*And b1 b2*)

let *?b2* = *bcomp b2 c j*
let *?m* = *if c then isize ?b2 else isize ?b2* + *j*
let *?b1* = *bcomp b1 False ?m*

have *j*: $isize\ (bcomp\ (And\ b1\ b2)\ c\ j) \leq i\ 0 \leq j$ **by** *fact*+

from *And.prems*
obtain $s''\ stk''\ i'\ k\ m$ **where**
b1: $?b1 \vdash (0, s, stk) \rightarrow\hat{}k\ (i', s'', stk'')$
$i' = isize\ ?b1 \vee i' = ?m + isize\ ?b1$ **and**
b2: $?b2 \vdash (i' - isize\ ?b1, s'', stk'') \rightarrow\hat{}m\ (i - isize\ ?b1, s', stk')$
by (*auto dest*!: *bcomp_split dest*: *exec_n_drop_left*)
from *b1 j*
have $i' = isize\ ?b1 + (if\ \neg bval\ b1\ s\ then\ ?m\ else\ 0) \wedge s'' = s \wedge stk'' =$
stk
by (*auto dest*!: *And.IH*)
with *b2 j*
show *?case*
by (*fastforce dest*!: *And.IH simp*: *exec_n_end split*: *split_if_asm*)
next
case *Less*
thus *?case* **by** (*auto dest*!: *exec_n_split_full simp*: *exec_n_simps*)
qed

lemma *ccomp_empty* [*elim*!]:
  ccomp c = [] $\Longrightarrow$ (*c*,*s*) $\Rightarrow$ *s*
  by (*induct c*) *auto*

declare *assign_simp* [*simp*]

lemma *ccomp_exec_n*:
  ccomp c $\vdash$ (*0*,*s*,*stk*) $\rightarrow$^*n* (*isize*(*ccomp c*),*t*,*stk*$'$)
  $\Longrightarrow$ (*c*,*s*) $\Rightarrow$ *t* $\wedge$ *stk*$'$=*stk*
proof (*induction c arbitrary*: *s t stk stk*$'$ *n*)
  case *SKIP*
  thus *?case* **by** *auto*
next
  case (*Assign x a*)
  thus *?case*
    by *simp* (*fastforce dest*!: *exec_n_split_full simp*: *exec_n_simps*)
next
  case (*Seq c1 c2*)
  thus *?case* **by** (*fastforce dest*!: *exec_n_split_full*)
next
  case (*If b c1 c2*)
  note *If*.*IH* [*dest*!]

  let *?if* = *IF b THEN c1 ELSE c2*
  let *?cs* = *ccomp ?if*
  let *?bcomp* = *bcomp b False* (*isize* (*ccomp c1*) + *1*)

  from ⟨*?cs* $\vdash$ (*0*,*s*,*stk*) $\rightarrow$^*n* (*isize ?cs*,*t*,*stk*$'$)⟩
  obtain *i*$'$ *k m s*$''$ *stk*$''$ **where**
    cs: *?cs* $\vdash$ (*i*$'$,*s*$''$,*stk*$''$) $\rightarrow$^*m* (*isize ?cs*,*t*,*stk*$'$) **and**
        ?bcomp $\vdash$ (*0*,*s*,*stk*) $\rightarrow$^*k* (*i*$'$, *s*$''$, *stk*$''$)
        i$'$ = *isize ?bcomp* $\vee$ *i*$'$ = *isize ?bcomp* + *isize* (*ccomp c1*) + *1*
    by (*auto dest*!: *bcomp_split*)

  hence *i*$'$:
    s$''$=*s stk*$''$ = *stk*
    i$'$ = (*if bval b s then isize ?bcomp else isize ?bcomp*+*isize*(*ccomp c1*)+*1*)
    by *auto*

  with *cs* **have** *cs*$'$:
    ccomp c1@*JMP* (*isize* (*ccomp c2*))#*ccomp c2* $\vdash$
      (*if bval b s then 0 else isize* (*ccomp c1*)+*1*, *s*, *stk*) $\rightarrow$^*m*
      (*1* + *isize* (*ccomp c1*) + *isize* (*ccomp c2*), *t*, *stk*$'$)
     by (*fastforce dest*: *exec_n_drop_left simp*: *exits_Cons isuccs_def alge-*

```
bra_simps)

  show ?case
  proof (cases bval b s)
    case True with cs'
    show ?thesis
      by simp
         (fastforce dest: exec_n_drop_right
                  split: split_if_asm simp: exec_n_simps)
  next
    case False with cs'
    show ?thesis
      by (auto dest!: exec_n_drop_Cons exec_n_drop_left
               simp: exits_Cons isuccs_def)
  qed
next
  case (While b c)

  from While.prems
  show ?case
  proof (induction n arbitrary: s rule: nat_less_induct)
    case (1 n)

    { assume ¬ bval b s
      with 1.prems
      have ?case
        by simp
           (fastforce dest!: bcomp_exec_n bcomp_split
                    simp: exec_n_simps)
    } moreover {
      assume b: bval b s
      let ?c0 = WHILE b DO c
      let ?cs = ccomp ?c0
      let ?bs = bcomp b False (isize (ccomp c) + 1)
      let ?jmp = [JMP (−((isize ?bs + isize (ccomp c) + 1)))]

      from 1.prems b
      obtain k where
        cs: ?cs ⊢ (isize ?bs, s, stk) →^k (isize ?cs, t, stk') and
        k:  k ≤ n
        by (fastforce dest!: bcomp_split)

      have ?case
      proof cases
```

```
      assume ccomp c = []
      with cs k
      obtain m where
        ?cs ⊢ (0,s,stk) →^m (isize (ccomp ?c0), t, stk')
        m < n
        by (auto simp: exec_n_step [where k=k])
      with 1.IH
      show ?case by blast
    next
      assume ccomp c ≠ []
      with cs
      obtain m m' s'' stk'' where
        c: ccomp c ⊢ (0, s, stk) →^m' (isize (ccomp c), s'', stk'') and
        rest: ?cs ⊢ (isize ?bs + isize (ccomp c), s'', stk'') →^m
                    (isize ?cs, t, stk') and
        m: k = m + m'
        by (auto dest: exec_n_split [where i=0, simplified])
      from c
      have (c,s) ⇒ s'' and stk: stk'' = stk
        by (auto dest!: While.IH)
      moreover
      from rest m k stk
      obtain k' where
        ?cs ⊢ (0, s'', stk) →^k' (isize ?cs, t, stk')
        k' < n
        by (auto simp: exec_n_step [where k=m])
      with 1.IH
      have (?c0, s'') ⇒ t ∧ stk' = stk by blast
      ultimately
      show ?case using b by blast
    qed
  }
  ultimately show ?case by cases
qed
qed

theorem ccomp_exec:
  ccomp c ⊢ (0,s,stk) →* (isize(ccomp c),t,stk') ⟹ (c,s) ⇒ t
  by (auto dest: exec_exec_n ccomp_exec_n)

corollary ccomp_sound:
  ccomp c ⊢ (0,s,stk) →* (isize(ccomp c),t,stk)  ⟷  (c,s) ⇒ t
  by (blast intro!: ccomp_exec ccomp_bigstep)
```

end

7 A Typed Language

theory *Types* **imports** *Star Complex_Main* **begin**

7.1 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*

7.2 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)

7.3 Syntax of Commands

datatype
$com = SKIP$
$\mid$ $Assign\ vname\ aexp$ $\quad (\_ ::= \_ [1000, 61]\ 61)$
$\mid$ $Seq\quad com\ \ com$ $\quad (\_;\ \_\ [60, 61]\ 60)$
$\mid$ $If\quad bexp\ com\ com$ $\quad (IF\ \_\ THEN\ \_\ ELSE\ \_\ [0, 0, 61]\ 61)$
$\mid$ $While\ \ bexp\ com$ $\quad (WHILE\ \_\ DO\ \_\ [0, 61]\ 61)$

7.4 Small-Step Semantics of Commands

inductive
$small\_step :: (com \times state) \Rightarrow (com \times state) \Rightarrow bool$ (**infix** $\to$ 55)
where
$Assign$: $taval\ a\ s\ v \Longrightarrow (x ::= a,\ s) \to (SKIP,\ s(x := v))$ |

$Seq1$: $(SKIP;c,s) \to (c,s)$ |
$Seq2$: $(c1,s) \to (c1',s') \Longrightarrow (c1;c2,s) \to (c1';c2,s')$ |

$IfTrue$: $tbval\ b\ s\ True \Longrightarrow (IF\ b\ THEN\ c1\ ELSE\ c2,s) \to (c1,s)$ |
$IfFalse$: $tbval\ b\ s\ False \Longrightarrow (IF\ b\ THEN\ c1\ ELSE\ c2,s) \to (c2,s)$ |

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

lemmas $small\_step\_induct = small\_step.induct[split\_format(complete)]$

7.5 The Type System

datatype $ty = Ity \mid Rty$

type_synonym $tyenv = vname \Rightarrow ty$

inductive $atyping :: tyenv \Rightarrow aexp \Rightarrow ty \Rightarrow bool$
$((1\_/ \vdash/ (\_ :/ \_))\ [50,0,50]\ 50)$
where
Ic_ty: $\Gamma \vdash Ic\ i : Ity$ |
Rc_ty: $\Gamma \vdash Rc\ r : Rty$ |
V_ty: $\Gamma \vdash V\ x : \Gamma\ x$ |
$Plus\_ty$: $\Gamma \vdash a1 : \tau \Longrightarrow \Gamma \vdash a2 : \tau \Longrightarrow \Gamma \vdash Plus\ a1\ a2 : \tau$

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* |
Seq_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*

7.6 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 Seq_ty thus ?case by simp (metis Seq1 Seq2)
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
```

8 Definite Initialization Analysis

```
theory Vars imports BExp
begin
```

8.1 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* :: $'a \Rightarrow$ *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* $\Rightarrow$ *vname set* **where**
vars (*N n*) = {} |
vars (*V x*) = {*x*} |
vars (*Plus* a_1 a_2) = *vars* a_1 $\cup$ *vars* a_2

instance ..

end

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

instantiation *bexp* :: *vars*
begin

fun *vars_bexp* :: *bexp* $\Rightarrow$ *vname set* **where**
vars (*Bc v*) = {} |
vars (*Not b*) = *vars b* |
vars (*And* b_1 b_2) = *vars* b_1 $\cup$ *vars* b_2 |
vars (*Less* a_1 a_2) = *vars* a_1 $\cup$ *vars* a_2

instance ..

end

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

abbreviation
eq_on :: $('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a\ set \Rightarrow bool$
((_ =/ _/ *on* _) [*50*,*0*,*50*] *50*) **where**
$f = g$ *on* $X == \forall\ x \in X.\ f\ x = g\ x$

lemma *aval_eq_if_eq_on_vars*[*simp*]:
$s_1 = s_2$ *on vars a* $\Longrightarrow$ *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* $\Longrightarrow$ *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_Init*
imports *Vars Com*
begin

8.2 Definite Initialization Analysis

inductive *D* :: *vname set* ⇒ *com* ⇒ *vname set* ⇒ *bool* **where**
Skip: *D A SKIP A* |
Assign: *vars a* ⊆ *A* $\Longrightarrow$ *D A* (*x* ::= *a*) (*insert x A*) |
Seq: ⟦ *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* ⊆ *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* ⊆ *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* ⊆ *A'*
by (*induct rule*: *D.induct*) *auto*

end

theory *Def_Init_Exp*
imports *Vars*

begin

8.3 Initialization-Sensitive Expressions Evaluation

type_synonym *state* = *vname* ⇒ *val option*

fun *aval* :: *aexp* ⇒ *state* ⇒ *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_Init_Big*
imports *Com Def_Init_Exp*
begin

8.4 Initialization-Sensitive Big Step Semantics

inductive
  big_step :: (*com* × *state option*) ⇒ *state option* ⇒ *bool* (**infix** ⇒ *55*)
where

None: $(c, None) \Rightarrow None$ |
Skip: $(SKIP, s) \Rightarrow s$ |
AssignNone: $aval\ a\ s = None \Longrightarrow (x ::= a,\ Some\ s) \Rightarrow None$ |
Assign: $aval\ a\ s = Some\ i \Longrightarrow (x ::= a,\ Some\ s) \Rightarrow Some(s(x := Some\ i))$
|
Seq: $(c_1, s_1) \Rightarrow s_2 \Longrightarrow (c_2, s_2) \Rightarrow s_3 \Longrightarrow (c_1; c_2, s_1) \Rightarrow s_3$ |

IfNone: $bval\ b\ s = None \Longrightarrow (IF\ b\ THEN\ c_1\ ELSE\ c_2, Some\ s) \Rightarrow None$ |
IfTrue: $[\![\ bval\ b\ s = Some\ True;\ (c_1, Some\ s) \Rightarrow s'\ ]\!] \Longrightarrow$
$(IF\ b\ THEN\ c_1\ ELSE\ c_2, Some\ s) \Rightarrow s'$ |
IfFalse: $[\![\ bval\ b\ s = Some\ False;\ (c_2, Some\ s) \Rightarrow s'\ ]\!] \Longrightarrow$
$(IF\ b\ THEN\ c_1\ ELSE\ c_2, Some\ s) \Rightarrow s'$ |

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_Init_Sound_Big*
imports *Def_Init Def_Init_Big*
begin

8.5 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 *Seq* **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*: $[\![\ D\ (dom\ s)\ c\ A';\ (c, Some\ s) \Rightarrow s'\ ]\!] \Longrightarrow s' \neq None$
by (*metis Sound not_Some_eq subset_refl*)

end

9 Live Variable Analysis

theory *Live* **imports** *Vars Big_Step*
begin

9.1 Liveness Analysis

fun *L* :: *com* ⇒ *vname set* ⇒ *vname set* **where**
$L\ SKIP\ X = X\ |$
$L\ (x ::= a)\ X = X - \{x\} \cup vars\ a\ |$
$L\ (c_1;\ c_2)\ X = (L\ c_1 \circ L\ c_2)\ X\ |$
$L\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ X = vars\ b \cup L\ c_1\ X \cup L\ c_2\ X\ |$
$L\ (WHILE\ b\ DO\ c)\ X = vars\ b \cup X \cup L\ c\ X$

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*)

9.2 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 (*Seq c1 s1 s2 c2 s3 X t1*)
  from *Seq.IH*(*1*) *Seq.prems* **obtain** *t2* **where**
    t12: (*c1*, *t1*) ⇒ *t2* **and** *s2t2*: *s2* = *t2 on L c2 X*
    by *simp blast*
  from *Seq.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.IH*[*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.IH*[*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

9.3 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 (Seq c1 s1 s2 c2 s3 X t1)
  from Seq.IH(1) Seq.prems obtain t2 where
    t12: (bury c1 (L c2 X), t1) ⇒ t2 and s2t2: s2 = t2 on L c2 X
    by simp blast
  from Seq.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.IH[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.IH[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* ⟷ *c* = *x*::=*a* & *x* : *X*
by (*cases c*) *auto*

lemma *Seq_bury*[*simp*]: bc_1;bc_2 = *bury c X* ⟷
(*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* ⟷
(*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* ⟷
(*EX c'*. *c* = *WHILE b DO c'* & *bc'* = *bury c'* (*vars b* ∪ *X* ∪ *L c X*))
by (*cases c*) *auto*

theorem *bury_sound2*:
(*bury c X*,*s*) ⇒ *s'* ⟹ *s* = *t on L c X* ⟹
∃ *t'*. (*c*,*t*) ⇒ *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 (*Seq 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* = *Seq.hyps*(*2*,*4*)
from *IH*(*1*)[*OF bc1*, *of t1*] *Seq.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*› ‹$(c', t1) \Rightarrow t2$› *c* **show** *?case* **by** *auto*
qed

corollary *final_bury_sound2*: $(bury\ c\ UNIV,s) \Rightarrow s' \Longrightarrow (c,s) \Rightarrow s'$
using *bury_sound2*[*of c UNIV*]
by (*auto simp*: *fun_eq_iff* [*symmetric*])

corollary *bury_iff*: $(bury\ c\ UNIV,s) \Rightarrow s' \longleftrightarrow (c,s) \Rightarrow s'$
by(*metis final_bury_sound final_bury_sound2*)

end

theory *Live_True*
imports ~~/*src*/*HOL*/*Library*/*While_Combinator Vars Big_Step*
begin

9.4 True Liveness Analysis

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

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 (λ*Y*. *X* ∪ *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* ∪ *X* ∪ *L c* (*L* (*WHILE b DO c*) *X*)
by(*metis lfp_unfold*[*OF mono_union_L*] *L.simps*(*5*))

9.5 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 (*Seq c1 s1 s2 c2 s3 X t1*)
from *Seq.IH*(*1*) *Seq.prems* **obtain** *t2* **where**
t12: (*c1*, *t1*) ⇒ *t2* **and** *s2t2*: *s2* = *t2 on L c2 X*
by *simp blast*
from *Seq.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* **and** *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.IH*[*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.IH*[*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
```

9.6 Executability

```
instantiation com :: vars
begin

fun vars_com :: com ⇒ vname set where
vars SKIP = {} |
vars (x::=e) = 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 L_subset_vars: L c X ⊆ vars c ∪ X
proof(induction c arbitrary: X)
  case (While b c)
  have lfp(λY. 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*

Some code generation magic: executing *lfp*

```
lemma lfp_while:
  assumes mono f and !!X. X ⊆ C ⟹ f X ⊆ C and finite C
  shows lfp f = while (λA. f A ≠ A) f {}
unfolding while_def using assms by (rule lfp_the_while_option) blast
```

Make *L* executable by replacing *lfp* with the *while* combinator from theory *While_Combinator*. The *while* combinator obeys the recursion equation

while b c s = (*if b s then while b c* (*c s*) *else s*)

and is thus executable.

```
lemma L_While: fixes b c X
assumes finite X defines f == λA. vars b ∪ X ∪ L c A
shows L (WHILE b DO c) X = while (λA. f A ≠ A) f {} (is _ = ?r)
proof −
  let ?V = vars b ∪ vars c ∪ X
  have lfp f = ?r
  proof(rule lfp_while[where C = ?V])
    show mono f by(simp add: f_def mono_union_L)
  next
    fix Y show Y ⊆ ?V ⟹ f Y ⊆ ?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

lemma L_While_set: L (WHILE b DO c) (set xs) =
  (let f = (λA. vars b ∪ set xs ∪ L c A)
   in while (λA. f A ≠ A) f {})
by(simp add: L_While del: L.simps(5))
```

Replace the equation for L WHILE by the executable *L_While_set*:

lemmas [*code*] = *L.simps*(*1*−*4*) *L_While_set*

Sorry, this syntax is odd.

```
lemma (let b = Less (N 0) (V ''y''); c = ''y'' ::= V ''x''; ''x'' ::= V ''z''
  in L (WHILE b DO c) {''y''}) = {''x'', ''y'', ''z''}
by eval
```

9.7 Limiting the number of iterations

The final parameter is the default value:

fun *iter* :: ('a ⇒ 'a) ⇒ nat ⇒ 'a ⇒ 'a ⇒ '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)

A version of *L* with a bounded number of iterations (here: 2) in the WHILE case:

fun *Lb* :: *com* ⇒ *vname set* ⇒ *vname set* **where**
Lb SKIP X = X |
Lb (x ::= a) X = (if x ∈ X then X − {x} ∪ vars a else X) |
Lb (c_1; c_2) *X = (Lb* c_1 ∘ *Lb* c_2) *X* |
Lb (IF b THEN c_1 *ELSE* c_2) *X = vars b ∪ Lb* c_1 *X ∪ Lb* c_2 *X* |
Lb (WHILE b DO c) X = iter (λA. vars b ∪ X ∪ Lb c A) 2 {} (vars b ∪ vars c ∪ X)

Lb (and *iter*) is not monotone!

lemma *let w = WHILE Bc False DO (''x'' ::= V ''y''; ''z'' ::= V ''x'')*
  in ¬ (Lb w {''z''} ⊆ Lb w {''y'',''z''})
by *eval*

lemma *lfp_subset_iter*:
  ⟦ *mono f*; !!*X*. *f X ⊆ f' X*; *lfp f ⊆ D* ⟧ ⟹ *lfp f ⊆ iter f' n A D*
proof(*induction n arbitrary*: *A*)
  case *0* **thus** *?case* **by** *simp*
next
  case *Suc* **thus** *?case* **by** *simp* (*metis lfp_lowerbound*)
qed

lemma *L c X ⊆ Lb c X*
proof(*induction c arbitrary*: *X*)
  case (*While b c*)
  let *?f = λA. vars b ∪ X ∪ L c A*
  let *?fb = λA. vars b ∪ X ∪ Lb c A*
  show *?case*
  proof (*simp*, *rule lfp_subset_iter*[*OF mono_union_L*])
    show !!*X*. *?f X ⊆ ?fb X* **using** *While.IH* **by** *blast*
    show *lfp ?f ⊆ vars b ∪ vars c ∪ X*
      by (*metis* (*full_types*) *L.simps*(*5*) *L_subset_vars vars_com.simps*(*5*))
  qed
next
  case *Seq* **thus** *?case* **by** *simp* (*metis* (*full_types*) *L_mono monoD subset_trans*)

qed *auto*

end

10 Security Type Systems

theory *Sec_Type_Expr* **imports** *Big_Step*
begin

10.1 Security Levels and Expressions

type_synonym *level* = *nat*

class *sec* =
fixes *sec* :: *'a* ⇒ *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:

instantiation *list* :: (*type*)*sec*
begin

definition *sec*(*x* :: *'a list*) = *length x*

instance ..

end

instantiation *aexp* :: *sec*
begin

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

instance ..

end

instantiation *bexp* :: *sec*
begin

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

instance ..

end

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 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 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

10.2 Syntax Directed Typing

inductive *sec_type* :: *nat* ⇒ *com* ⇒ *bool* ((_/ ⊢ _) [*0*,*0*] *50*) **where**
Skip:
l ⊢ *SKIP* |
Assign:
⟦ *sec x* ≥ *sec a*; *sec x* ≥ *l* ⟧ ⟹ *l* ⊢ *x* ::= *a* |
Seq:
⟦ *l* ⊢ c_1; *l* ⊢ c_2 ⟧ ⟹ *l* ⊢ c_1;c_2 |
If:

$\llbracket$ *max* (*sec b*) $l \vdash c_1$; *max* (*sec b*) $l \vdash c_2$ $\rrbracket \Longrightarrow l \vdash$ *IF b THEN* c_1 *ELSE* c_2 |
While:
max (*sec b*) $l \vdash c \Longrightarrow l \vdash$ *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*!]:
$l \vdash x$ ::= a $l \vdash c_1;c_2$ $l \vdash$ *IF b THEN* c_1 *ELSE* c_2 $l \vdash$ *WHILE b DO c*

An important property: anti-monotonicity.

lemma *anti_mono*: $\llbracket$ $l \vdash c$; $l' \le l$ $\rrbracket \Longrightarrow l' \vdash c$
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*))
apply (*metis If le_refl sup_mono sup_nat_def*)
apply (*metis While le_refl sup_mono sup_nat_def*)
done

lemma *confinement*: $\llbracket$ $(c,s) \Rightarrow t$; $l \vdash c$ $\rrbracket \Longrightarrow s = t$ $(< l)$
proof(*induction rule*: *big_step_induct*)
case *Skip* **thus** *?case* **by** *simp*
next
case *Assign* **thus** *?case* **by** *auto*
next
case *Seq* **thus** *?case* **by** *auto*
next
case (*IfTrue b s c1*)
hence *max* (*sec 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 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 max (sec 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 a using ‹0 ⊢ x ::= a› by auto
  show ?case
  proof auto
    assume sec x ≤ l
    with ‹sec x >= sec a› have sec 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 Seq thus ?case by blast
next
  case (IfTrue b s c1 s' c2)
  have sec b ⊢ c1 sec b ⊢ c2 using IfTrue.prems(2) by auto
  show ?case
  proof cases
    assume sec b ≤ l
    hence s = t (≤ sec 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 b ⊢ c1›  anti_mono
    show ?thesis by auto
  next
    assume ¬ sec b ≤ l
    have 1: sec b ⊢ IF b THEN c1 ELSE c2
      by(rule sec_type.intros)(simp_all add: ‹sec b ⊢ c1› ‹sec b ⊢ c2›)
    from confinement[OF IfTrue.hyps(2) ‹sec b ⊢ c1›] ‹¬ sec b ≤ l›
    have s = s' (≤ l) by auto
```

```
    moreover
    from confinement[OF IfTrue.prems(1) 1] ‹¬ sec 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 b ⊢ c1 sec b ⊢ c2 using IfFalse.prems(2) by auto
  show ?case
  proof cases
    assume sec b ≤ l
    hence s = t (≤ sec 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 b ⊢ c2› anti_mono
    show ?thesis by auto
  next
    assume ¬ sec b ≤ l
    have 1: sec b ⊢ IF b THEN c1 ELSE c2
      by(rule sec_type.intros)(simp_all add: ‹sec b ⊢ c1› ‹sec b ⊢ c2›)
    from confinement[OF big_step.IfFalse[OF IfFalse(1,2)] 1] ‹¬ sec b ≤ l›
    have s = s' (≤ l) by auto
    moreover
    from confinement[OF IfFalse.prems(1) 1] ‹¬ sec 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 b ⊢ c using WhileFalse.prems(2) by auto
  show ?case
  proof cases
    assume sec b ≤ l
    hence s = t (≤ sec 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 b ≤ l
    have 1: sec b ⊢ WHILE b DO c
      by(rule sec_type.intros)(simp_all add: ‹sec b ⊢ c›)
    from confinement[OF WhileFalse.prems(1) 1] ‹¬ sec 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 b* ⊢ *c* **using** *WhileTrue.prems(2)* **by** *auto*
show *?case*
proof *cases*
  assume *sec b* ≤ *l*
  hence *s1* = *t1* (≤ *sec 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 b* ⊢ *c*›]
      ‹*s1* = *t1* (≤ *l*)›]]
  show *?thesis* **by** *simp*
next
  assume ¬ *sec b* ≤ *l*
  have *1*: *sec b* ⊢ *?w* **by**(*rule sec_type.intros*)(*simp_all add*: ‹*sec b* ⊢ *c*›)
  from *confinement*[*OF big_step.WhileTrue*[*OF WhileTrue.hyps*] *1*] ‹¬ *sec b* ≤ *l*›
  have *s1* = *s3* (≤ *l*) **by** *auto*
  moreover
  from *confinement*[*OF WhileTrue.prems(1) 1*] ‹¬ *sec b* ≤ *l*›
  have *t1* = *t3* (≤ *l*) **by** *auto*
  ultimately show *s3* = *t3* (≤ *l*) **using** ‹*s1* = *t1* (≤ *l*)› **by** *auto*
qed
qed

10.3 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 a*; *sec x* ≥ *l* ⟧ ⟹ $l \vdash' x ::= a$ |
Seq′:
  ⟦ $l \vdash' c_1$; $l \vdash' c_2$ ⟧ ⟹ $l \vdash' c_1;c_2$ |
If′:
  ⟦ *sec b* ≤ *l*; $l \vdash' c_1$; $l \vdash' c_2$ ⟧ ⟹ $l \vdash'$ *IF b THEN* c_1 *ELSE* c_2 |
While′:

$[\![$ *sec b* $\leq l$; $l \vdash' c$ $]\!] \Longrightarrow l \vdash'$ *WHILE b DO c* |
anti_mono':
$[\![$ $l \vdash' c$; $l' \leq l$ $]\!] \Longrightarrow l' \vdash' c$

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 Seq'*)
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 Seq*)
apply (*metis min_max.sup_absorb2 If*)
apply (*metis min_max.sup_absorb2 While*)
by (*metis anti_mono*)

10.4 A Bottom-Up Typing System

inductive *sec_type2* :: *com* ⇒ *level* ⇒ *bool* ((⊢ _ : _) [*0*,*0*] *50*) **where**
Skip2:
⊢ *SKIP* : *l* |
Assign2:
sec x $\geq$ *sec a* $\Longrightarrow \vdash$ *x* ::= *a* : *sec x* |
Seq2:
$[\![\vdash c_1 : l_1;\ \vdash c_2 : l_2]\!] \Longrightarrow \vdash c_1;c_2 : min\ l_1\ l_2$ |
If2:
$[\![$ *sec b* $\leq min\ l_1\ l_2;\ \vdash c_1 : l_1;\ \vdash c_2 : l_2]\!]$
$\Longrightarrow \vdash$ *IF b THEN* c_1 *ELSE* $c_2 : min\ l_1\ l_2$ |
While2:
$[\![$ *sec b* $\leq l;\ \vdash c : l]\!] \Longrightarrow \vdash$ *WHILE b DO c* : *l*

lemma *sec_type2_sec_type'*: $\vdash c : l \Longrightarrow l \vdash' c$
apply(*induction rule*: *sec_type2.induct*)
apply (*metis Skip'*)
apply (*metis Assign' eq_imp_le*)

apply (*metis Seq′ 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*: $l \vdash' c \Longrightarrow \exists\ l' \geq l.\ \vdash c : l'$
apply(*induction rule*: *sec_type′.induct*)
apply (*metis Skip2 le_refl*)
apply (*metis Assign2*)
apply (*metis Seq2 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

10.5 A Termination-Sensitive Syntax Directed System

inductive *sec_type* :: *nat* ⇒ *com* ⇒ *bool* ((_/ ⊢ _) [*0*,*0*] *50*) **where**
Skip:
  $l \vdash SKIP$ |
Assign:
  $[\![\ sec\ x \geq sec\ a;\ \ sec\ x \geq l\ ]\!] \Longrightarrow l \vdash x ::= a$ |
Seq:
  $l \vdash c_1 \Longrightarrow l \vdash c_2 \Longrightarrow l \vdash c_1;c_2$ |
If:
  $[\![\ max\ (sec\ b)\ l \vdash c_1;\ \ max\ (sec\ b)\ l \vdash c_2\ ]\!]$
  $\Longrightarrow l \vdash IF\ b\ THEN\ c_1\ ELSE\ c_2$ |
While:
  $sec\ b = 0 \Longrightarrow 0 \vdash c \Longrightarrow 0 \vdash WHILE\ b\ DO\ c$

code_pred (*expected_modes*: *i* => *i* => *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*))

```
apply (metis If le_refl sup_mono sup_nat_def)
by (metis While le_0_eq)


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 Seq thus ?case by auto
next
  case (IfTrue b s c1)
  hence max (sec 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 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 l ⊢ c by auto
  thus ?case using WhileTrue by metis
qed

lemma termi_if_non0: l ⊢ c ⟹ l ≠ 0 ⟹ ∃ t. (c,s) ⇒ t
apply(induction arbitrary: s rule: sec_type.induct)
apply (metis big_step.Skip)
apply (metis big_step.Assign)
apply (metis big_step.Seq)
apply (metis IfFalse IfTrue le0 le_antisym le_maxI2)
apply simp
done

theorem noninterference: (c,s) ⇒ s' ⟹ 0 ⊢ c ⟹  s = t (≤ l)
  ⟹ ∃ t'. (c,t) ⇒ t' ∧ s' = t' (≤ 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 a using ‹0 ⊢ x ::= a› by auto
  have (x ::= a,t) ⇒ t(x := aval a t) by auto
  moreover
  have s(x := aval a s) = t(x := aval a t) (≤ l)
  proof auto
    assume sec x ≤ l
    with ‹sec x ≥ sec a› have sec 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
  ultimately show ?case by blast
next
  case Seq thus ?case by blast
next
  case (IfTrue b s c1 s' c2)
  have sec b ⊢ c1 sec b ⊢ c2 using IfTrue.prems by auto
  obtain t' where t': (c1, t) ⇒ t' s' = t' (≤ l)
    using IfTrue(3)[OF anti_mono[OF ‹sec b ⊢ c1›] IfTrue.prems(2)] by
blast
  show ?case
  proof cases
    assume sec b ≤ l
    hence s = t (≤ sec 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 b ≤ l
    hence 0: sec b ≠ 0 by arith
    have 1: sec b ⊢ IF b THEN c1 ELSE c2
      by(rule sec_type.intros)(simp_all add: ‹sec b ⊢ c1› ‹sec b ⊢ c2›)
    from confinement[OF big_step.IfTrue[OF IfTrue(1,2)] 1] ‹¬ sec 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 b ≤ 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 b ⊢ c1 sec 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 b ⊢ c2›] IfFalse.prems(2)] by
blast
  show ?case
  proof cases
    assume sec b ≤ l
    hence s = t (≤ sec 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 b ≤ l
    hence 0: sec b ≠ 0 by arith
    have 1: sec b ⊢ IF b THEN c1 ELSE c2
      by(rule sec_type.intros)(simp_all add: ‹sec b ⊢ c1› ‹sec b ⊢ c2›)
    from confinement[OF big_step.IfFalse[OF IfFalse(1,2)] 1] ‹¬ sec 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 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 b = 0 by auto
  have s = t (≤ sec 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 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

10.6 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':
 $l \vdash' SKIP$ |
Assign':
 ⟦ $sec\ x \geq sec\ a$; $sec\ x \geq l$ ⟧ ⟹ $l \vdash' x ::= a$ |
Seq':
 $l \vdash' c_1 \Longrightarrow l \vdash' c_2 \Longrightarrow l \vdash' c_1;c_2$ |
If':
 ⟦ $sec\ b \leq l$; $l \vdash' c_1$; $l \vdash' c_2$ ⟧ ⟹ $l \vdash' IF\ b\ THEN\ c_1\ ELSE\ c_2$ |
While':
 ⟦ $sec\ b = 0$; $0 \vdash' c$ ⟧ ⟹ $0 \vdash' WHILE\ b\ DO\ c$ |
anti_mono':
 ⟦ $l \vdash' c$; $l' \leq l$ ⟧ ⟹ $l' \vdash' c$

lemma $l \vdash c \Longrightarrow l \vdash' c$
apply(*induction rule*: *sec_type.induct*)
apply (*metis Skip'*)
apply (*metis Assign'*)
apply (*metis Seq'*)
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 Seq*)
apply (*metis min_max.sup_absorb2 If*)
apply (*metis While*)
by (*metis anti_mono*)

end

11 Hoare Logic

theory *Hoare* **imports** *Big_Step* **begin**

11.1 Hoare Logic for Partial Correctness

type_synonym *assn* = *state* ⇒ *bool*

abbreviation *state_subst* :: *state* ⇒ *aexp* ⇒ *vname* ⇒ *state*
(_[_'/_] [1000,0,0] 999)
where $s[a/x]$ == $s(x := aval\ a\ s)$

inductive
hoare :: *assn* ⇒ *com* ⇒ *assn* ⇒ *bool* (⊢ ({(1_)}/ (_)/ {(1_)}) 50)
where
Skip: ⊢ {P} SKIP {P} |

Assign: ⊢ {λs. P(s[a/x])} x::=a {P} |

Seq: ⟦ ⊢ {P} c_1 {Q}; ⊢ {Q} c_2 {R} ⟧
⟹ ⊢ {P} c_1;c_2 {R} |

If: ⟦ ⊢ {λs. P s ∧ bval b s} c_1 {Q}; ⊢ {λs. P s ∧ ¬ bval b s} c_2 {Q} ⟧
⟹ ⊢ {P} IF b THEN c_1 ELSE c_2 {Q} |

While: ⊢ {λs. P s ∧ bval b s} c {P} ⟹
⊢ {P} WHILE b DO c {λs. P s ∧ ¬ bval b s} |

conseq: ⟦ ∀ s. P' s ⟶ P s; ⊢ {P} c {Q}; ∀ s. Q s ⟶ Q' s ⟧
⟹ ⊢ {P'} c {Q'}

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

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

lemma *strengthen_pre*:
⟦ ∀ s. P' s ⟶ P s; ⊢ {P} c {Q} ⟧ ⟹ ⊢ {P'} c {Q}
by (*blast intro*: *conseq*)

lemma *weaken_post*:

$\llbracket \vdash \{P\}\ c\ \{Q\};\ \ \forall s.\ Q\ s \longrightarrow Q'\ s\ \rrbracket \Longrightarrow\ \vdash \{P\}\ c\ \{Q'\}$
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**

11.2 Example: Sums

Summing up the first n natural numbers. The sum is accumulated in variable x, the loop counter is variable y.

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 $\{m..n\}$.

11.2.1 Proof by Operational Semantics

The behaviour of the loop is proved by induction:

lemma *setsum_head_plus_1*:
  $m \le n \Longrightarrow setsum\ f\ \{m..n\} = f\ m + setsum\ f\ \{m{+}1..n::int\}$
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

11.2.2 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.Seq*)
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 Seq*)
prefer *2*
apply(*rule Assign*)
apply(*rule Assign'*)
apply(*fastforce simp*: *atLeastAtMostPlus1_int_conv algebra_simps*)
apply(*fastforce*)
apply(*rule Seq*)
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**

11.3 Soundness

definition
hoare_valid :: *assn* ⇒ *com* ⇒ *assn* ⇒ *bool* (⊨ {(*1*_)}/ (_)/ {(*1*_)} *50*) **where**
⊨ {*P*}*c*{*Q*} = (∀ *s t*. (*c*,*s*) ⇒ *t* ⟶ *P s* ⟶ *Q t*)

lemma *hoare_sound*: ⊢ {*P*}*c*{*Q*} ⟹ ⊨ {*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*)

11.4 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_Seq*[*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)
```

11.5 Completeness

```
lemma wp_is_pre: ⊢ {wp c Q} c {Q}
proof(induction c arbitrary: Q)
  case Seq thus ?case by(auto intro: Seq)
next
  case (If b c1 c2)
  let ?If = IF b THEN c1 ELSE c2
  show ?case
  proof(rule hoare.If)
    show ⊢ {λs. wp ?If Q s ∧ bval b s} c1 {Q}
    proof(rule strengthen_pre[OF _ If(1)])
      show ∀ s. wp ?If Q s ∧ bval b s ⟶ wp c1 Q s by auto
    qed
    show ⊢ {λs. wp ?If Q s ∧ ¬ bval b s} c2 {Q}
    proof(rule strengthen_pre[OF _ If(2)])
      show ∀ s. wp ?If Q s ∧ ¬ bval b s ⟶ wp c2 Q s by auto
    qed
  qed
next
  case (While b c)
  let ?w = WHILE b DO c
  have ⊢ {wp ?w Q} ?w {λs. wp ?w Q s ∧ ¬ bval b s}
  proof(rule hoare.While)
    show ⊢ {λs. wp ?w Q s ∧ bval b s} c {wp ?w Q}
    proof(rule strengthen_pre[OF _ While(1)])
      show ∀ s. wp ?w Q s ∧ bval b s ⟶ wp c (wp ?w Q) s by auto
    qed
  qed
```

```
  thus ?case
  proof(rule weaken_post)
    show ∀ s. wp ?w Q s ∧ ¬ bval b s ⟶ Q s by auto
  qed
qed auto

lemma hoare_relative_complete: assumes ⊨ {P}c{Q} shows ⊢ {P}c{Q}
proof(rule strengthen_pre)
  show ∀ s. P s ⟶ wp c Q s using assms
    by (auto simp: hoare_valid_def wp_def)
  show ⊢ {wp c Q} c {Q} by(rule wp_is_pre)
qed

end
```

```
theory VC imports Hoare begin
```

11.6 Verification Conditions

Annotated commands: commands where loops are annotated with invariants.

```
datatype acom =
  ASKIP |
  Aassign vname aexp     ((_ ::= _) [1000, 61] 61) |
  Aseq   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 = (λs. Q(s(x := aval a s))) |
pre (Aseq c₁ c₂) Q = pre c₁ (pre c₂ Q) |
pre (Aif b c₁ c₂) Q =
  (λs. (bval b s ⟶ pre c₁ Q s) ∧
       (¬ bval b s ⟶ pre c₂ Q s)) |
pre (Awhile I b c) Q = I
```

Verification condition:

```
fun vc :: acom ⇒ assn ⇒ assn where
vc ASKIP Q = (λs. True) |
vc (Aassign x a) Q = (λs. True) |
```

$vc\ (Aseq\ c_1\ c_2)\ Q = (\lambda s.\ vc\ c_1\ (pre\ c_2\ Q)\ s \wedge vc\ c_2\ Q\ s)\ |$
$vc\ (Aif\ b\ c_1\ c_2)\ Q = (\lambda s.\ vc\ c_1\ Q\ s \wedge vc\ c_2\ Q\ s)\ |$
$vc\ (Awhile\ I\ b\ c)\ Q =$
$\quad (\lambda s.\ (I\ s \wedge \neg\ bval\ b\ s \longrightarrow Q\ s) \wedge$
$\qquad (I\ s \wedge bval\ b\ s \longrightarrow pre\ c\ I\ s) \wedge$
$\qquad vc\ c\ I\ s)$

Strip annotations:

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

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*)

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 *Aseq* **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 Aseq thus ?case by simp (metis pre_mono)
qed simp_all

lemma vc_complete:
⊢ {P}c{Q} ⟹ ∃ c'. strip c' = c ∧ (∀ s. vc c' Q s) ∧ (∀ s. P s ⟶ pre c'
Q s)
  (is _ ⟹ ∃ c'. ?G P c Q c')
proof (induction rule: hoare.induct)
  case Skip
  show ?case (is ∃ 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 (Seq P c1 Q c2 R)
  from Seq.IH obtain ac1 where ih1: ?G P c1 Q ac1 by blast
  from Seq.IH obtain ac2 where ih2: ?G Q c2 R ac2 by blast
  show ?case (is ∃ ac. ?C ac)
  proof
    show ?C(Aseq 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 (λs. P s ∧ bval b s) c1 Q ac1
    by blast
  from If.IH obtain ac2 where ih2: ?G (λs. 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 (λs. 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
```

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 (*Aseq* 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 (λ*s*. vc_1 *s* ∧ vc_2 *s*, λ*s*. (*bval b s* ⟶ wp_1 *s*) ∧ (¬*bval b s* ⟶ wp_2 *s*)))
|
vcpre (*Awhile I b c*) *Q* =
  (*let* (*vcc*,*wpc*) = *vcpre c I*
   in (λ*s*. (*I s* ∧ ¬ *bval b s* ⟶ *Q s*) ∧
        (*I s* ∧ *bval b s* ⟶ *wpc s*) ∧ *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

theory *HoareT* **imports** *Hoare_Sound_Complete* **begin**

11.7 Hoare Logic for Total Correctness

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*} ≡ ∀*s*. *P s* ⟶ (∃*t*. (*c*,*s*) ⇒ *t* ∧ *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 ⊢ 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: $\vdash_t \{\lambda s.\ P(s[a/x])\}\ x{::=}a\ \{P\}$ |
Seq: $[\![\vdash_t \{P_1\}\ c_1\ \{P_2\}; \vdash_t \{P_2\}\ c_2\ \{P_3\}]\!] \Longrightarrow \vdash_t \{P_1\}\ c_1;c_2\ \{P_3\}$ |
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 $f :: state \Rightarrow nat$ 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 Seq*)
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 Seq*)
prefer *2*

apply(*rule Assign*)
apply(*rule Assign'*)
apply (*simp add*: *atLeastAtMostPlus1_int_conv algebra_simps*)
apply *clarsimp*
apply *fastforce*
apply(*rule Seq*)
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* = (λ*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*: $\llbracket$ *bval b s*; (*c*, *s*) ⇒ *s'*; (*WHILE b DO c*, *s'*) ⇒ *t*$\rrbracket$
⟹ *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*: $\vdash_t$ {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 *Seq* **thus** *?case* **by** *simp* (*blast intro*:*hoaret.Seq*)
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*: $\models_t$ {*P*}*c*{*Q*} ⟹ $\vdash_t$ {*P*}*c*{*Q*}
apply(*rule strengthen_pre*[*OF _ wpt_is_pre*])
apply(*auto simp*: *hoare_tvalid_def hoare_valid_def wpt_def*)
done

end

12 Abstract Interpretation

theory *Complete_Lattice*
imports *Main*
begin

locale *Complete_Lattice* =
fixes L :: $'a$::*order set* **and** *Glb* :: $'a$ *set* $\Rightarrow$ $'a$
assumes *Glb_lower*: $A \subseteq L \Longrightarrow a \in A \Longrightarrow$ *Glb* $A \leq a$
and *Glb_greatest*: $b : L \Longrightarrow \forall\, a{\in}A.\ b \leq a \Longrightarrow b \leq$ *Glb* A
and *Glb_in_L*: $A \subseteq L \Longrightarrow$ *Glb* $A : L$
begin

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

lemma *index_lfp*: *lfp* $f : L$
by(*auto simp*: *lfp_def intro*: *Glb_in_L*)

lemma *lfp_lowerbound*:
  $[\![\ a : L;\ \ f\ a \leq a\ ]\!] \Longrightarrow$ *lfp* $f \leq a$
by (*auto simp add*: *lfp_def intro*: *Glb_lower*)

lemma *lfp_greatest*:
  $[\![\ a : L;\ \ \bigwedge u.\ [\![\ u : L;\ f\ u \leq u]\!] \Longrightarrow a \leq u\ ]\!] \Longrightarrow a \leq$ *lfp* f
by (*auto simp add*: *lfp_def intro*: *Glb_greatest*)

lemma *lfp_unfold*: **assumes** $\bigwedge x.\ f\ x : L \longleftrightarrow x : L$
and *mono*: *mono* f **shows** *lfp* $f = f$ (*lfp* f)
proof−
  note *assms*(*1*)[*simp*] *index_lfp*[*simp*]
  have *1*: f (*lfp* f) $\leq$ *lfp* f
    apply(*rule lfp_greatest*)
    apply *simp*
    by (*blast intro*: *lfp_lowerbound monoD*[*OF mono*] *order_trans*)
  have *lfp* $f \leq f$ (*lfp* f)
    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

12.1 Annotated Commands

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

fun *post* :: *'a acom* ⇒ *'a* **where**
post (*SKIP* {*P*}) = *P* |
post (*x* ::= *e* {*P*}) = *P* |
post (C_1; C_2) = *post* C_2 |
post (*IF b THEN* {P_1} C_1 *ELSE* {P_2} C_2 {*Q*}) = *Q* |
post ({*I*} *WHILE b DO* {*P*} *C* {*Q*}) = *Q*

fun *strip* :: *'a acom* ⇒ *com* **where**
strip (*SKIP* {*P*}) = *com.SKIP* |
strip (*x* ::= *e* {*P*}) = *x* ::= *e* |
strip (C_1;C_2) = *strip* C_1; *strip* C_2 |
strip (*IF b THEN* {P_1} C_1 *ELSE* {P_2} C_2 {*P*}) =
  IF b THEN strip C_1 *ELSE strip* C_2 |
strip ({*I*} *WHILE b DO* {*P*} *C* {*Q*}) = *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 (c_1;c_2) = *anno A* c_1; *anno A* c_2 |
anno A (*IF b THEN* c_1 *ELSE* c_2) =
  IF b THEN {*A*} *anno A* c_1 *ELSE* {*A*} *anno A* c_2 {*A*} |
anno A (*WHILE b DO c*) =
  {*A*} *WHILE b DO* {*A*} *anno A c* {*A*}

fun *annos* :: *'a acom* ⇒ *'a list* **where**
annos (*SKIP* {*P*}) = [*P*] |

$annos\ (x ::= e\ \{P\}) = [P]\ |$
$annos\ (C_1;C_2) = annos\ C_1\ @\ annos\ C_2\ |$
$annos\ (IF\ b\ THEN\ \{P_1\}\ C_1\ ELSE\ \{P_2\}\ C_2\ \{Q\}) =$
$\quad P_1\ \#\ P_2\ \#\ Q\ \#\ annos\ C_1\ @\ annos\ C_2\ |$
$annos\ (\{I\}\ WHILE\ b\ DO\ \{P\}\ C\ \{Q\}) = I\ \#\ P\ \#\ Q\ \#\ annos\ C$

fun *map_acom* :: $('a \Rightarrow 'b) \Rightarrow 'a\ acom \Rightarrow 'b\ acom$ **where**
$map\_acom\ f\ (SKIP\ \{P\}) = SKIP\ \{f\ P\}\ |$
$map\_acom\ f\ (x ::= e\ \{P\}) = x ::= e\ \{f\ P\}\ |$
$map\_acom\ f\ (C_1;C_2) = map\_acom\ f\ C_1;\ map\_acom\ f\ C_2\ |$
$map\_acom\ f\ (IF\ b\ THEN\ \{P_1\}\ C_1\ ELSE\ \{P_2\}\ C_2\ \{Q\}) =$
$\quad IF\ b\ THEN\ \{f\ P_1\}\ map\_acom\ f\ C_1\ ELSE\ \{f\ P_2\}\ map\_acom\ f\ C_2$
$\quad \{f\ Q\}\ |$
$map\_acom\ f\ (\{I\}\ WHILE\ b\ DO\ \{P\}\ C\ \{Q\}) =$
$\quad \{f\ I\}\ WHILE\ b\ DO\ \{f\ P\}\ map\_acom\ f\ C\ \{f\ Q\}$

lemma *post_map_acom*[*simp*]: $post(map\_acom\ f\ C) = f(post\ C)$
by (*induction C*) *simp_all*

lemma *strip_acom*[*simp*]: $strip\ (map\_acom\ f\ C) = strip\ C$
by (*induction C*) *auto*

lemma *map_acom_SKIP*:
$map\_acom\ f\ C = SKIP\ \{S'\} \longleftrightarrow (\exists S.\ C = SKIP\ \{S\} \wedge S' = f\ S)$
by (*cases C*) *auto*

lemma *map_acom_Assign*:
$map\_acom\ f\ C = x ::= e\ \{S'\} \longleftrightarrow (\exists S.\ C = x{::=}e\ \{S\} \wedge S' = f\ S)$
by (*cases C*) *auto*

lemma *map_acom_Seq*:
$map\_acom\ f\ C = C1';C2' \longleftrightarrow$
$(\exists C1\ C2.\ C = C1;C2 \wedge map\_acom\ f\ C1 = C1' \wedge map\_acom\ f\ C2 = C2')$
by (*cases C*) *auto*

lemma *map_acom_If*:
$map\_acom\ f\ C = IF\ b\ THEN\ \{P1'\}\ C1'\ ELSE\ \{P2'\}\ C2'\ \{Q'\} \longleftrightarrow$
$(\exists P1\ P2\ C1\ C2\ Q.\ C = IF\ b\ THEN\ \{P1\}\ C1\ ELSE\ \{P2\}\ C2\ \{Q\} \wedge$
$\quad map\_acom\ f\ C1 = C1' \wedge map\_acom\ f\ C2 = C2' \wedge P1' = f\ P1 \wedge P2'$
$= f\ P2 \wedge Q' = f\ Q)$
by (*cases C*) *auto*

lemma *map_acom_While*:
$map\_acom\ f\ w = \{I'\}\ WHILE\ b\ DO\ \{p'\}\ C'\ \{P'\} \longleftrightarrow$

$(\exists I\ p\ P\ C.\ w = \{I\}$ *WHILE b DO* $\{p\}\ C\ \{P\} \wedge$ *map_acom f C* $= C' \wedge$ $I' = f\ I \wedge p' = f\ p \wedge P' = f\ P)$
by (*cases w*) *auto*

lemma *strip_anno*[*simp*]: *strip* (*anno a c*) = *c*
by(*induct c*) *simp_all*

lemma *strip_eq_SKIP*:
strip C = *com.SKIP* $\longleftrightarrow$ (*EX P. C* = *SKIP* $\{P\}$)
by (*cases C*) *simp_all*

lemma *strip_eq_Assign*:
strip C = *x::=e* $\longleftrightarrow$ (*EX P. C* = *x::=e* $\{P\}$)
by (*cases C*) *simp_all*

lemma *strip_eq_Seq*:
strip C = *c1*;*c2* $\longleftrightarrow$ (*EX C1 C2. C* = *C1*;*C2* & *strip C1* = *c1* & *strip C2* = *c2*)
by (*cases C*) *simp_all*

lemma *strip_eq_If*:
strip C = *IF b THEN c1 ELSE c2* $\longleftrightarrow$
(*EX P1 P2 C1 C2 Q. C* = *IF b THEN* $\{P1\}$ *C1 ELSE* $\{P2\}$ *C2* $\{Q\}$ & *strip C1* = *c1* & *strip C2* = *c2*)
by (*cases C*) *simp_all*

lemma *strip_eq_While*:
strip C = *WHILE b DO c1* $\longleftrightarrow$
(*EX I P C1 Q. C* = $\{I\}$ *WHILE b DO* $\{P\}$ *C1* $\{Q\}$ & *strip C1* = *c1*)
by (*cases C*) *simp_all*

lemma *set_annos_anno*[*simp*]: *set* (*annos* (*anno a c*)) = $\{a\}$
by(*induction c*)(*auto*)

lemma *size_annos_same*: *strip C1* = *strip C2* $\Longrightarrow$ *size*(*annos C1*) = *size*(*annos C2*)
apply(*induct C2 arbitrary*: *C1*)
apply (*auto simp*: *strip_eq_SKIP strip_eq_Assign strip_eq_Seq strip_eq_If strip_eq_While*)
done

lemmas *size_annos_same2* = *eqTrueI*[*OF size_annos_same*]

end

theory *Collecting*
imports *Complete_Lattice Big_Step ACom*
begin

12.2 Collecting Semantics of Commands

12.2.1 Annotated commands as a complete lattice

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

fun *less_eq_acom* :: ($'a$::*order*)*acom* ⇒ $'a$ *acom* ⇒ *bool* **where**
(*SKIP* {*P*}) ≤ (*SKIP* {*P'*}) = (*P* ≤ *P'*) |
(*x* ::= *e* {*P*}) ≤ (*x'* ::= *e'* {*P'*}) = (*x*=*x'* ∧ *e*=*e'* ∧ *P* ≤ *P'*) |
(*C1*;*C2*) ≤ (*C1'*;*C2'*) = (*C1* ≤ *C1'* ∧ *C2* ≤ *C2'*) |
(*IF b THEN* {*P1*} *C1 ELSE* {*P2*} *C2* {*Q*}) ≤ (*IF b' THEN* {*P1'*} *C1'* *ELSE* {*P2'*} *C2'* {*Q'*}) =
  (*b*=*b'* ∧ *P1* ≤ *P1'* ∧ *C1* ≤ *C1'* ∧ *P2* ≤ *P2'* ∧ *C2* ≤ *C2'* ∧ *Q* ≤ *Q'*) |
({*I*} *WHILE b DO* {*P*} *C* {*Q*}) ≤ ({*I'*} *WHILE b' DO* {*P'*} *C'* {*Q'*}) =
  (*b*=*b'* ∧ *C* ≤ *C'* ∧ *I* ≤ *I'* ∧ *P* ≤ *P'* ∧ *Q* ≤ *Q'*) |
less_eq_acom _ _ = *False*

lemma *SKIP_le*: *SKIP* {*S*} ≤ *c* ⟷ (∃ *S'*. *c* = *SKIP* {*S'*} ∧ *S* ≤ *S'*)
by (*cases c*) *auto*

lemma *Assign_le*: *x* ::= *e* {*S*} ≤ *c* ⟷ (∃ *S'*. *c* = *x* ::= *e* {*S'*} ∧ *S* ≤ *S'*)
by (*cases c*) *auto*

lemma *Seq_le*: *C1*;*C2* ≤ *C* ⟷ (∃ *C1' C2'*. *C* = *C1'*;*C2'* ∧ *C1* ≤ *C1'* ∧ *C2* ≤ *C2'*)
by (*cases C*) *auto*

lemma *If_le*: *IF b THEN* {*p1*} *C1 ELSE* {*p2*} *C2* {*S*} ≤ *C* ⟷
  (∃ *p1' p2' C1' C2' S'*. *C* = *IF b THEN* {*p1'*} *C1' ELSE* {*p2'*} *C2'* {*S'*} ∧
    p1 ≤ *p1'* ∧ *p2* ≤ *p2'* ∧ *C1* ≤ *C1'* ∧ *C2* ≤ *C2'* ∧ *S* ≤ *S'*)
by (*cases C*) *auto*

lemma *While_le*: {*I*} *WHILE b DO* {*p*} *C* {*P*} ≤ *W* ⟷
  (∃ *I' p' C' P'*. *W* = {*I'*} *WHILE b DO* {*p'*} *C'* {*P'*} ∧ *C* ≤ *C'* ∧ *p* ≤ *p'* ∧ *I* ≤ *I'* ∧ *P* ≤ *P'*)
by (*cases W*) *auto*

definition *less_acom* :: *'a acom* ⇒ *'a acom* ⇒ *bool* **where**
less_acom x y = (*x* ≤ *y* ∧ ¬ *y* ≤ *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 Seq_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* ⇒ *'a acom* **where**
sub_1(*C1*;*C2*) = *C1* |
sub_1(*IF b THEN* {*P1*} *C1 ELSE* {*P2*} *C2* {*Q*}) = *C1* |
sub_1({*I*} *WHILE b DO* {*P*} *C* {*Q*}) = *C*

fun sub_2 :: *'a acom* ⇒ *'a acom* **where**
sub_2(*C1*;*C2*) = *C2* |
sub_2(*IF b THEN* {*P1*} *C1 ELSE* {*P2*} *C2* {*Q*}) = *C2*

fun $anno_1$ :: *'a acom* ⇒ *'a* **where**
$anno_1$(*IF b THEN* {*P1*} *C1 ELSE* {*P2*} *C2* {*Q*}) = *P1* |
$anno_1$({*I*} *WHILE b DO* {*P*} *C* {*Q*}) = *I*

fun $anno_2$ :: *'a acom* ⇒ *'a* **where**
$anno_2$(*IF b THEN* {*P1*} *C1 ELSE* {*P2*} *C2* {*Q*}) = *P2* |
$anno_2$({*I*} *WHILE b DO* {*P*} *C* {*Q*}) = *P*

fun *Union_acom* :: *com* ⇒ *'a acom set* ⇒ *'a set acom* **where**
Union_acom com.SKIP M = (*SKIP* {*post* ‘ *M*}) |
Union_acom (*x* ::= *a*) *M* = (*x* ::= *a* {*post* ‘ *M*}) |
Union_acom (*c1*;*c2*) *M* =

Union_acom c1 (*sub*$_1$ ' *M*); *Union_acom c2* (*sub*$_2$ ' *M*) |
Union_acom (*IF b THEN c1 ELSE c2*) *M* =
IF b THEN {*anno*$_1$ ' *M*} *Union_acom c1* (*sub*$_1$ ' *M*) *ELSE* {*anno*$_2$ ' *M*}
Union_acom c2 (*sub*$_2$ ' *M*)
{*post* ' *M*} |
Union_acom (*WHILE b DO c*) *M* =
{*anno*$_1$ ' *M*}
WHILE b DO {*anno*$_2$ ' *M*} *Union_acom c* (*sub*$_1$ ' *M*)
{*post* ' *M*}

interpretation
Complete_Lattice {*C*. *strip C* = *c*} *map_acom Inter o* (*Union_acom c*) **for**
c
proof
case *goal1*
have *a*:*A* ⟹ *map_acom Inter* (*Union_acom* (*strip a*) *A*) ≤ *a*
proof(*induction a arbitrary*: *A*)
case *Seq* **from** *Seq.prems* **show** *?case* **by**(*force intro*!: *Seq.IH*)
next
case *If* **from** *If.prems* **show** *?case* **by**(*force intro*!: *If.IH*)
next
case *While* **from** *While.prems* **show** *?case* **by**(*force intro*!: *While.IH*)
qed *force+*
with *goal1* **show** *?case* **by** *auto*
next
case *goal2*
thus *?case*
proof(*simp*, *induction b arbitrary*: *c A*)
case *SKIP* **thus** *?case* **by** (*force simp*:*SKIP_le*)
next
case *Assign* **thus** *?case* **by** (*force simp*:*Assign_le*)
next
case *Seq* **from** *Seq.prems* **show** *?case* **by**(*force intro*!: *Seq.IH simp*:*Seq_le*)
next
case *If* **from** *If.prems* **show** *?case* **by** (*force simp*: *If_le intro*!: *If.IH*)
next
case *While* **from** *While.prems* **show** *?case* **by**(*fastforce simp*: *While_le*
intro: *While.IH*)
qed
next
case *goal3*
have *strip*(*Union_acom c A*) = *c*
proof(*induction c arbitrary*: *A*)
case *Seq* **from** *Seq.prems* **show** *?case* **by** (*fastforce simp*: *strip_eq_Seq*

subset_iff intro!: *Seq.IH*)
  next
    case *If* **from** *If.prems* **show** *?case* **by** (*fastforce intro*!: *If.IH simp*: *strip_eq_If*)
  next
    case *While* **from** *While.prems* **show** *?case* **by**(*fastforce intro*: *While.IH simp*: *strip_eq_While*)
  qed *auto*
  thus *?case* **by** *auto*
qed

lemma *le_post*: $c \leq d \Longrightarrow post\ c \leq post\ d$
by(*induction c d rule*: *less_eq_acom.induct*) *auto*

12.2.2 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*(*x* := *aval e s*) |*s*. *s* : *S*}} |
step S (*C1*; *C2*) = *step S C1*; *step* (*post C1*) *C2* |
step S (*IF b THEN* {*P1*} *C1 ELSE* {*P2*} *C2* {*P*}) =
  IF b THEN {{*s*:*S*. *bval b s*}} *step P1 C1 ELSE* {{*s*:*S*. ¬ *bval b s*}} *step P2 C2*
  {*post C1* ∪ *post C2*} |
step S ({*I*} *WHILE b DO* {*P*} *C* {*P'*}) =
  {*S* ∪ *post C*} *WHILE b DO* {{*s*:*I*. *bval b s*}} *step P C* {{*s*:*I*. ¬ *bval b s*}}

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

lemma *mono2_step*: $c1 \leq c2 \Longrightarrow S1 \subseteq S2 \Longrightarrow step\ S1\ c1 \leq step\ S2\ c2$
proof(*induction c1 c2 arbitrary*: *S1 S2 rule*: *less_eq_acom.induct*)
  case *2* **thus** *?case* **by** *fastforce*
next
  case *3* **thus** *?case* **by**(*simp add*: *le_post*)
next
  case *4* **thus** *?case* **by**(*simp add*: *subset_iff*)(*metis le_post 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 c* (*step S*) = *step S* (*lfp c* (*step S*))
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*])

12.2.3 Relation to big-step semantics

lemma *post_Union_acom*: $\forall\ c' \in M.\ strip\ c' = c \Longrightarrow post\ (Union\_acom\ c\ M) = post\ `\ M$
proof(*induction c arbitrary*: *M*)
 case (*Seq c1 c2*)
 have $post\ `\ M = post\ `\ sub_2\ `\ M$ **using** *Seq.prems* **by** (*force simp*: *strip_eq_Seq*)
 moreover have $\forall\ c' \in sub_2\ `\ M.\ strip\ c' = c2$ **using** *Seq.prems* **by** (*auto simp*: *strip_eq_Seq*)
 ultimately show *?case* **using** *Seq.IH*(*2*)[*of* sub_2 ‘ *M*] **by** *simp*
qed *simp_all*

lemma *post_lfp*: $post(lfp\ c\ f) = (\bigcap\{post\ C|C.\ strip\ C = c \wedge f\ C \leq C\})$
by(*auto simp add*: *lfp_def post_Union_acom*)

lemma *big_step_post_step*:
 $[\![\ (c,\ s) \Rightarrow t;\ \ strip\ C = c;\ \ s \in S;\ \ step\ S\ C \leq C\ ]\!] \Longrightarrow t \in post\ C$
proof(*induction arbitrary*: *C S rule*: *big_step_induct*)
 case *Skip* **thus** *?case* **by**(*auto simp*: *strip_eq_SKIP*)
next
 case *Assign* **thus** *?case* **by**(*fastforce simp*: *strip_eq_Assign*)
next
 case *Seq* **thus** *?case* **by**(*fastforce simp*: *strip_eq_Seq*)
next
 case *IfTrue* **thus** *?case* **apply**(*auto simp*: *strip_eq_If*)
  by (*metis* (*lifting*) *mem_Collect_eq set_mp*)

next
  case *IfFalse* **thus** *?case* **apply**(*auto simp*: *strip_eq_If*)
    by (*metis* (*lifting*) *mem_Collect_eq set_mp*)
next
  case (*WhileTrue b s1 c′ s2 s3*)
  from *WhileTrue.prems*(*1*) **obtain** *I P C′ Q* **where** *C* = {*I*} *WHILE b DO* {*P*} *C′* {*Q*} *strip C′* = *c′*
    by(*auto simp*: *strip_eq_While*)
  from *WhileTrue.prems*(*3*) ‹*C* = _›
  have *step P C′* ≤ *C′* {*s* ∈ *I*. *bval b s*} ≤ *P* *S* ≤ *I* *step* (*post C′*) *C* ≤ *C* **by** *auto*
  have *step* {*s* ∈ *I*. *bval b s*} *C′* ≤ *C′*
    by (*rule order_trans*[*OF mono2_step*[*OF order_refl* ‹{*s* ∈ *I*. *bval b s*} ≤ *P*›] ‹*step P C′* ≤ *C′*›])
  have *s1*: {*s*:*I*. *bval b s*} **using** ‹*s1* ∈ *S*› ‹*S* ⊆ *I*› ‹*bval b s1*› **by** *auto*
  note *s2_in_post_C′* = *WhileTrue.IH*(*1*)[*OF* ‹*strip C′* = *c′*› *this* ‹*step* {*s* ∈ *I*. *bval b s*} *C′* ≤ *C′*›]
  from *WhileTrue.IH*(*2*)[*OF WhileTrue.prems*(*1*) *s2_in_post_C′* ‹*step* (*post C′*) *C* ≤ *C*›]
  show *?case* **.**
next
  case (*WhileFalse b s1 c′*) **thus** *?case* **by** (*force simp*: *strip_eq_While*)
qed

lemma *big_step_lfp*: ⟦ (*c*,*s*) ⇒ *t*; *s* ∈ *S* ⟧ ⟹ *t* ∈ *post*(*lfp c* (*step S*))
by(*auto simp add*: *post_lfp intro*: *big_step_post_step*)

lemma *big_step_CS*: (*c*,*s*) ⇒ *t* ⟹ *t* : *post*(*CS c*)
by(*simp add*: *CS_def big_step_lfp*)

end

theory *Abs_Int_Tests*
imports *Com*
begin

12.3 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_Int_init*
imports ~~/src/HOL/ex/Interpretation_with_Defs
~~/src/HOL/Library/While_Combinator
Vars Collecting Abs_Int_Tests
begin

hide_const (**open**) *top bot dom* — to avoid qualified names

end

theory *Abs_Int0*
imports *Abs_Int_init*
begin

12.4 Orderings

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

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

declare *le_trans*[*trans*]

end

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

class *join* = *preord* +
fixes *join* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $\sqcup$ *65*)

class *semilattice* = *join* +
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*, *preord*) *preord*
begin

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

instance
proof
  case *goal2* **thus** *?case* **by** (*metis le_fun_def preord_class.le_trans*)
qed (*simp_all add*: *le_fun_def*)

end

instantiation *fun* :: (*type*, *semilattice*) *semilattice*
begin

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
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*) (*D1*;*D2*) = (*C1* ⊑ *D1* ∧ *C2* ⊑ *D2*) |
le_acom (*IF b THEN* {*p1*} *C1 ELSE* {*p2*} *C2* {*S*}) (*IF b' THEN* {*q1*} *D1 ELSE* {*q2*} *D2* {*S'*}) =
(*b*=*b'* ∧ *p1* ⊑ *q1* ∧ *C1* ⊑ *D1* ∧ *p2* ⊑ *q2* ∧ *C2* ⊑ *D2* ∧ *S* ⊑ *S'*) |
le_acom ({*I*} *WHILE b DO* {*p*} *C* {*P*}) ({*I'*} *WHILE b' DO* {*p'*} *C'* {*P'*}) =
(*b*=*b'* ∧ *p* ⊑ *p'* ∧ *C* ⊑ *C'* ∧ *I* ⊑ *I'* ∧ *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* ⟷ (∃ *D1 D2*. *C* = *D1*;*D2* ∧ *C1* ⊑ *D1* ∧ *C2* ⊑ *D2*)
by (*cases C*) *auto*

lemma [*simp*]: *IF b THEN* {*p1*} *C1 ELSE* {*p2*} *C2* {*S*} ⊑ *C* ⟷
(∃ *q1 q2 D1 D2 S'*. *C* = *IF b THEN* {*q1*} *D1 ELSE* {*q2*} *D2* {*S'*} ∧
p1 ⊑ *q1* ∧ *C1* ⊑ *D1* ∧ *p2* ⊑ *q2* ∧ *C2* ⊑ *D2* ∧ *S* ⊑ *S'*)
by (*cases C*) *auto*

lemma [*simp*]: {*I*} *WHILE b DO* {*p*} *C* {*P*} ⊑ *W* ⟷
(∃ *I' p' C' P'*. *W* = {*I'*} *WHILE b DO* {*p'*} *C'* {*P'*} ∧ *p* ⊑ *p'* ∧ *C* ⊑ *C'* ∧ *I* ⊑ *I'* ∧ *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

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* :: (*join*)*join*
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*

instance ..

end

instantiation *option* :: (*semilattice*)*semilattice*
begin

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

class *bot* = *preord* +
fixes *bot* :: *'a* (⊥)
assumes *bot*[*simp*]: ⊥ ⊑ *x*

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

definition *bot_option* :: *'a option* **where**
⊥ = *None*

instance
proof
 case *goal1* **thus** *?case* **by**(*auto simp*: *bot_option_def*)
qed

end

definition *bot* :: *com* ⇒ *'a option acom* **where**
bot c = *anno None c*

lemma *bot_least*: *strip C* = *c* ⟹ *bot c* ⊑ *C*
by(*induct C arbitrary*: *c*)(*auto simp*: *bot_def*)

lemma *strip_bot*[*simp*]: *strip*(*bot c*) = *c*

by(*simp add*: *bot_def*)

12.4.1 Post-fixed point iteration

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

lemma *pfp_pfp*: **assumes** $pfp\ f\ x0 = Some\ x$ **shows** $f\ x \sqsubseteq x$
using *while_option_stop*[*OF assms*[*simplified pfp_def*]] **by** *simp*

lemma *while_least*:
assumes $\forall x{\in}L.\forall y{\in}L.\ x \sqsubseteq y \longrightarrow f\ x \sqsubseteq f\ y$ **and** $\forall x.\ x \in L \longrightarrow f\ x \in L$
and $\forall x \in L.\ b \sqsubseteq x$ **and** $b \in L$ **and** $f\ q \sqsubseteq q$ **and** $q \in L$
and $while\_option\ P\ f\ b = Some\ p$
shows $p \sqsubseteq q$
using *while_option_rule*[*OF* _ *assms*(*7*)[*unfolded pfp_def*],
where $P = \%x.\ x \in L \wedge x \sqsubseteq q$]
by (*metis assms*(*1−6*) *le_trans*)

lemma *pfp_inv*:
$pfp\ f\ x = Some\ y \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow P(f\ x)) \Longrightarrow P\ x \Longrightarrow P\ y$
unfolding *pfp_def* **by** (*metis* (*lifting*) *while_option_rule*)

lemma *strip_pfp*:
assumes $\bigwedge x.\ g(f\ x) = g\ x$ **and** $pfp\ f\ x0 = Some\ x$ **shows** $g\ x = g\ x0$
using *pfp_inv*[*OF assms*(*2*), **where** $P = \%x.\ g\ x = g\ x0$] *assms*(*1*) **by** *simp*

12.5 Abstract Interpretation

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

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

The interface for abstract values:

locale *Val_abs* =
fixes γ :: $'av::semilattice \Rightarrow val\ set$
assumes *mono_gamma*: $a \sqsubseteq b \Longrightarrow \gamma\ a \subseteq \gamma\ b$
and *gamma_Top*[*simp*]: $\gamma\ \top = UNIV$
fixes num' :: $val \Rightarrow 'av$
and $plus'$ :: $'av \Rightarrow 'av \Rightarrow 'av$
assumes *gamma_num'*: $i \in \gamma(num'\ i)$

and *gamma_plus'*: $i1 \in \gamma\ a1 \Longrightarrow i2 \in \gamma\ a2 \Longrightarrow i1+i2 \in \gamma(plus'\ a1\ a2)$

type_synonym $'av\ st = (vname \Rightarrow 'av)$

locale *Abs_Int_Fun* = *Val_abs* γ **for** γ :: *'av::semilattice* $\Rightarrow$ *val set*
begin

fun *aval'* :: *aexp* $\Rightarrow$ *'av st* $\Rightarrow$ *'av* **where**
aval' (*N i*) *S* = *num' i* |
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* | *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* {*P1*} *C1 ELSE* {*P2*} *C2* {*Q*}) =
IF b THEN {*S*} *step' P1 C1 ELSE* {*S*} *step' P2 C2*
{*post C1* $\sqcup$ *post C2*} |
step' S ({*I*} *WHILE b DO* {*P*} *C* {*Q*}) =
{*S* $\sqcup$ *post C*} *WHILE b DO* {*I*} *step' P C* {*I*}

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

lemma *strip_step'*[*simp*]: *strip*(*step' S C*) = *strip C*
by(*induct C arbitrary*: *S*) (*simp_all add*: *Let_def*)

abbreviation γ_s :: *'av st* $\Rightarrow$ *state set*
where γ_s == γ*_fun* γ

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

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

lemma *gamma_s_Top*[*simp*]: γ_s *Top* = *UNIV*
by(*simp add*: *Top_fun_def* γ*_fun_def*)

lemma *gamma_o_Top*[*simp*]: γ_o *Top* = *UNIV*
by (*simp add*: *Top_option_def*)

lemma *mono_gamma_s*: *f1* $\sqsubseteq$ *f2* $\Longrightarrow \gamma_s$ *f1* $\subseteq \gamma_s$ *f2*
by(*auto simp*: *le_fun_def* γ*_fun_def dest*: *mono_gamma*)

lemma *mono_gamma_o*:
  S1 $\sqsubseteq$ *S2* $\Longrightarrow \gamma_o$ *S1* $\subseteq \gamma_o$ *S2*
by(*induction S1 S2 rule*: *le_option.induct*)(*simp_all add*: *mono_gamma_s*)

lemma *mono_gamma_c*: *C1* $\sqsubseteq$ *C2* $\Longrightarrow \gamma_c$ *C1* $\le \gamma_c$ *C2*
by (*induction C1 C2 rule*: *le_acom.induct*) (*simp_all add*:*mono_gamma_o*)

Soundness:

lemma *aval'_sound*: *s* : γ_s *S* $\Longrightarrow$ *aval a s* : γ(*aval' a S*)
by (*induct a*) (*auto simp*: *gamma_num' gamma_plus'* γ*_fun_def*)

lemma *in_gamma_update*:
  $\llbracket$ *s* : γ_s *S*; *i* : γ *a* $\rrbracket \Longrightarrow$ *s*(*x* := *i*) : γ_s(*S*(*x* := *a*))
by(*simp add*: γ*_fun_def*)

lemma *step_step'*: *step* (γ_o *S*) (γ_c *C*) $\le \gamma_c$ (*step' S C*)
proof(*induction C arbitrary*: *S*)
  case *SKIP* **thus** *?case* **by** *auto*
next
  case *Assign* **thus** *?case*
    by (*fastforce intro*: *aval'_sound in_gamma_update split*: *option.splits*)
next
  case *Seq* **thus** *?case* **by** *auto*
next
  case *If* **thus** *?case* **by** (*auto simp*: *mono_gamma_o*)
next
  case *While* **thus** *?case* **by** (*auto simp*: *mono_gamma_o*)
qed

lemma *AI_sound*: *AI c* = *Some C* $\Longrightarrow$ *CS c* $\le \gamma_c$ *C*
proof(*simp add*: *CS_def AI_def*)
  assume *1*: *pfp* (*step'* $\top$) (*bot c*) = *Some C*
  have *pfp'*: *step'* $\top$ *C* $\sqsubseteq$ *C* **by**(*rule pfp_pfp*[*OF 1*])
  have *2*: *step* (γ_o $\top$) (γ_c *C*) $\le \gamma_c$ *C* — transfer the pfp'
  proof(*rule order_trans*)
    show *step* (γ_o $\top$) (γ_c *C*) $\le \gamma_c$ (*step'* $\top$ *C*) **by**(*rule step_step'*)
    show ... $\le \gamma_c$ *C* **by** (*metis mono_gamma_c*[*OF pfp'*])

qed
have *3*: *strip* (γ_c *C*) = *c* **by**(*simp add*: *strip_pfp*[*OF _ 1*])
have *lfp c* (*step* (γ_o ⊤)) ≤ γ_c *C*
by(*rule lfp_lowerbound*[*simplified*,**where** *f*=*step* (γ_o ⊤), *OF 3 2*])
thus *lfp c* (*step UNIV*) ≤ γ_c *C* **by** *simp*
qed

end

12.5.1 Monotonicity

lemma *mono_post*: *C1* ⊑ *C2* ⟹ *post C1* ⊑ *post C2*
by(*induction C1 C2 rule*: *le_acom.induct*) (*auto*)

locale *Abs_Int_Fun_mono* = *Abs_Int_Fun* +
assumes *mono_plus'*: *a1* ⊑ *b1* ⟹ *a2* ⊑ *b2* ⟹ *plus' a1 a2* ⊑ *plus' b1 b2*
begin

lemma *mono_aval'*: *S* ⊑ *S'* ⟹ *aval' e S* ⊑ *aval' e S'*
by(*induction e*)(*auto simp*: *le_fun_def mono_plus'*)

lemma *mono_update*: *a* ⊑ *a'* ⟹ *S* ⊑ *S'* ⟹ *S*(*x* := *a*) ⊑ *S'*(*x* := *a'*)
by(*simp add*: *le_fun_def*)

lemma *mono_step'*: *S1* ⊑ *S2* ⟹ *C1* ⊑ *C2* ⟹ *step' S1 C1* ⊑ *step' S2 C2*
apply(*induction C1 C2 arbitrary*: *S1 S2 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*
begin

12.5.2 Set-based lattices

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*)

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

instantiation *acom* :: (*L*)*L*
begin

definition *L_acom* **where**
L X = {*C*. *vars*(*strip C*) ⊆ *X* ∧ (∀ *a* ∈ *set*(*annos C*). *a* ∈ *L X*)}

instance ..

end

instantiation *option* :: (*L*)*L*
begin

definition *L_option* **where**
$L\ X = \{opt.\ case\ opt\ of\ None \Rightarrow True \mid Some\ x \Rightarrow x \in L\ X\}$

lemma *L_option_simps*[*simp*]: $None \in L\ X\ (Some\ x \in L\ X) = (x \in L\ X)$
by(*simp_all add*: *L_option_def*)

instance ..

end

class *semilatticeL* = *join* + *L* +
fixes *top* :: *vname set* ⇒ *'a*
assumes *join_ge1* [*simp*]: $x \in L\ X \Longrightarrow y \in L\ X \Longrightarrow x \sqsubseteq x \sqcup y$
and *join_ge2* [*simp*]: $x \in L\ X \Longrightarrow y \in L\ X \Longrightarrow y \sqsubseteq x \sqcup y$
and *join_least*[*simp*]: $x \sqsubseteq z \Longrightarrow y \sqsubseteq z \Longrightarrow x \sqcup y \sqsubseteq z$
and *top*[*simp*]: $x \in L\ X \Longrightarrow x \sqsubseteq top\ X$
and *top_in_L*[*simp*]: $top\ X \in L\ X$
and *join_in_L*[*simp*]: $x \in L\ X \Longrightarrow y \in L\ X \Longrightarrow x \sqcup y \in L\ X$

notation (*input*) *top* ($\top_{\_}$)
notation (*latex* **output**) *top* ($\top_{\_}$)

instantiation *option* :: (*semilatticeL*)*semilatticeL*
begin

definition *top_option* **where** *top c* = *Some*(*top c*)

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*)
next
  case *goal5* **thus** *?case* **by**(*simp add*: *top_option_def*)
next
  case *goal6* **thus** *?case* **by**(*simp add*: *L_option_def split*: *option.splits*)
qed

end

12.6 Abstract State with Computable Ordering

hide_type *st* — to avoid long names

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

datatype $'a\ st = FunDom\ vname \Rightarrow\ 'a\ vname\ set$

fun *fun* **where** $fun\ (FunDom\ f\ X) = f$
fun *dom* **where** $dom\ (FunDom\ f\ X) = X$

definition $show\_st\ S = (\lambda x.\ (x,\ fun\ S\ x))\ `\ dom\ S$

value $[code]\ show\_st\ (FunDom\ (\lambda x.\ 1::int)\ \{''a'',''b''\})$

definition $show\_acom = map\_acom\ (Option.map\ show\_st)$
definition $show\_acom\_opt = Option.map\ show\_acom$

definition $update\ F\ x\ y = FunDom\ ((fun\ F)(x{:=}y))\ (dom\ F)$

lemma $fun\_update[simp]{:}\ fun\ (update\ S\ x\ y) = (fun\ S)(x{:=}y)$
by(*rule ext*)(*auto simp*: *update_def*)

lemma $dom\_update[simp]{:}\ dom\ (update\ S\ x\ y) = dom\ S$
by(*simp add*: *update_def*)

definition $\gamma\_st\ \gamma\ F = \{f.\ \forall x{\in}dom\ F.\ f\ x \in \gamma(fun\ F\ x)\}$

instantiation *st* :: (*preord*) *preord*
begin

definition $le\_st :: 'a\ st \Rightarrow 'a\ st \Rightarrow bool$ **where**
$F \sqsubseteq G = (dom\ F = dom\ G \wedge (\forall x \in dom\ F.\ fun\ F\ x \sqsubseteq fun\ G\ x))$

instance
proof
 case *goal2* **thus** *?case* **by**(*auto simp*: *le_st_def*)(*metis preord_class.le_trans*)
qed (*auto simp*: *le_st_def*)

end

instantiation *st* :: (*join*) *join*
begin

definition *join_st* :: *'a st* ⇒ *'a st* ⇒ *'a st* **where**
$F \sqcup G = FunDom\ (\lambda x.\ fun\ F\ x \sqcup fun\ G\ x)\ (dom\ F)$

instance ..

end

instantiation *st* :: (*type*) *L*
begin

definition *L_st* :: *vname set* ⇒ *'a st set* **where**
$L\ X = \{F.\ dom\ F = X\}$

instance ..

end

instantiation *st* :: (*semilattice*) *semilatticeL*
begin

definition *top_st* **where** $top\ X = FunDom\ (\lambda x.\ \top)\ X$

instance
proof
qed (*auto simp*: *le_st_def join_st_def top_st_def L_st_def*)

end

Trick to make code generator happy.

lemma [*code*]: $L = (L :: \_ \Rightarrow \_\ st\ set)$
by(*rule refl*)

lemma *mono_fun*: $F \sqsubseteq G \Longrightarrow x : dom\ F \Longrightarrow fun\ F\ x \sqsubseteq fun\ G\ x$
by(*auto simp*: *le_st_def*)

lemma *mono_update*[*simp*]:
$a1 \sqsubseteq a2 \Longrightarrow S1 \sqsubseteq S2 \Longrightarrow update\ S1\ x\ a1 \sqsubseteq update\ S2\ x\ a2$
by(*auto simp add*: *le_st_def update_def*)

locale *Gamma* = *Val_abs* **where** $\gamma=\gamma$ **for** γ :: *'av::semilattice* ⇒ *val set*
begin

abbreviation γ_s :: *'av st* $\Rightarrow$ *state set*
where γ_s == γ*_st* γ

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

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

lemma *gamma_s_Top*[*simp*]: γ_s (*top c*) = *UNIV*
by(*auto simp*: *top_st_def* γ*_st_def*)

lemma *gamma_o_Top*[*simp*]: γ_o (*top c*) = *UNIV*
by (*simp add*: *top_option_def*)

lemma *mono_gamma_s*: $f \sqsubseteq g \Longrightarrow \gamma_s\ f \subseteq \gamma_s\ g$
apply(*simp add*:γ*_st_def subset_iff le_st_def split*: *if_splits*)
by (*metis mono_gamma subsetD*)

lemma *mono_gamma_o*:
$S1 \sqsubseteq S2 \Longrightarrow \gamma_o\ S1 \subseteq \gamma_o\ S2$
by(*induction S1 S2 rule*: *le_option.induct*)(*simp_all add*: *mono_gamma_s*)

lemma *mono_gamma_c*: $C1 \sqsubseteq C2 \Longrightarrow \gamma_c\ C1 \le \gamma_c\ C2$
by (*induction C1 C2 rule*: *le_acom.induct*) (*simp_all add*:*mono_gamma_o*)

lemma *in_gamma_option_iff*:
$x : \gamma\_option\ r\ u \longleftrightarrow (\exists\, u'.\ u = Some\ u' \wedge x : r\ u')$
by (*cases u*) *auto*

end

end

theory *Abs_Int1*
imports *Abs_State*
begin

lemma *le_iff_le_annos_zip*: $C1 \sqsubseteq C2 \longleftrightarrow$
$(\forall\ (a1,a2) \in set(zip\ (annos\ C1)\ (annos\ C2)).\ a1 \sqsubseteq a2) \wedge strip\ C1 = strip\ C2$

by(*induct C1 C2 rule*: *le_acom.induct*) (*auto simp*: *size_annos_same2*)

lemma *le_iff_le_annos*: *C1* ⊑ *C2* ⟷
strip C1 = *strip C2* ∧ (∀ *i*<*size*(*annos C1*). *annos C1* ! *i* ⊑ *annos C2* ! *i*)
by(*auto simp add*: *le_iff_le_annos_zip set_zip*) (*metis size_annos_same2*)

lemma *mono_fun_L*[*simp*]: *F* ∈ *L X* ⟹ *F* ⊑ *G* ⟹ *x* : *X* ⟹ *fun F x* ⊑ *fun G x*
by(*simp add*: *mono_fun L_st_def*)

lemma *bot_in_L*[*simp*]: *bot c* ∈ *L*(*vars c*)
by(*simp add*: *L_acom_def bot_def*)

lemma *L_acom_simps*[*simp*]: *SKIP* {*P*} ∈ *L X* ⟷ *P* ∈ *L X*
(*x* ::= *e* {*P*}) ∈ *L X* ⟷ *x* : *X* ∧ *vars e* ⊆ *X* ∧ *P* ∈ *L X*
(*C1*;*C2*) ∈ *L X* ⟷ *C1* ∈ *L X* ∧ *C2* ∈ *L X*
(*IF b THEN* {*P1*} *C1 ELSE* {*P2*} *C2* {*Q*}) ∈ *L X* ⟷
vars b ⊆ *X* ∧ *C1* ∈ *L X* ∧ *C2* ∈ *L X* ∧ *P1* ∈ *L X* ∧ *P2* ∈ *L X* ∧ *Q* ∈ *L X*
({*I*} *WHILE b DO* {*P*} *C* {*Q*}) ∈ *L X* ⟷
I ∈ *L X* ∧ *vars b* ⊆ *X* ∧ *P* ∈ *L X* ∧ *C* ∈ *L X* ∧ *Q* ∈ *L X*
by(*auto simp add*: *L_acom_def*)

lemma *post_in_annos*: *post C* : *set*(*annos C*)
by(*induction C*) *auto*

lemma *post_in_L*[*simp*]: *C* ∈ *L X* ⟹ *post C* ∈ *L X*
by(*simp add*: *L_acom_def post_in_annos*)

12.7 Computable Abstract Interpretation

Abstract interpretation over type *st* instead of functions.

context *Gamma*
begin

fun *aval′* :: *aexp* ⇒ *′av st* ⇒ *′av* **where**
aval′ (*N i*) *S* = *num′ i* |
aval′ (*V x*) *S* = *fun S x* |
aval′ (*Plus a1 a2*) *S* = *plus′* (*aval′ a1 S*) (*aval′ a2 S*)

lemma *aval′_sound*: *s* : γ_s *S* ⟹ *vars a* ⊆ *dom S* ⟹ *aval a s* : γ(*aval′ a S*)

by (*induction a*) (*auto simp*: *gamma_num′ gamma_plus′ γ_st_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*::*semilattice* ⇒ *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* {*P1*} *C1 ELSE* {*P2*} *C2* {*Q*}) =
 (*IF b THEN* {*S*} *step′ P1 C1 ELSE* {*S*} *step′ P2 C2* {*post C1* ⊔ *post C2*}) |
step′ S ({*I*} *WHILE b DO* {*P*} *C* {*Q*}) =
 {*S* ⊔ *post C*} *WHILE b DO* {*I*} *step′ P C* {*I*}

definition *AI* :: *com* ⇒ *′av st option acom option* **where**
AI c = *pfp* (*step′* (*top*(*vars c*))) (*bot c*)

lemma *strip_step′*[*simp*]: *strip*(*step′ S C*) = *strip C*
by(*induct C arbitrary*: *S*) (*simp_all add*: *Let_def*)

Soundness:

lemma *in_gamma_update*:
 ⟦ $s : \gamma_s\ S$; $i : \gamma\ a$ ⟧ ⟹ $s(x := i) : \gamma_s(update\ S\ x\ a)$
by(*simp add*: *γ_st_def*)

lemma *step_step′*: $C \in L\ X \Longrightarrow S \in L\ X \Longrightarrow step\ (\gamma_o\ S)\ (\gamma_c\ C) \le \gamma_c\ (step'\ S\ C)$
proof(*induction C arbitrary*: *S*)
 case *SKIP* **thus** *?case* **by** *auto*
next
 case *Assign* **thus** *?case*
  by (*fastforce simp*: *L_st_def intro*: *aval′_sound in_gamma_update split*: *option.splits*)
next
 case *Seq* **thus** *?case* **by** *auto*
next

case (*If b p1 C1 p2 C2 P*)
hence *post C1* ⊑ *post C1* ⊔ *post C2* ∧ *post C2* ⊑ *post C1* ⊔ *post C2*
by(*simp*, *metis post_in_L join_ge1 join_ge2*)
thus *?case* **using** *If* **by** (*auto simp*: *mono_gamma_o*)
next
case *While* **thus** *?case* **by** (*auto simp*: *mono_gamma_o*)
qed

lemma *step′_in_L*[*simp*]:
⟦ *C* ∈ *L X*; *S* ∈ *L X* ⟧ ⟹ (*step′ S C*) ∈ *L X*
proof(*induction C arbitrary*: *S*)
case *Assign* **thus** *?case*
by(*auto simp*: *L_st_def update_def split*: *option.splits*)
qed *auto*

lemma *AI_sound*: *AI c* = *Some C* ⟹ *CS c* ≤ γ_c *C*
proof(*simp add*: *CS_def AI_def*)
assume *1*: *pfp* (*step′* (*top*(*vars c*))) (*bot c*) = *Some C*
have *C* ∈ *L*(*vars c*)
by(*rule pfp_inv*[**where** *P* = %*C*. *C* ∈ *L*(*vars c*), *OF 1 _ bot_in_L*])
(*erule step′_in_L*[*OF _ top_in_L*])
have *pfp′*: *step′* (*top*(*vars c*)) *C* ⊑ *C* **by**(*rule pfp_pfp*[*OF 1*])
have *2*: *step* (γ_o(*top*(*vars c*))) (γ_c *C*) ≤ γ_c *C*
proof(*rule order_trans*)
show *step* (γ_o (*top*(*vars c*))) (γ_c *C*) ≤ γ_c (*step′* (*top*(*vars c*)) *C*)
by(*rule step_step′*[*OF* ‹*C* ∈ *L*(*vars c*)› *top_in_L*])
show γ_c (*step′* (*top*(*vars c*)) *C*) ≤ γ_c *C*
by(*rule mono_gamma_c*[*OF pfp′*])
qed
have *3*: *strip* (γ_c *C*) = *c* **by**(*simp add*: *strip_pfp*[*OF _ 1*])
have *lfp c* (*step* (γ_o(*top*(*vars c*)))) ≤ γ_c *C*
by(*rule lfp_lowerbound*[*simplified*,**where** *f*=*step* (γ_o(*top*(*vars c*))), *OF*
3 2])
thus *lfp c* (*step UNIV*) ≤ γ_c *C* **by** *simp*
qed

end

12.7.1 Monotonicity

lemma *le_join_disj*: *y* ∈ *L X* ⟹ (*z*::_::*semilatticeL*) ∈ *L X* ⟹
x ⊑ *y* ∨ *x* ⊑ *z* ⟹ *x* ⊑ *y* ⊔ *z*
by (*metis join_ge1 join_ge2 preord_class.le_trans*)

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'*:
$S1 \sqsubseteq S2 \Longrightarrow S1 \in L\ X \Longrightarrow S2 \in L\ X \Longrightarrow vars\ e \subseteq X \Longrightarrow aval'\ e\ S1 \sqsubseteq aval'\ e\ S2$
by(*induction e*) (*auto simp*: *le_st_def mono_plus' L_st_def*)

theorem *mono_step'*: $S1 \in L\ X \Longrightarrow S2 \in L\ X \Longrightarrow C1 \in L\ X \Longrightarrow C2 \in L\ X \Longrightarrow$
$S1 \sqsubseteq S2 \Longrightarrow C1 \sqsubseteq C2 \Longrightarrow step'\ S1\ C1 \sqsubseteq step'\ S2\ C2$
apply(*induction C1 C2 arbitrary*: *S1 S2 rule*: *le_acom.induct*)
apply (*auto simp*: *Let_def mono_aval' mono_post*
le_join_disj le_join_disj[*OF post_in_L post_in_L*]
split: *option.split*)
done

lemma *mono_step'_top*: $C \in L\ X \Longrightarrow C' \in L\ X \Longrightarrow$
$C \sqsubseteq C' \Longrightarrow step'\ (top\ X)\ C \sqsubseteq step'\ (top\ X)\ C'$
by (*metis top_in_L mono_step' preord_class.le_refl*)

lemma *pfp_bot_least*:
assumes $\forall x{\in}L(vars\ c){\cap}\{C.\ strip\ C = c\}.\forall y{\in}L(vars\ c){\cap}\{C.\ strip\ C = c\}.$
$x \sqsubseteq y \longrightarrow f\,x \sqsubseteq f\,y$
and $\forall C.\ C \in L(vars\ c){\cap}\{C.\ strip\ C = c\} \longrightarrow f\ C \in L(vars\ c){\cap}\{C.\ strip\ C = c\}$
and $f\ C' \sqsubseteq C'$ $strip\ C' = c$ $C' \in L(vars\ c)$ $pfp\ f\ (bot\ c) = Some\ C$
shows $C \sqsubseteq C'$
apply(*rule while_least*[*OF assms*(*1*,*2*) _ _ *assms*(*3*) _ *assms*(*6*)[*unfolded pfp_def*]])
by (*simp_all add*: *assms*(*4*,*5*) *bot_least*)

lemma *AI_least_pfp*: **assumes** $AI\ c = Some\ C$
and $step'\ (top(vars\ c))\ C' \sqsubseteq C'$ $strip\ C' = c$ $C' \in L(vars\ c)$
shows $C \sqsubseteq C'$
apply(*rule pfp_bot_least*[*OF* _ _ *assms*(*2*−*4*) *assms*(*1*)[*unfolded AI_def*]])
by (*simp_all add*: *mono_step'_top*)

end

12.7.2 Termination

abbreviation *sqless* (**infix** $\sqsubset$ *50*) **where**
$x \sqsubset y == x \sqsubseteq y \wedge \neg\, y \sqsubseteq x$

lemma *pfp_termination*:
fixes *x0* :: *'a::preord* **and** *m* :: *'a* ⇒ *nat*
assumes *mono*: $\bigwedge x\ y.\ I\ x \Longrightarrow I\ y \Longrightarrow x \sqsubseteq y \Longrightarrow f\ x \sqsubseteq f\ y$
and *m*: $\bigwedge x\ y.\ I\ x \Longrightarrow I\ y \Longrightarrow x \sqsubset y \Longrightarrow m\ x > m\ y$
and *I*: $\bigwedge x\ y.\ I\ x \Longrightarrow I(f\ x)$ **and** *I x0* **and** $x0 \sqsubseteq f\ x0$
shows $\exists x.\ pfp\ f\ x0 = Some\ x$
proof(*simp add*: *pfp_def*, *rule wf_while_option_Some*[**where** *P* = %*x*. *I x* & $x \sqsubseteq f\ x$])
  show *wf* $\{(y,x).\ ((I\ x \wedge x \sqsubseteq f\ x) \wedge \neg\, f\ x \sqsubseteq x) \wedge y = f\ x\}$
    by(*rule wf_subset*[*OF wf_measure*[*of m*]]) (*auto simp*: *m I*)
next
  show $I\ x0 \wedge x0 \sqsubseteq f\ x0$ **using** ‹*I x0*› ‹$x0 \sqsubseteq f\ x0$› **by** *blast*
next
  fix *x* **assume** $I\ x \wedge x \sqsubseteq f\ x$ **thus** $I(f\ x) \wedge f\ x \sqsubseteq f(f\ x)$
    by (*blast intro*: *I mono*)
qed

locale *Measure1* =
fixes *m* :: *'av::preord* ⇒ *nat*
fixes *h* :: *nat*
assumes *m1*: $x \sqsubseteq y \Longrightarrow m\ x \ge m\ y$
assumes *h*: $m\ x \le h$
begin

definition *m_s* :: *'av st* ⇒ *nat* (m_s) **where**
$m\_s\ S = (\sum\ x \in dom\ S.\ m(fun\ S\ x))$

lemma *m_s_h*: $x \in L\ X \Longrightarrow finite\ X \Longrightarrow m\_s\ x \le h * card\ X$
by(*simp add*: *L_st_def m_s_def*)
  (*metis nat_mult_commute of_nat_id setsum_bounded*[*OF h*])

lemma *m_s1*: $S1 \sqsubseteq S2 \Longrightarrow m\_s\ S1 \ge m\_s\ S2$
proof(*auto simp add*: *le_st_def m_s_def*)
  assume $\forall x{\in}dom\ S2.\ fun\ S1\ x \sqsubseteq fun\ S2\ x$
  hence $\forall x{\in}dom\ S2.\ m(fun\ S1\ x) \ge m(fun\ S2\ x)$ **by** (*metis m1*)
  thus $(\sum x{\in}dom\ S2.\ m\ (fun\ S2\ x)) \le (\sum x{\in}dom\ S2.\ m\ (fun\ S1\ x))$
    by (*metis setsum_mono*)
qed

definition *m_o* :: *nat* ⇒ *'av st option* ⇒ *nat* (m_o) **where**
m_o d opt = (*case opt of None* ⇒ *h*∗*d*+*1* | *Some S* ⇒ *m_s S*)

lemma *m_o_h*: *ost* ∈ *L X* ⟹ *finite X* ⟹ *m_o* (*card X*) *ost* ≤ (*h*∗*card X* + *1*)
by(*auto simp add*: *m_o_def m_s_h split*: *option.split dest*!:*m_s_h*)

lemma *m_o1*: *finite X* ⟹ *o1* ∈ *L X* ⟹ *o2* ∈ *L X* ⟹
  o1 ⊑ *o2* ⟹ *m_o* (*card X*) *o1* ≥ *m_o* (*card X*) *o2*
proof(*induction o1 o2 rule*: *le_option.induct*)
  case *1* **thus** *?case* **by** (*simp add*: *m_o_def*)(*metis m_s1*)
next
  case *2* **thus** *?case*
    by(*simp add*: *L_option_def m_o_def le_SucI m_s_h split*: *option.splits*)
next
  case *3* **thus** *?case* **by** *simp*
qed

definition *m_c* :: *'av st option acom* ⇒ *nat* (m_c) **where**
m_c C = ($\sum$ *i*<*size*(*annos C*). *m_o* (*card*(*vars*(*strip C*))) (*annos C* ! *i*))

lemma *m_c_h*: **assumes** *C* ∈ *L*(*vars*(*strip C*))
shows *m_c C* ≤ *size*(*annos C*) ∗ (*h* ∗ *card*(*vars*(*strip C*)) + *1*)
proof−
  let *?X* = *vars*(*strip C*) **let** *?n* = *card ?X* **let** *?a* = *size*(*annos C*)
  { fix *i* **assume** *i* < *?a*
    hence *annos C* ! *i* ∈ *L ?X* **using** *assms* **by**(*simp add*: *L_acom_def*)
    note *m_o_h*[*OF this finite_cvars*]
  } note *1* = *this*
  have *m_c C* = ($\sum$ *i*<*?a*. *m_o ?n* (*annos C* ! *i*)) **by**(*simp add*: *m_c_def*)
  also have . . . ≤ ($\sum$ *i*<*?a*. *h* ∗ *?n* + *1*)
    apply(*rule setsum_mono*) **using** *1* **by** *simp*
  also have . . . = *?a* ∗ (*h* ∗ *?n* + *1*) **by** *simp*
  finally show *?thesis* **.**
qed

end

locale *Measure* = *Measure1* +
assumes *m2*: *x* ⊏ *y* ⟹ *m x* > *m y*
begin

lemma *m_s2*: *finite*(*dom S1*) ⟹ *S1* ⊏ *S2* ⟹ *m_s S1* > *m_s S2*

proof(*auto simp add*: *le_st_def m_s_def*)
  assume *finite*(*dom S2*) **and** *0*: $\forall x \in dom\ S2.\ fun\ S1\ x \sqsubseteq fun\ S2\ x$
  hence *1*: $\forall x \in dom\ S2.\ m(fun\ S1\ x) \geq m(fun\ S2\ x)$ **by** (*metis m1*)
  fix *x* **assume** $x \in dom\ S2\ \neg\ fun\ S2\ x \sqsubseteq fun\ S1\ x$
  hence *2*: $\exists x \in dom\ S2.\ m(fun\ S1\ x) > m(fun\ S2\ x)$ **using** *0 m2* **by** *blast*
  from *setsum_strict_mono_ex1*[*OF* ‹*finite*(*dom S2*)› *1 2*]
  show $(\sum x \in dom\ S2.\ m\ (fun\ S2\ x)) < (\sum x \in dom\ S2.\ m\ (fun\ S1\ x))$ **.**
qed

lemma *m_o2*: *finite X* $\Longrightarrow$ $o1 \in L\ X \Longrightarrow o2 \in L\ X \Longrightarrow$
  $o1 \sqsubset o2 \Longrightarrow m\_o\ (card\ X)\ o1 > m\_o\ (card\ X)\ o2$
proof(*induction o1 o2 rule*: *le_option.induct*)
  case *1* **thus** *?case* **by** (*simp add*: *m_o_def L_st_def m_s2*)
next
  case *2* **thus** *?case*
    by(*auto simp add*: *m_o_def le_imp_less_Suc m_s_h*)
next
  case *3* **thus** *?case* **by** *simp*
qed

lemma *m_c2*: $C1 \in L(vars(strip\ C1)) \Longrightarrow C2 \in L(vars(strip\ C2)) \Longrightarrow$
  $C1 \sqsubset C2 \Longrightarrow m\_c\ C1 > m\_c\ C2$
proof(*auto simp add*: *le_iff_le_annos m_c_def size_annos_same*[*of C1 C2*] *L_acom_def*)
  let *?X* = *vars*(*strip C2*)
  let *?n* = *card ?X*
  assume *V1*: $\forall a \in set(annos\ C1).\ a \in L\ ?X$
    and *V2*: $\forall a \in set(annos\ C2).\ a \in L\ ?X$
    and *strip_eq*: *strip C1* = *strip C2*
    and *0*: $\forall i < size(annos\ C2).\ annos\ C1\ !\ i \sqsubseteq annos\ C2\ !\ i$
  hence *1*: $\forall i < size(annos\ C2).\ m\_o\ ?n\ (annos\ C1\ !\ i) \geq m\_o\ ?n\ (annos\ C2\ !\ i)$
    by (*auto simp*: *all_set_conv_all_nth*)
      (*metis finite_cvars m_o1 size_annos_same2*)
  fix *i* **assume** $i < size(annos\ C2)\ \neg\ annos\ C2\ !\ i \sqsubseteq annos\ C1\ !\ i$
  hence $m\_o\ ?n\ (annos\ C1\ !\ i) > m\_o\ ?n\ (annos\ C2\ !\ i)$ (**is** *?P i*)
    by(*metis m_o2*[*OF finite_cvars*] *V1 V2 nth_mem size_annos_same*[*OF strip_eq*] *0*)
  hence *2*: $\exists i < size(annos\ C2).\ ?P\ i$ **using** ‹$i < size(annos\ C2)$› **by** *blast*
  show $(\sum i < size(annos\ C2).\ m\_o\ ?n\ (annos\ C2\ !\ i))$
        $< (\sum i < size(annos\ C2).\ m\_o\ ?n\ (annos\ C1\ !\ i))$
    apply(*rule setsum_strict_mono_ex1*) **using** *1 2* **by** (*auto*)
qed

end

locale *Abs_Int_measure* =
  Abs_Int_mono **where** γ=γ + *Measure* **where** *m*=*m*
  for γ :: *'av::semilattice* ⇒ *val set* **and** *m* :: *'av* ⇒ *nat*
begin

lemma *AI_Some_measure*: ∃ *C*. *AI c* = *Some C*
unfolding *AI_def*
apply(*rule pfp_termination*[**where** *I* = %*C*. *strip C* = *c* ∧ *C* ∈ *L*(*vars c*)
  and *m*=*m_c*])
apply(*simp_all add*: *m_c2 mono_step'_top bot_least*)
done

end

end

theory *Abs_Int1_const*
imports *Abs_Int1*
begin

12.8 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* :: *semilattice*
begin

```
fun le_const where
_ ⊑ Any = True |
Const n ⊑ Const m = (n=m) |
Any ⊑ Const _ = False

fun join_const where
Const m ⊔ Const n = (if n=m then Const m else Any) |
_ ⊔ _ = Any

definition ⊤ = 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'*
..

12.8.1 Tests

definition *steps c i* = (*step_const*(*top*(*vars c*)) ^^ *i*) (*bot c*)

value *show_acom* (*steps test1_const 0*)
value *show_acom* (*steps test1_const 1*)
value *show_acom* (*steps test1_const 2*)
value *show_acom* (*steps test1_const 3*)
value *show_acom* (*the*(*AI_const test1_const*))

value *show_acom* (*the*(*AI_const test2_const*))
value *show_acom* (*the*(*AI_const test3_const*))

value *show_acom* (*steps test4_const 0*)
value *show_acom* (*steps test4_const 1*)
value *show_acom* (*steps test4_const 2*)
value *show_acom* (*steps test4_const 3*)
value *show_acom* (*steps test4_const 4*)
value *show_acom* (*the*(*AI_const test4_const*))

value *show_acom* (*steps test5_const 0*)
value *show_acom* (*steps test5_const 1*)
value *show_acom* (*steps test5_const 2*)
value *show_acom* (*steps test5_const 3*)
value *show_acom* (*steps test5_const 4*)
value *show_acom* (*steps test5_const 5*)
value *show_acom* (*steps test5_const 6*)
value *show_acom* (*the*(*AI_const test5_const*))

value *show_acom* (*steps test6_const 0*)
value *show_acom* (*steps test6_const 1*)
value *show_acom* (*steps test6_const 2*)
value *show_acom* (*steps test6_const 3*)
value *show_acom* (*steps test6_const 4*)
value *show_acom* (*steps test6_const 5*)
value *show_acom* (*steps test6_const 6*)
value *show_acom* (*steps test6_const 7*)
value *show_acom* (*steps test6_const 8*)
value *show_acom* (*steps test6_const 9*)
value *show_acom* (*steps test6_const 10*)

value *show_acom* (*steps test6_const 11*)
value *show_acom* (*steps test6_const 12*)
value *show_acom* (*steps test6_const 13*)
value *show_acom* (*the*(*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*)

interpretation *Abs_Int_measure*
where $\gamma = \gamma$*_const* **and** *num'* = *Const* **and** *plus'* = *plus_const*
and *m* = *m_const* **and** *h* = *1*
proof
  case *goal1* **thus** *?case* **by**(*auto simp*: *m_const_def split*: *const.splits*)
next
  case *goal2* **thus** *?case* **by**(*auto simp*: *m_const_def split*: *const.splits*)
next
  case *goal3* **thus** *?case* **by**(*auto simp*: *m_const_def split*: *const.splits*)
qed

thm *AI_Some_measure*

end

theory *Abs_Int1_parity*
imports *Abs_Int1*
begin

12.9 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 ⊑. Note that the header of the definition must refer to the ascii name *op* ⊑ 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 *semilattice* with type *parity*:

instantiation *parity* :: *semilattice*
begin

definition *join_parity* **where**
$x \sqcup y = (\textit{if } x \sqsubseteq y \textit{ then } y \textit{ else if } y \sqsubseteq x \textit{ then } x \textit{ else } Either)$

definition *Top_parity* **where**
$\top = Either$

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 mod 2 = 0*} |
γ*_parity Odd* = {*i. i mod 2 = 1*} |
γ*_parity Either* = *UNIV*

fun *num_parity* :: *val* $\Rightarrow$ *parity* **where**
num_parity i = (*if i mod 2 = 0 then Even else 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_parity*:

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*
..

12.9.1 Tests

definition *test1_parity* =
  ''x'' ::= N 1;
  WHILE Less (V ''x'') (N 100) DO ''x'' ::= Plus (V ''x'') (N 2)
value [*code*] *show_acom* (*the*(*AI_parity test1_parity*))

definition *test2_parity* =
  ''x'' ::= N 1;
  WHILE Less (V ''x'') (N 100) DO ''x'' ::= Plus (V ''x'') (N 3)

definition *steps c i* = (*step_parity*(*top*(*vars c*)) ^^ *i*) (*bot c*)

value *show_acom* (*steps test2_parity 0*)
value *show_acom* (*steps test2_parity 1*)
value *show_acom* (*steps test2_parity 2*)
value *show_acom* (*steps test2_parity 3*)
value *show_acom* (*steps test2_parity 4*)
value *show_acom* (*steps test2_parity 5*)
value *show_acom* (*steps test2_parity 6*)
value *show_acom* (*the*(*AI_parity test2_parity*))

12.9.2 Termination

interpretation *Abs_Int_mono*
where γ = γ*_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*)

interpretation *Abs_Int_measure*
where $\gamma = \gamma\_parity$ **and** $num' = num\_parity$ **and** $plus' = plus\_parity$
and $m = m\_parity$ **and** $h = 1$
proof
  case *goal1* **thus** *?case* **by**(*auto simp add*: *m_parity_def le_parity_def*)
next
  case *goal2* **thus** *?case* **by**(*auto simp add*: *m_parity_def le_parity_def*)
next
  case *goal3* **thus** *?case* **by**(*auto simp add*: *m_parity_def le_parity_def*)
qed

thm *AI_Some_measure*

end

theory *Abs_Int2*
imports *Abs_Int1*
begin

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

definition $le\_prod\ p1\ p2 = (fst\ p1 \sqsubseteq fst\ p2 \wedge snd\ p1 \sqsubseteq snd\ p2)$

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

12.10 Backward Analysis of Expressions

class *lattice* = *semilattice* + *bot* +
fixes $meet :: 'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $\sqcap$ *65*)
assumes *meet_le1* [*simp*]: $x \sqcap y \sqsubseteq x$
and *meet_le2* [*simp*]: $x \sqcap y \sqsubseteq y$
and *meet_greatest*: $x \sqsubseteq y \Longrightarrow x \sqsubseteq z \Longrightarrow x \sqsubseteq y \sqcap z$
begin

lemma *mono_meet*: $x \sqsubseteq x' \Longrightarrow y \sqsubseteq y' \Longrightarrow x \sqcap y \sqsubseteq x' \sqcap y'$
by (*metis meet_greatest meet_le1 meet_le2 le_trans*)

end

locale *Val_abs1_gamma* =
  Gamma **where** $\gamma = \gamma$ **for** γ :: *'av::lattice* ⇒ *val set* +
assumes *inter_gamma_subset_gamma_meet*:
  $\gamma\ a1 \cap \gamma\ a2 \subseteq \gamma(a1 \sqcap a2)$
and *gamma_bot*[*simp*]: $\gamma\ \bot = \{\}$
begin

lemma *in_gamma_meet*: $x : \gamma\ a1 \Longrightarrow x : \gamma\ a2 \Longrightarrow x : \gamma(a1 \sqcap a2)$
by (*metis IntI inter_gamma_subset_gamma_meet 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::lattice* ⇒ *val set* +
fixes *test_num'* :: *val* ⇒ *'av* ⇒ *bool*
and *filter_plus'* :: *'av* ⇒ *'av* ⇒ *'av* ⇒ *'av* ∗ *'av*
and *filter_less'* :: *bool* ⇒ *'av* ⇒ *'av* ⇒ *'av* ∗ *'av*
assumes *test_num'*: *test_num' n a* = $(n : \gamma\ a)$
and *filter_plus'*: *filter_plus' a a1 a2* = $(b1,b2) \Longrightarrow$
  $n1 : \gamma\ a1 \Longrightarrow n2 : \gamma\ a2 \Longrightarrow n1{+}n2 : \gamma\ a \Longrightarrow n1 : \gamma\ b1 \wedge n2 : \gamma\ b2$
and *filter_less'*: *filter_less'* $(n1{<}n2)\ a1\ a2 = (b1,b2) \Longrightarrow$
  $n1 : \gamma\ a1 \Longrightarrow n2 : \gamma\ a2 \Longrightarrow n1 : \gamma\ b1 \wedge n2 : \gamma\ b2$

locale *Abs_Int1* =
  Val_abs1 **where** $\gamma = \gamma$ **for** γ :: *'av::lattice* ⇒ *val set*
begin

lemma *in_gamma_join_UpI*:
  $S1 \in L\ X \Longrightarrow S2 \in L\ X \Longrightarrow s : \gamma_o\ S1 \vee s : \gamma_o\ S2 \Longrightarrow s : \gamma_o(S1 \sqcup S2)$
by (*metis* (*hide_lams*, *no_types*) *semilatticeL_class.join_ge1 semilatticeL_class.join_ge2 mono_gamma_o subsetD*)

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

lemma *aval''_sound*: $s : \gamma_o\ S \Longrightarrow S \in L\ X \Longrightarrow vars\ a \subseteq X \Longrightarrow aval\ a\ s : \gamma(aval''\ a\ S)$
by(*simp add*: *L_option_def L_st_def aval'_sound split*: *option.splits*)

12.10.1 Backward analysis

fun *afilter* :: *aexp* ⇒ *'av* ⇒ *'av st option* ⇒ *'av st option* **where**
afilter (*N n*) *a S* = (*if test_num' n a then S else None*) |
afilter (*V x*) *a S* = (*case S of None* ⇒ *None* | *Some S* ⇒
  let a' = *fun 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 *Abs_Int0.bot* in the *V*-case is important: *Abs_Int0.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-*Abs_Int0.bot* values. Otherwise the (pointwise) join of two abstract states, one of which contains *Abs_Int0.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* (*a1*,*a2*) = *filter_less' res* (*aval'' e1 S*) (*aval'' e2 S*)
   in afilter e1 a1 (*afilter e2 a2 S*))

lemma *afilter_in_L*: $S \in L\ X \Longrightarrow vars\ e \subseteq X \Longrightarrow afilter\ e\ a\ S \in L\ X$
by(*induction e arbitrary*: *a S*)
  (*auto simp*: *Let_def update_def L_st_def*
      split: *option.splits prod.split*)

lemma *afilter_sound*: $S \in L\ X \Longrightarrow vars\ e \subseteq X \Longrightarrow$

$s : \gamma_o\ S \Longrightarrow$ *aval e s* $: \gamma\ a \Longrightarrow s : \gamma_o$ (*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** $s : \gamma_s\ S'$ **using** ‹$s : \gamma_o\ S$›
by(*auto simp*: *in_gamma_option_iff*)
moreover hence $s\ x : \gamma$ (*fun* S' *x*)
using *V*(*1*,*2*) **by**(*simp add*: *γ_st_def L_st_def*)
moreover have $s\ x : \gamma\ a$ **using** *V* **by** *simp*
ultimately show *?case* **using** *V*(*3*)
by(*simp add*: *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.prems*(*3*)] *aval''_sound*[*OF Plus.prems*(*3*)]]
by (*auto simp*: *afilter_in_L split*: *prod.split*)
qed

lemma *bfilter_in_L*: $S \in L\ X \Longrightarrow vars\ b \subseteq X \Longrightarrow bfilter\ b\ bv\ S \in L\ X$
by(*induction b arbitrary*: *bv S*)(*auto simp*: *afilter_in_L split*: *prod.split*)

lemma *bfilter_sound*: $S \in L\ X \Longrightarrow vars\ b \subseteq X \Longrightarrow$
$s : \gamma_o\ S \Longrightarrow bv = bval\ b\ s \Longrightarrow s : \gamma_o(bfilter\ b\ bv\ S)$
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 *simp* (*metis And*(*1*) *And*(*2*) *bfilter_in_L in_gamma_join_UpI*)
next
case (*Less e1 e2*) **thus** *?case*
by(*auto split*: *prod.split*)
(*metis* (*lifting*) *afilter_in_L afilter_sound aval''_sound filter_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* {*P1*} *C1 ELSE* {*P2*} *C2* {*Q*}) =
  (*let P1′* = *bfilter b True S*; *C1′* = *step′ P1 C1*; *P2′* = *bfilter b False S*; *C2′* = *step′ P2 C2*
   in IF b THEN {*P1′*} *C1′ ELSE* {*P2′*} *C2′* {*post C1* ⊔ *post C2*}) |
step′ S ({*I*} *WHILE b DO* {*p*} *C* {*Q*}) =
   {*S* ⊔ *post C*}
   WHILE b DO {*bfilter b True I*} *step′ p C*
   {*bfilter b False I*}

definition *AI* :: *com* ⇒ *′av st option acom option* **where**
AI c = *pfp* (*step′* $\top_{vars\ c}$) (*bot c*)

lemma *strip_step′*[*simp*]: *strip*(*step′ S c*) = *strip c*
by(*induct c arbitrary*: *S*) (*simp_all add*: *Let_def*)

12.10.2 Soundness

lemma *in_gamma_update*:
  ⟦ *s* : γ_s *S*; *i* : γ *a* ⟧ ⟹ *s*(*x* := *i*) : γ_s(*update S x a*)
by(*simp add*: *γ_st_def*)

lemma *step_step′*: *C* ∈ *L X* ⟹ *S* ∈ *L X* ⟹ *step* (γ_o *S*) (γ_c *C*) ≤ γ_c (*step′ S C*)
proof(*induction C arbitrary*: *S*)
  case *SKIP* **thus** *?case* **by** *auto*
next
  case *Assign* **thus** *?case*
    by (*fastforce simp*: *L_st_def intro*: *aval′_sound in_gamma_update split*: *option.splits*)
next
  case *Seq* **thus** *?case* **by** *auto*
next
  case (*If b _ C1 _ C2*)
  hence *0*: *post C1* ⊑ *post C1* ⊔ *post C2* ∧ *post C2* ⊑ *post C1* ⊔ *post C2*
   by(*simp*, *metis post_in_L join_ge1 join_ge2*)
  have *vars b* ⊆ *X* **using** *If.prems* **by** *simp*
  note *vars* = ‹*S* ∈ *L X*› ‹*vars b* ⊆ *X*›
  show *?case* **using** *If 0*
    by (*auto simp*: *mono_gamma_o bfilter_sound*[*OF vars*] *bfilter_in_L*[*OF vars*])
next
  case (*While I b*)

hence *vars*: $I \in L\ X\ vars\ b \subseteq X$ **by** *simp_all*
thus *?case* **using** *While*
by (*auto simp*: *mono_gamma_o bfilter_sound*[*OF vars*] *bfilter_in_L*[*OF vars*])
qed

lemma *step'_in_L*[*simp*]: $\llbracket\ C \in L\ X;\ S \in L\ X\ \rrbracket \Longrightarrow step'\ S\ C \in L\ X$
proof(*induction C arbitrary*: *S*)
case *Assign* **thus** *?case* **by**(*simp add*: *L_option_def L_st_def update_def split*: *option.splits*)
qed (*auto simp add*: *bfilter_in_L*)

lemma *AI_sound*: $AI\ c = Some\ C \Longrightarrow CS\ c \le \gamma_c\ C$
proof(*simp add*: *CS_def AI_def*)
assume *1*: $pfp\ (step'\ (top(vars\ c)))\ (bot\ c) = Some\ C$
have $C \in L(vars\ c)$
by(*rule pfp_inv*[**where** $P = \%C.\ C \in L(vars\ c)$, *OF 1 _ bot_in_L*])
(*erule step'_in_L*[*OF _ top_in_L*])
have *pfp'*: $step'\ (top(vars\ c))\ C \sqsubseteq C$ **by**(*rule pfp_pfp*[*OF 1*])
have *2*: $step\ (\gamma_o(top(vars\ c)))\ (\gamma_c\ C) \le \gamma_c\ C$
proof(*rule order_trans*)
show $step\ (\gamma_o\ (top(vars\ c)))\ (\gamma_c\ C) \le\ \gamma_c\ (step'\ (top(vars\ c))\ C)$
by(*rule step_step'*[*OF* ‹$C \in L(vars\ c)$› *top_in_L*])
show $\gamma_c\ (step'\ (top(vars\ c))\ C) \le \gamma_c\ C$
by(*rule mono_gamma_c*[*OF pfp'*])
qed
have *3*: $strip\ (\gamma_c\ C) = c$ **by**(*simp add*: *strip_pfp*[*OF _ 1*])
have $lfp\ c\ (step\ (\gamma_o(top(vars\ c)))) \le \gamma_c\ C$
by(*rule lfp_lowerbound*[*simplified*,**where** f=$step\ (\gamma_o(top(vars\ c)))$, *OF 3 2*])
thus $lfp\ c\ (step\ UNIV) \le \gamma_c\ C$ **by** *simp*
qed

end

12.10.3 Monotonicity

locale *Abs_Int1_mono* = *Abs_Int1* +
assumes *mono_plus'*: $a1 \sqsubseteq b1 \Longrightarrow a2 \sqsubseteq b2 \Longrightarrow plus'\ a1\ a2 \sqsubseteq plus'\ b1\ b2$
and *mono_filter_plus'*: $a1 \sqsubseteq b1 \Longrightarrow a2 \sqsubseteq b2 \Longrightarrow r \sqsubseteq r' \Longrightarrow$
$filter\_plus'\ r\ a1\ a2 \sqsubseteq filter\_plus'\ r'\ b1\ b2$
and *mono_filter_less'*: $a1 \sqsubseteq b1 \Longrightarrow a2 \sqsubseteq b2 \Longrightarrow$
$filter\_less'\ bv\ a1\ a2 \sqsubseteq filter\_less'\ bv\ b1\ b2$
begin

lemma *mono_aval'*:
  $S1 \sqsubseteq S2 \Longrightarrow S1 \in L\ X \Longrightarrow vars\ e \subseteq X \Longrightarrow aval'\ e\ S1 \sqsubseteq aval'\ e\ S2$
by(*induction e*) (*auto simp*: *le_st_def mono_plus' L_st_def*)

lemma *mono_aval''*:
  $S1 \sqsubseteq S2 \Longrightarrow S1 \in L\ X \Longrightarrow vars\ e \subseteq X \Longrightarrow aval''\ e\ S1 \sqsubseteq aval''\ e\ S2$
apply(*cases S1*)
 apply *simp*
apply(*cases S2*)
 apply *simp*
by (*simp add*: *mono_aval'*)

lemma *mono_afilter*: $S1 \in L\ X \Longrightarrow S2 \in L\ X \Longrightarrow vars\ e \subseteq X \Longrightarrow$
  $r1 \sqsubseteq r2 \Longrightarrow S1 \sqsubseteq S2 \Longrightarrow afilter\ e\ r1\ S1 \sqsubseteq afilter\ e\ r2\ S2$
apply(*induction e arbitrary*: *r1 r2 S1 S2*)
apply(*auto simp*: *test_num' Let_def mono_meet split*: *option.splits prod.splits*)
apply (*metis mono_gamma subsetD*)
apply(*drule* (*2*) *mono_fun_L*)
apply (*metis mono_meet le_trans*)
apply(*metis mono_aval'' mono_filter_plus'*[*simplified le_prod_def*] *fst_conv snd_conv afilter_in_L*)
done

lemma *mono_bfilter*: $S1 \in L\ X \Longrightarrow S2 \in L\ X \Longrightarrow vars\ b \subseteq X \Longrightarrow$
  $S1 \sqsubseteq S2 \Longrightarrow bfilter\ b\ bv\ S1 \sqsubseteq bfilter\ b\ bv\ S2$
apply(*induction b arbitrary*: *bv S1 S2*)
apply(*simp*)
apply(*simp*)
apply *simp*
apply(*metis join_least le_trans*[*OF _ join_ge1*] *le_trans*[*OF _ join_ge2*] *bfilter_in_L*)
apply (*simp split*: *prod.splits*)
apply(*metis mono_aval'' mono_afilter mono_filter_less'*[*simplified le_prod_def*] *fst_conv snd_conv afilter_in_L*)
done

theorem *mono_step'*: $S1 \in L\ X \Longrightarrow S2 \in L\ X \Longrightarrow C1 \in L\ X \Longrightarrow C2 \in L\ X \Longrightarrow$
  $S1 \sqsubseteq S2 \Longrightarrow C1 \sqsubseteq C2 \Longrightarrow step'\ S1\ C1 \sqsubseteq step'\ S2\ C2$
apply(*induction C1 C2 arbitrary*: *S1 S2 rule*: *le_acom.induct*)
apply (*auto simp*: *Let_def mono_bfilter mono_aval' mono_post*
  le_join_disj le_join_disj[*OF post_in_L post_in_L*] *bfilter_in_L*
    split: *option.split*)

done

lemma *mono_step'_top*: $C1 \in L\ X \Longrightarrow C2 \in L\ X \Longrightarrow$
$C1 \sqsubseteq C2 \Longrightarrow step'\ (top\ X)\ C1 \sqsubseteq step'\ (top\ X)\ C2$
by (*metis top_in_L mono_step' preord_class.le_refl*)

end

end

theory *Abs_Int2_ivl*
imports *Abs_Int2*
begin

12.11 Interval Analysis

datatype *ivl* = *Ivl int option int option*

definition *γ_ivl i* = (*case i of*
Ivl (*Some l*) (*Some h*) ⇒ {*l..h*} |
Ivl (*Some l*) *None* ⇒ {*l..*} |
Ivl None (*Some h*) ⇒ {*..h*} |
Ivl None None ⇒ *UNIV*)

abbreviation *Ivl_Some_Some* :: *int* ⇒ *int* ⇒ *ivl* ({_..._}) **where**
{*lo...hi*} == *Ivl* (*Some lo*) (*Some hi*)
abbreviation *Ivl_Some_None* :: *int* ⇒ *ivl* ({_...}) **where**
{*lo...*} == *Ivl* (*Some lo*) *None*
abbreviation *Ivl_None_Some* :: *int* ⇒ *ivl* ({..._}) **where**
{*...hi*} == *Ivl None* (*Some hi*)
abbreviation *Ivl_None_None* :: *ivl* ({...}) **where**
{...} == *Ivl None None*

definition *num_ivl n* = {*n...n*}

fun *in_ivl* :: *int* ⇒ *ivl* ⇒ *bool* **where**
in_ivl k (*Ivl* (*Some l*) (*Some h*)) ⟷ $l \le k \land k \le h$ |
in_ivl k (*Ivl* (*Some l*) *None*) ⟷ $l \le k$ |
in_ivl k (*Ivl None* (*Some h*)) ⟷ $k \le h$ |
in_ivl k (*Ivl 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(Ivl 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 (Ivl l1 h1, Ivl l2 h2) ⇒ Ivl (l1+l2) (h1+h2))

instantiation *ivl* :: *semilattice*
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 (Ivl l1 h1) (Ivl 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* (*Ivl l1 h1*, *Ivl l2 h2*) ⇒
Ivl (*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* :: *lattice*

begin

definition *i1* ⊓ *i2* = (*if is_empty i1* ∨ *is_empty i2 then empty else*
  case (*i1*,*i2*) *of* (*Ivl l1 h1*, *Ivl l2 h2*) ⇒
    Ivl (*max_option False l1 l2*) (*min_option True h1 h2*))

definition ⊥ = *empty*

instance
proof
  case *goal2* **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 *goal3* **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 *goal4* **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 *goal1* **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* (*Ivl l1 h1*, *Ivl l2 h2*) ⇒ *Ivl* (*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 (*Ivl l1 h1*) (*Ivl l2 h2*) =
  (*if is_empty*(*Ivl l1 h1*) ∨ *is_empty*(*Ivl l2 h2*) *then* (*empty*, *empty*) *else*
   if res
   then (*Ivl l1* (*min_option True h1* (*h2* − *Some 1*)),
         Ivl (*max_option False* (*l1* + *Some 1*) *l2*) *h2*)
   else (*Ivl* (*max_option False l1 l2*) *h1*, *Ivl 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 γ = γ*_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 ⊑ *i1′* ⟹ *i2* ⊑ *i2′* ⟹ *minus_ivl i1 i2* ⊑ *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

12.11.1 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*)

definition *steps c i* = (*step_ivl*(*top*(*vars c*)) ^^ *i*) (*bot c*)

value *show_acom_opt* (*AI_ivl test2_ivl*)
value *show_acom* (*steps test2_ivl 0*)
value *show_acom* (*steps test2_ivl 1*)
value *show_acom* (*steps test2_ivl 2*)
value *show_acom* (*steps test2_ivl 3*)

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* (*steps test3_ivl 0*)
value *show_acom* (*steps test3_ivl 1*)
value *show_acom* (*steps test3_ivl 2*)
value *show_acom* (*steps test3_ivl 3*)
value *show_acom* (*steps test3_ivl 4*)
value *show_acom* (*steps test3_ivl 5*)

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* (*steps test4_ivl 50*)

Relationships between variables are NOT captured:

value *show_acom_opt* (*AI_ivl test5_ivl*)

Again, the analysis would not terminate:

value *show_acom* (*steps test6_ivl 50*)

end

theory *Abs_Int3*
imports *Abs_Int2_ivl*
begin

12.11.2 Welltypedness

class *Lc* =
fixes *Lc* :: *com* ⇒ *'a set*

instantiation *st* :: (*type*)*Lc*
begin

definition *Lc_st* :: *com* ⇒ *'a st set* **where**
Lc_st c = *L* (*vars c*)

instance ..

end

instantiation *acom* :: (*Lc*)*Lc*
begin

definition *Lc_acom* :: *com* ⇒ *'a acom set* **where**
Lc c = {*C*. *strip C* = *c* ∧ (∀ *a*∈*set*(*annos C*). *a* ∈ *Lc c*)}

instance ..

end

instantiation *option* :: (*Lc*)*Lc*
begin

definition *Lc_option* :: *com* ⇒ *'a option set* **where**
Lc c = {*None*} ∪ *Some* ‘ *Lc c*

lemma *Lc_option_simps*[*simp*]: *None* ∈ *Lc c* (*Some x* ∈ *Lc c*) = (*x* ∈ *Lc c*)
by(*auto simp*: *Lc_option_def*)

instance ..

end

lemma *Lc_option_iff_wt*[*simp*]: **fixes** a :: _ *st option*
shows $(a \in Lc\ c) = (a \in L\ (vars\ c))$
by(*auto simp add*: *L_option_def Lc_st_def split*: *option.splits*)

context *Abs_Int1*
begin

lemma *step'_in_Lc*: $C \in Lc\ c \Longrightarrow S \in Lc\ c \Longrightarrow step'\ S\ C \in Lc\ c$
apply(*auto simp add*: *Lc_acom_def*)
by(*metis step'_in_L*[*simplified L_acom_def mem_Collect_eq*] *order_refl*)

end

12.12 Widening and Narrowing

class *widen* =
fixes *widen* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infix** ∇ *65*)

class *narrow* =
fixes *narrow* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infix** $\triangle$ *65*)

class *WN* = *widen* + *narrow* + *preord* +
assumes *widen1*: $x \sqsubseteq x \nabla y$
assumes *widen2*: $y \sqsubseteq x \nabla y$
assumes *narrow1*: $y \sqsubseteq x \Longrightarrow y \sqsubseteq x \triangle y$
assumes *narrow2*: $y \sqsubseteq x \Longrightarrow x \triangle y \sqsubseteq x$

class *WN_Lc* = *widen* + *narrow* + *preord* + *Lc* +
assumes *widen1*: $x \in Lc\ c \Longrightarrow y \in Lc\ c \Longrightarrow x \sqsubseteq x \nabla y$
assumes *widen2*: $x \in Lc\ c \Longrightarrow y \in Lc\ c \Longrightarrow y \sqsubseteq x \nabla y$
assumes *narrow1*: $y \sqsubseteq x \Longrightarrow y \sqsubseteq x \triangle y$
assumes *narrow2*: $y \sqsubseteq x \Longrightarrow x \triangle y \sqsubseteq x$
assumes *Lc_widen*[*simp*]: $x \in Lc\ c \Longrightarrow y \in Lc\ c \Longrightarrow x \nabla y \in Lc\ c$
assumes *Lc_narrow*[*simp*]: $x \in Lc\ c \Longrightarrow y \in Lc\ c \Longrightarrow x \triangle y \in Lc\ c$

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* (*Ivl l1 h1*, *Ivl l2 h2*) ⇒
Ivl (*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* (*Ivl l1 h1*, *Ivl l2 h2*) ⇒
Ivl (*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

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

definition *widen_st F1 F2* = *FunDom* (*λx. fun F1 x* ∇ *fun F2 x*) (*dom F1*)

definition *narrow_st F1 F2* = *FunDom* (*λx. fun F1 x* △ *fun F2 x*) (*dom F1*)

instance
proof
case *goal1* **thus** *?case*
by(*simp add*: *widen_st_def le_st_def WN_class.widen1*)
next
case *goal2* **thus** *?case*
by(*simp add*: *widen_st_def le_st_def WN_class.widen2 Lc_st_def L_st_def*)
next
case *goal3* **thus** *?case*
by(*auto simp*: *narrow_st_def le_st_def WN_class.narrow1*)
next
case *goal4* **thus** *?case*
by(*auto simp*: *narrow_st_def le_st_def WN_class.narrow2*)

next
  case *goal5* **thus** *?case* **by**(*auto simp*: *widen_st_def Lc_st_def L_st_def*)
next
  case *goal6* **thus** *?case* **by**(*auto simp*: *narrow_st_def Lc_st_def L_st_def*)
qed

end

instantiation *option* :: (*WN_Lc*) *WN_Lc*
begin

fun *widen_option* **where**
None ∇ *x* = *x* |
x ∇ *None* = *x* |
(*Some x*) ∇ (*Some y*) = *Some*(*x* ∇ *y*)

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

instance
proof
  case *goal1* **thus** *?case*
    by(*induct x y rule*: *widen_option.induct*)(*simp_all add*: *widen1*)
next
  case *goal2* **thus** *?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*)
next
  case *goal5* **thus** *?case*
    by(*induction x y rule*: *widen_option.induct*)(*auto simp*: *Lc_st_def*)
next
  case *goal6* **thus** *?case*
    by(*induction x y rule*: *narrow_option.induct*)(*auto simp*: *Lc_st_def*)
qed

end

fun *map2_acom* :: $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a$ *acom* $\Rightarrow 'a$ *acom* $\Rightarrow 'a$ *acom* **where**
map2_acom f (SKIP {a1}) (SKIP {a2}) = (SKIP {f a1 a2}) |
map2_acom f (x ::= e {a1}) (x' ::= e' {a2}) = (x ::= e {f a1 a2}) |
map2_acom f (C1;C2) (D1;D2) = (map2_acom f C1 D1; map2_acom f C2 D2) |
map2_acom f (IF b THEN {p1} C1 ELSE {p2} C2 {a1}) (IF b' THEN {q1} D1 ELSE {q2} D2 {a2}) =
(IF b THEN {f p1 q1} map2_acom f C1 D1 ELSE {f p2 q2} map2_acom f C2 D2 {f a1 a2}) |
map2_acom f ({a1} WHILE b DO {p} C {a2}) ({a3} WHILE b' DO {p'} C' {a4}) =
({f a1 a3} WHILE b DO {f p p'} map2_acom f C C' {f a2 a4})

instantiation *acom* :: *(widen)widen*
begin
definition *widen_acom* = *map2_acom* (*op* ∇)
instance ..
end

instantiation *acom* :: *(narrow)narrow*
begin
definition *narrow_acom* = *map2_acom* (*op* $\triangle$)
instance ..
end

instantiation *acom* :: *(WN_Lc)WN_Lc*
begin

lemma *widen_acom1*: **fixes** *C1* :: *'a acom* **shows**
$[\![\forall a \in set(annos\ C1).\ a \in Lc\ c; \forall a \in set\ (annos\ C2).\ a \in Lc\ c; strip\ C1 = strip\ C2]\!]$
$\Longrightarrow C1 \sqsubseteq C1 \nabla C2$
by(*induct C1 C2 rule*: *le_acom.induct*)
(*auto simp*: *widen_acom_def widen1 Lc_acom_def*)

lemma *widen_acom2*: **fixes** *C1* :: *'a acom* **shows**
$[\![\forall a \in set(annos\ C1).\ a \in Lc\ c; \forall a \in set\ (annos\ C2).\ a \in Lc\ c; strip\ C1 = strip\ C2]\!]$
$\Longrightarrow C2 \sqsubseteq C1 \nabla C2$
by(*induct C1 C2 rule*: *le_acom.induct*)
(*auto simp*: *widen_acom_def widen2 Lc_acom_def*)

lemma *strip_map2_acom*[*simp*]:
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 *strip_widen_acom*[*simp*]:
strip C1 = *strip C2* $\Longrightarrow$ *strip*(*C1* ∇ *C2*) = *strip C1*
by(*simp add*: *widen_acom_def*)

lemma *strip_narrow_acom*[*simp*]:
strip C1 = *strip C2* $\Longrightarrow$ *strip*(*C1* $\triangle$ *C2*) = *strip C1*
by(*simp add*: *narrow_acom_def*)

lemma *annos_map2_acom*[*simp*]: *strip C2* = *strip C1* $\Longrightarrow$
annos(*map2_acom f C1 C2*) = *map* (%(*x*,*y*).*f x y*) (*zip* (*annos C1*) (*annos C2*))
by(*induction f C1 C2 rule*: *map2_acom.induct*)(*simp_all add*: *size_annos_same2*)

instance
proof
case *goal1* **thus** *?case* **by**(*auto simp*: *Lc_acom_def widen_acom1*)
next
case *goal2* **thus** *?case* **by**(*auto simp*: *Lc_acom_def widen_acom2*)
next
case *goal3* **thus** *?case*
by(*induct x y rule*: *le_acom.induct*)(*simp_all add*: *narrow_acom_def narrow1*)
next
case *goal4* **thus** *?case*
by(*induct x y rule*: *le_acom.induct*)(*simp_all add*: *narrow_acom_def narrow2*)
next
case *goal5* **thus** *?case*
by(*auto simp*: *Lc_acom_def widen_acom_def split_conv elim*!: *in_set_zipE*)
next
case *goal6* **thus** *?case*
by(*auto simp*: *Lc_acom_def narrow_acom_def split_conv elim*!: *in_set_zipE*)
qed

end

lemma *widen_o_in_L*[*simp*]: **fixes** *x1 x2* :: _ *st option*
shows *x1* $\in$ *L X* $\Longrightarrow$ *x2* $\in$ *L X* $\Longrightarrow$ *x1* ∇ *x2* $\in$ *L X*
by(*induction x1 x2 rule*: *widen_option.induct*)

($simp\_all$ add: $widen\_st\_def$ L_st_def)

lemma $narrow\_o\_in\_L[simp]$: **fixes** $x1\ x2 :: \_\ st\ option$
shows $x1 \in L\ X \Longrightarrow x2 \in L\ X \Longrightarrow x1 \bigtriangleup x2 \in L\ X$
by($induction\ x1\ x2\ rule$: $narrow\_option.induct$)
($simp\_all$ add: $narrow\_st\_def$ L_st_def)

lemma $widen\_c\_in\_L$: **fixes** $C1\ C2 :: \_\ st\ option\ acom$
shows $strip\ C1 = strip\ C2 \Longrightarrow C1 \in L\ X \Longrightarrow C2 \in L\ X \Longrightarrow C1 \nabla C2 \in L\ X$
by($induction\ C1\ C2\ rule$: $le\_acom.induct$)
($auto\ simp$: $widen\_acom\_def$)

lemma $narrow\_c\_in\_L$: **fixes** $C1\ C2 :: \_\ st\ option\ acom$
shows $strip\ C1 = strip\ C2 \Longrightarrow C1 \in L\ X \Longrightarrow C2 \in L\ X \Longrightarrow C1 \bigtriangleup C2 \in L\ X$
by($induction\ C1\ C2\ rule$: $le\_acom.induct$)
($auto\ simp$: $narrow\_acom\_def$)

lemma $bot\_in\_Lc[simp]$: $bot\ c \in Lc\ c$
by($simp\ add$: $Lc\_acom\_def\ bot\_def$)

12.12.1 Post-fixed point computation

definition $iter\_widen :: ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow ('a::\{preord,widen\})option$
where $iter\_widen\ f = while\_option\ (\lambda x.\ \neg\ f\ x \sqsubseteq x)\ (\lambda x.\ x \nabla f\ x)$

definition $iter\_narrow :: ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow ('a::\{preord,narrow\})option$
where $iter\_narrow\ f = while\_option\ (\lambda x.\ \neg\ x \sqsubseteq x \bigtriangleup f\ x)\ (\lambda x.\ x \bigtriangleup f\ x)$

definition $pfp\_wn :: ('a::\{preord,widen,narrow\} \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ option$
where $pfp\_wn\ f\ x =$
$(case\ iter\_widen\ f\ x\ of\ None \Rightarrow None \mid Some\ p \Rightarrow iter\_narrow\ f\ p)$

lemma $iter\_widen\_pfp$: $iter\_widen\ f\ x = Some\ p \Longrightarrow f\ p \sqsubseteq p$
by($auto\ simp\ add$: $iter\_widen\_def\ dest$: $while\_option\_stop$)

lemma $iter\_widen\_inv$:
assumes $!!x.\ P\ x \Longrightarrow P(f\ x)\ !!x1\ x2.\ P\ x1 \Longrightarrow P\ x2 \Longrightarrow P(x1 \nabla x2)$ **and** $P\ x$
and $iter\_widen\ f\ x = Some\ y$ **shows** $P\ y$
using $while\_option\_rule$[**where** $P = P$, $OF\ \_\ assms(4)[unfolded\ iter\_widen\_def]$]
by ($blast\ intro$: $assms(1-3)$)

```
lemma strip_while: fixes f :: 'a acom ⇒ 'a acom
assumes ∀ C. strip (f C) = strip C and while_option P f C = Some C'
shows strip C' = strip C
using while_option_rule[where P = λC'. strip C' = strip C, OF _ assms(2)]
by (metis assms(1))

lemma strip_iter_widen: fixes f :: 'a::{preord,widen} acom ⇒ 'a acom
assumes ∀ C. strip (f C) = strip C and iter_widen f C = Some C'
shows strip C' = strip C
proof−
  have ∀ C. strip(C ∇ f C) = strip C
    by (metis assms(1) strip_map2_acom widen_acom_def)
 from strip_while[OF this] assms(2) show ?thesis by(simp add: iter_widen_def)
qed

lemma iter_narrow_pfp:
assumes mono: !!x1 x2::_::WN_Lc. P x1 ⟹ P x2 ⟹ x1 ⊑ x2 ⟹ f x1
⊑ f x2
and Pinv: !!x. P x ⟹ P(f x) !!x1 x2. P x1 ⟹ P x2 ⟹ P(x1 △ x2)
and P p0 and f p0 ⊑ p0 and iter_narrow f p0 = Some p
shows P p ∧ f p ⊑ p
proof−
  let ?Q = %p. P p ∧ f p ⊑ p ∧ p ⊑ p0
  { fix p assume ?Q p
    note P = conjunct1[OF this] and 12 = conjunct2[OF this]
    note 1 = conjunct1[OF 12] and 2 = conjunct2[OF 12]
    let ?p' = p △ f p
    have ?Q ?p'
    proof auto
      show P ?p' by (blast intro: P Pinv)
      have f ?p' ⊑ f p by(rule mono[OF ‹P (p △ f p)›  P narrow2[OF 1]])
      also have ... ⊑ ?p' by(rule narrow1[OF 1])
      finally show f ?p' ⊑ ?p' .
      have ?p' ⊑ p by (rule narrow2[OF 1])
      also have p ⊑ p0 by(rule 2)
      finally show ?p' ⊑ p0 .
    qed
  }
  thus ?thesis
   using while_option_rule[where P = ?Q, OF _ assms(6)[simplified iter_narrow_def]]
    by (blast intro: assms(4,5) le_refl)
qed
```

lemma *pfp_wn_pfp*:
assumes *mono*: $!!x1\ x2{::}\_{::}WN\_Lc.\ P\ x1 \Longrightarrow P\ x2 \Longrightarrow x1 \sqsubseteq x2 \Longrightarrow f\ x1 \sqsubseteq f\ x2$
and *Pinv*: $P\ x\ \ !!x.\ P\ x \Longrightarrow P(f\ x)$
$!!x1\ x2.\ P\ x1 \Longrightarrow P\ x2 \Longrightarrow P(x1\ \nabla\ x2)$
$!!x1\ x2.\ P\ x1 \Longrightarrow P\ x2 \Longrightarrow P(x1\ \triangle\ x2)$
and *pfp_wn*: $pfp\_wn\ f\ x = Some\ p$ **shows** $P\ p \wedge f\ p \sqsubseteq p$
proof−
from *pfp_wn* **obtain** *p0*
where *its*: $iter\_widen\ f\ x = Some\ p0\ iter\_narrow\ f\ p0 = Some\ p$
by(*auto simp*: *pfp_wn_def split*: *option.splits*)
have *P p0* **by** (*blast intro*: *iter_widen_inv*[**where** *P=P*] *its*(*1*) *Pinv*(*1−3*))
thus *?thesis*
by − (*assumption* |
rule iter_narrow_pfp[**where** *P=P*] *mono Pinv*(*2*,*4*) *iter_widen_pfp its*)+
qed

lemma *strip_pfp_wn*:
$[\![\ \forall C.\ strip(f\ C) = strip\ C;\ pfp\_wn\ f\ C = Some\ C'\ ]\!] \Longrightarrow strip\ C' = strip\ C$
by(*auto simp add*: *pfp_wn_def iter_narrow_def split*: *option.splits*)
(*metis* (*no_types*) *narrow_acom_def strip_iter_widen strip_map2_acom strip_while*)

locale *Abs_Int2* = *Abs_Int1_mono*
where $\gamma=\gamma$ **for** γ :: $'av{::}\{WN,lattice\} \Rightarrow val\ set$
begin

definition *AI_wn* :: $com \Rightarrow\ 'av\ st\ option\ acom\ option$ **where**
$AI\_wn\ c = pfp\_wn\ (step'\ (top(vars\ c)))\ (bot\ c)$

lemma *AI_wn_sound*: $AI\_wn\ c = Some\ C \Longrightarrow CS\ c \leq \gamma_c\ C$
proof(*simp add*: *CS_def AI_wn_def*)
assume *1*: $pfp\_wn\ (step'\ (top(vars\ c)))\ (bot\ c) = Some\ C$
have *2*: $(strip\ C = c\ \&\ C \in L(vars\ c)) \wedge step'\ \top_{vars\ c}\ C \sqsubseteq C$
by(*rule pfp_wn_pfp*[**where** *x=bot c*])
(*simp_all add*: *1 mono_step'_top widen_c_in_L narrow_c_in_L*)
have *pfp*: $step\ (\gamma_o(top(vars\ c)))\ (\gamma_c\ C) \leq \gamma_c\ C$
proof(*rule order_trans*)
show $step\ (\gamma_o\ (top(vars\ c)))\ (\gamma_c\ C) \leq\ \gamma_c\ (step'\ (top(vars\ c))\ C)$
by(*rule step_step'*[*OF conjunct2*[*OF conjunct1*[*OF 2*]] *top_in_L*])
show ... $\leq \gamma_c\ C$
by(*rule mono_gamma_c*[*OF conjunct2*[*OF 2*]])

qed
have *3*: *strip* (γ_c *C*) = *c* **by**(*simp add*: *strip_pfp_wn*[*OF _ 1*])
have *lfp c* (*step* (γ_o (*top*(*vars c*)))) $\le$ γ_c *C*
by(*rule lfp_lowerbound*[*simplified*,**where** *f*=*step* (γ_o(*top*(*vars c*))), *OF 3 pfp*])
thus *lfp c* (*step UNIV*) $\le$ γ_c *C* **by** *simp*
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*
..

12.12.2 Tests

lemma [*code*]: *equal_class.equal* (*x*::'*a st*) *y* = *equal_class.equal x y*
by(*rule refl*)

definition *step_up_ivl n* =
((λ*C*. *C* ∇ *step_ivl* (*top*(*vars*(*strip C*))) *C*)^^*n*)
definition *step_down_ivl n* =
((λ*C*. *C* $\triangle$ *step_ivl* (*top*(*vars*(*strip C*))) *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 test3_ivl*))
value *show_acom* (*step_up_ivl 2* (*bot test3_ivl*))
value *show_acom* (*step_up_ivl 3* (*bot test3_ivl*))
value *show_acom* (*step_up_ivl 4* (*bot test3_ivl*))
value *show_acom* (*step_up_ivl 5* (*bot test3_ivl*))
value *show_acom* (*step_up_ivl 6* (*bot test3_ivl*))
value *show_acom* (*step_up_ivl 7* (*bot test3_ivl*))
value *show_acom* (*step_up_ivl 8* (*bot test3_ivl*))
value *show_acom* (*step_down_ivl 1* (*step_up_ivl 8* (*bot test3_ivl*)))
value *show_acom* (*step_down_ivl 2* (*step_up_ivl 8* (*bot test3_ivl*)))
value *show_acom* (*step_down_ivl 3* (*step_up_ivl 8* (*bot test3_ivl*)))
value *show_acom* (*step_down_ivl 4* (*step_up_ivl 8* (*bot test3_ivl*)))
value *show_acom_opt* (*AI_ivl' 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*)

12.12.3 Generic Termination Proof

locale *Measure_WN* = *Measure1* **where** *m=m* **for** $m :: {'av}{::}WN \Rightarrow nat$
+
fixes $n :: {'av} \Rightarrow nat$
assumes *m_widen*: $\sim y \sqsubseteq x \Longrightarrow m(x \nabla y) < m\ x$
assumes *n_mono*: $x \sqsubseteq y \Longrightarrow n\ x \le n\ y$
assumes *n_narrow*: $y \sqsubseteq x \Longrightarrow \sim x \sqsubseteq x \triangle y \Longrightarrow n(x \triangle y) < n\ x$

begin

lemma *m_s_widen*: $S1 \in L\ X \Longrightarrow S2 \in L\ X \Longrightarrow finite\ X \Longrightarrow$
  $\sim S2 \sqsubseteq S1 \Longrightarrow m\_s(S1 \nabla S2) < m\_s\ S1$
proof(*auto simp add*: *le_st_def m_s_def L_st_def widen_st_def*)
  assume *finite*(*dom S1*)
  have *1*: $\forall x{\in}dom\ S1.\ m(fun\ S1\ x) \ge m(fun\ S1\ x \nabla fun\ S2\ x)$
    by (*metis m1 WN_class.widen1*)
  fix *x* **assume** $x \in dom\ S1\ \neg\ fun\ S2\ x \sqsubseteq fun\ S1\ x$
  hence *2*: $EX\ x : dom\ S1.\ m(fun\ S1\ x) > m(fun\ S1\ x \nabla fun\ S2\ x)$
    using *m_widen* **by** *blast*
  from *setsum_strict_mono_ex1*[*OF* ‹*finite*(*dom S1*)› *1 2*]
  show $(\sum x{\in}dom\ S1.\ m\ (fun\ S1\ x \nabla fun\ S2\ x)) < (\sum x{\in}dom\ S1.\ m\ (fun\ S1\ x))$ **.**
qed

lemma *m_o_widen*: $[\![\ S1 \in L\ X;\ S2 \in L\ X;\ finite\ X;\ \neg\ S2 \sqsubseteq S1\ ]\!] \Longrightarrow$
  $m\_o\ (card\ X)\ (S1 \nabla S2) < m\_o\ (card\ X)\ S1$
by(*auto simp*: *m_o_def L_st_def m_s_h less_Suc_eq_le m_s_widen*
    split: *option.split*)

lemma *m_c_widen*:
  $C1 \in Lc\ c \Longrightarrow C2 \in Lc\ c \Longrightarrow \neg\ C2 \sqsubseteq C1 \Longrightarrow m\_c\ (C1 \nabla C2) < m\_c\ C1$
apply(*auto simp*: *Lc_acom_def m_c_def Let_def widen_acom_def*)
apply(*subgoal_tac length*(*annos C2*) = *length*(*annos C1*))
prefer *2* **apply** (*simp add*: *size_annos_same2*)
apply (*auto*)
apply(*rule setsum_strict_mono_ex1*)
apply *simp*
apply (*clarsimp*)
apply(*simp add*: *m_o1 finite_cvars widen1*[**where** *c* = *strip C2*])

apply(*auto simp*: *le_iff_le_annos listrel_iff_nth*)
apply(*rule_tac x=i* **in** *bexI*)
prefer *2* **apply** *simp*
apply(*rule m_o_widen*)
apply (*simp add*: *finite_cvars*)+
done

definition *n_s* :: *'av st* ⇒ *nat* (n_s) **where**
n_s *S* = (∑ *x*∈*dom S*. *n*(*fun S x*))

lemma *n_s_mono*: **assumes** *S1* ⊑ *S2* **shows** n_s *S1* ≤ n_s *S2*
proof−
from *assms* **have** [*simp*]: *dom S1* = *dom S2* ∀ *x*∈*dom S1*. *fun S1 x* ⊑ *fun S2 x*
by(*simp_all add*: *le_st_def*)
have (∑ *x*∈*dom S1*. *n*(*fun S1 x*)) ≤ (∑ *x*∈*dom S1*. *n*(*fun S2 x*))
by(*rule setsum_mono*)(*simp add*: *le_st_def n_mono*)
thus *?thesis* **by**(*simp add*: *n_s_def*)
qed

lemma *n_s_narrow*:
assumes *finite*(*dom S1*) **and** *S2* ⊑ *S1* ¬ *S1* ⊑ *S1* △ *S2*
shows n_s (*S1* △ *S2*) < n_s *S1*
proof−
from ‹*S2*⊑*S1*› **have** [*simp*]: *dom S1* = *dom S2* ∀ *x*∈*dom S1*. *fun S2 x* ⊑ *fun S1 x*
by(*simp_all add*: *le_st_def*)
have *1*: ∀ *x*∈*dom S1*. *n*(*fun* (*S1* △ *S2*) *x*) ≤ *n*(*fun S1 x*)
by(*auto simp*: *le_st_def narrow_st_def n_mono WN_class.narrow2*)
have *2*: ∃ *x*∈*dom S1*. *n*(*fun* (*S1* △ *S2*) *x*) < *n*(*fun S1 x*) **using** ‹¬ *S1* ⊑ *S1* △ *S2*›
by(*force simp*: *le_st_def narrow_st_def intro*: *n_narrow*)
have (∑ *x*∈*dom S1*. *n*(*fun* (*S1* △ *S2*) *x*)) < (∑ *x*∈*dom S1*. *n*(*fun S1 x*))
apply(*rule setsum_strict_mono_ex1*[*OF* ‹*finite*(*dom S1*)›]) **using** *1 2* **by** *blast*+
moreover have *dom* (*S1* △ *S2*) = *dom S1* **by**(*simp add*: *narrow_st_def*)
ultimately show *?thesis* **by**(*simp add*: *n_s_def*)
qed

definition *n_o* :: *'av st option* ⇒ *nat* (n_o) **where**
n_o *opt* = (*case opt of None* ⇒ *0* | *Some S* ⇒ n_s *S* + *1*)

```
lemma n_o_mono: S1 ⊑ S2 ⟹ n_o S1 ≤ n_o S2
by(induction S1 S2 rule: le_option.induct)(auto simp: n_o_def n_s_mono)

lemma n_o_narrow:
  S1 ∈ L X ⟹ S2 ∈ L X ⟹ finite X
  ⟹ S2 ⊑ S1 ⟹ ¬ S1 ⊑ S1 △ S2 ⟹ n_o (S1 △ S2) < n_o S1
apply(induction S1 S2 rule: narrow_option.induct)
apply(auto simp: n_o_def L_st_def n_s_narrow)
done


definition n_c :: 'av st option acom ⇒ nat (n_c) where
n_c C = (let as = annos C in ∑ i<size as. n_o (as!i))

lemma n_c_narrow: C1 ∈ Lc c ⟹ C2 ∈ Lc c ⟹
  C2 ⊑ C1 ⟹ ¬ C1 ⊑ C1 △ C2 ⟹ n_c (C1 △ C2) < n_c C1
apply(auto simp: n_c_def Let_def Lc_acom_def narrow_acom_def)
apply(subgoal_tac length(annos C2) = length(annos C1))
prefer 2 apply (simp add: size_annos_same2)
apply (auto)
apply(rule setsum_strict_mono_ex1)
apply simp
apply (clarsimp)
apply(rule n_o_mono)
apply(rule narrow2)
apply(fastforce simp: le_iff_le_annos listrel_iff_nth)
apply(auto simp: le_iff_le_annos listrel_iff_nth)
apply(rule_tac x=i in bexI)
prefer 2 apply simp
apply(rule n_o_narrow[where X = vars(strip C1)])
apply (simp add: finite_cvars)+
done

end


lemma iter_widen_termination:
fixes m :: 'a::WN_Lc ⇒ nat
assumes P_f: ⋀C. P C ⟹ P(f C)
and P_widen: ⋀C1 C2. P C1 ⟹ P C2 ⟹ P(C1 ∇ C2)
and m_widen: ⋀C1 C2. P C1 ⟹ P C2 ⟹ ~ C2 ⊑ C1 ⟹ m(C1 ∇
C2) < m C1
and P C shows EX C'. iter_widen f C = Some C'
proof(simp add: iter_widen_def,
```

rule measure_while_option_Some[**where** $P = P$ **and** $f=m$])
show $P\ C$ **by**(*rule* ‹$P\ C$›)
next
fix C **assume** $P\ C$ $\neg\ f\ C \sqsubseteq C$ **thus** $P\ (C \nabla f\ C) \wedge m\ (C \nabla f\ C) < m\ C$
by(*simp add*: *P_f P_widen m_widen*)
qed

lemma *iter_narrow_termination*:
fixes $n :: 'a::WN\_Lc \Rightarrow nat$
assumes *P_f*: $\bigwedge C.\ P\ C \Longrightarrow P(f\ C)$
and *P_narrow*: $\bigwedge C1\ C2.\ P\ C1 \Longrightarrow P\ C2 \Longrightarrow P(C1 \triangle C2)$
and *mono*: $\bigwedge C1\ C2.\ P\ C1 \Longrightarrow P\ C2 \Longrightarrow C1 \sqsubseteq C2 \Longrightarrow f\ C1 \sqsubseteq f\ C2$
and *n_narrow*: $\bigwedge C1\ C2.\ P\ C1 \Longrightarrow P\ C2 \Longrightarrow C2 \sqsubseteq C1 \Longrightarrow \sim C1 \sqsubseteq C1 \triangle C2 \Longrightarrow n(C1 \triangle C2) < n\ C1$
and *init*: $P\ C\ f\ C \sqsubseteq C$ **shows** $EX\ C'.\ iter\_narrow\ f\ C = Some\ C'$
proof(*simp add*: *iter_narrow_def*,
rule measure_while_option_Some[**where** $f=n$ **and** $P = \%C.\ P\ C \wedge f\ C \sqsubseteq C$])
show $P\ C \wedge f\ C \sqsubseteq C$ **using** *init* **by** *blast*
next
fix C **assume** *1*: $P\ C \wedge f\ C \sqsubseteq C$ **and** *2*: $\neg\ C \sqsubseteq C \triangle f\ C$
hence $P\ (C \triangle f\ C)$ **by**(*simp add*: *P_f P_narrow*)
moreover then have $f\ (C \triangle f\ C) \sqsubseteq C \triangle f\ C$
by (*metis narrow1 narrow2 1 mono preord_class.le_trans*)
moreover have $n\ (C \triangle f\ C) < n\ C$ **using** *1 2* **by**(*simp add*: *n_narrow P_f*)
ultimately show $(P\ (C \triangle f\ C) \wedge f\ (C \triangle f\ C) \sqsubseteq C \triangle f\ C) \wedge n(C \triangle f\ C) < n\ C$
by *blast*
qed

locale *Abs_Int2_measure* =
Abs_Int2 **where** $\gamma=\gamma$ + *Measure_WN* **where** $m=m$
for $\gamma :: 'av::\{WN,lattice\} \Rightarrow val\ set$ **and** $m :: 'av \Rightarrow nat$

12.12.4 Termination: Intervals

definition $m\_ivl :: ivl \Rightarrow nat$ **where**
$m\_ivl\ ivl = (case\ ivl\ of\ Ivl\ l\ h \Rightarrow$
$(case\ l\ of\ None \Rightarrow 0 \mid Some\ \_ \Rightarrow 1) + (case\ h\ of\ None \Rightarrow 0 \mid Some\ \_ \Rightarrow 1))$

lemma *m_ivl_height*: $m\_ivl\ ivl \le 2$

by(*simp add*: *m_ivl_def split*: *ivl.split option.split*)

lemma *m_ivl_anti_mono*: $(y::ivl) \sqsubseteq x \Longrightarrow m\_ivl\ x \le m\_ivl\ y$
by(*auto simp*: *m_ivl_def le_option_def le_ivl_def*
    split: *ivl.split option.split if_splits*)

lemma *m_ivl_widen*:
  $\sim y \sqsubseteq x \Longrightarrow m\_ivl(x \nabla y) < m\_ivl\ x$
by(*auto simp*: *m_ivl_def widen_ivl_def le_option_def le_ivl_def*
    split: *ivl.splits option.splits if_splits*)

definition *n_ivl* :: *ivl* ⇒ *nat* **where**
$n\_ivl\ ivl = 2 - m\_ivl\ ivl$

lemma *n_ivl_mono*: $(x::ivl) \sqsubseteq y \Longrightarrow n\_ivl\ x \le n\_ivl\ y$
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*)

interpretation *Abs_Int2_measure*
where $\gamma = \gamma\_ivl$ **and** $num' = num\_ivl$ **and** $plus' = plus\_ivl$
and $test\_num' = in\_ivl$
and $filter\_plus' = filter\_plus\_ivl$ **and** $filter\_less' = filter\_less\_ivl$
and $m = m\_ivl$ **and** $n = n\_ivl$ **and** $h = 2$
proof
  case *goal1* **thus** *?case* **by**(*rule m_ivl_anti_mono*)
next
  case *goal2* **thus** *?case* **by**(*rule m_ivl_height*)
next
  case *goal3* **thus** *?case* **by**(*rule m_ivl_widen*)
next
  case *goal4* **thus** *?case* **by**(*rule n_ivl_mono*)
next
  case *goal5* **from** *goal5*(*2*) **show** *?case* **by**(*rule n_ivl_narrow*)
  — note that the first assms is unnecessary for intervals
qed

lemma *iter_winden_step_ivl_termination*:
  $\exists C.\ iter\_widen\ (step\_ivl\ (top(vars\ c)))\ (bot\ c) = Some\ C$

apply(*rule iter_widen_termination*[**where** $m = m\_c$ **and** $P = \%C.\ C \in Lc\ c$])
apply (*simp_all add*: *step'_in_Lc m_c_widen*)
done

lemma *iter_narrow_step_ivl_termination*:
$C0 \in Lc\ c \Longrightarrow step\_ivl\ (top(vars\ c))\ C0 \sqsubseteq C0 \Longrightarrow$
$\exists\, C.\ iter\_narrow\ (step\_ivl\ (top(vars\ c)))\ C0 = Some\ C$
apply(*rule iter_narrow_termination*[**where** $n = n\_c$ **and** $P = \%C.\ C \in Lc\ c$])
apply (*simp add*: *step'_in_Lc*)
apply (*simp*)
apply(*rule mono_step'_top*)
apply(*simp add*: *Lc_acom_def L_acom_def*)
apply(*simp add*: *Lc_acom_def L_acom_def*)
apply *assumption*
apply(*erule* (*3*) *n_c_narrow*)
apply *assumption*
apply *assumption*
done

theorem *AI_ivl'_termination*:
$\exists\, C.\ AI\_ivl'\ c = Some\ C$
apply(*auto simp*: *AI_wn_def pfp_wn_def iter_winden_step_ivl_termination*
split: *option.split*)
apply(*rule iter_narrow_step_ivl_termination*)
apply(*blast intro*: *iter_widen_inv*[**where** $f = step'\ \top_{vars\ c}$ **and** $P = \%C.\ C \in Lc\ c$] *bot_in_Lc Lc_widen step'_in_Lc*[**where** $S = \top_{vars\ c}$ **and** $c{=}c$, *simplified*])
apply(*erule iter_widen_pfp*)
done

12.12.5 Counterexamples

Widening is increasing by assumption, but $x \sqsubseteq f\ x$ is not an invariant of widening. It can already be lost after the first step:

lemma assumes $!!x\ y{::}'a{::}WN.\ x \sqsubseteq y \Longrightarrow f\ x \sqsubseteq f\ y$
and $x \sqsubseteq f\ x$ **and** $\neg\ f\ x \sqsubseteq x$ **shows** $x\ \nabla\ f\ x \sqsubseteq f(x\ \nabla\ f\ x)$
nitpick[*card* = *3*, *expect* = *genuine*, *show_consts*]

oops

Widening terminates but may converge more slowly than Kleene iteration. In the following model, Kleene iteration goes from 0 to the least pfp in one step but widening takes 2 steps to reach a strictly larger pfp:

lemma assumes $!!x\ y::'a::WN.\ x \sqsubseteq y \Longrightarrow f\ x \sqsubseteq f\ y$
and $x \sqsubseteq f\ x$ **and** $\neg\ f\ x \sqsubseteq x$ **and** $f(f\ x) \sqsubseteq f\ x$
shows $f(x\ \nabla\ f\ x) \sqsubseteq x\ \nabla\ f\ x$
nitpick[*card* = *4*, *expect* = *genuine*, *show_consts*]

oops

end

13 Extensions and Variations of IMP

theory *Procs* **imports** *BExp* **begin**

13.1 Procedures and Local Variables

type_synonym *pname* = *string*

```
datatype
  com = SKIP
      | Assign vname aexp      (_ ::= _ [1000, 61] 61)
      | Seq    com  com        (_;/ _  [60, 61] 60)
      | If     bexp com com    ((IF _/ THEN _/ ELSE _)  [0, 0, 61] 61)
      | While  bexp com        ((WHILE _/ DO _)  [0, 61] 61)
      | Var    vname com       ((1{VAR _;;/ _}))
      | Proc   pname com com   ((1{PROC _ = _;;/ _}))
      | CALL   pname
```

```
definition test_com =
{VAR ''x'';;
 {PROC ''p'' = ''x'' ::= N 1;;
  {PROC ''q'' = CALL ''p'';;
   {VAR ''x'';;
    ''x'' ::= N 2;
    {PROC ''p'' = ''x'' ::= N 3;;
     CALL ''q''; ''y'' ::= V ''x''}}}}}
```

end

theory *Procs_Dyn_Vars_Dyn* **imports** *Procs*
begin

13.1.1 Dynamic Scoping of Procedures and Variables

type_synonym *penv* = *pname* ⇒ *com*

inductive
  big_step :: *penv* ⇒ *com* × *state* ⇒ *state* ⇒ *bool* (_ ⊢ _ ⇒ _ [60,0,60] 55)
where
Skip: $pe \vdash (SKIP,s) \Rightarrow s$ |
Assign: $pe \vdash (x ::= a,s) \Rightarrow s(x := aval\ a\ s)$ |
Seq: $[\![\ pe \vdash (c_1,s_1) \Rightarrow s_2;\ pe \vdash (c_2,s_2) \Rightarrow s_3\ ]\!] \Longrightarrow$
$pe \vdash (c_1;c_2,\ s_1) \Rightarrow s_3$ |

IfTrue: $[\![\ bval\ b\ s;\ pe \vdash (c_1,s) \Rightarrow t\ ]\!] \Longrightarrow$
$pe \vdash (IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \Rightarrow t$ |
IfFalse: $[\![\ \neg bval\ b\ s;\ pe \vdash (c_2,s) \Rightarrow t\ ]\!] \Longrightarrow$
$pe \vdash (IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \Rightarrow t$ |

WhileFalse: $\neg bval\ b\ s \Longrightarrow pe \vdash (WHILE\ b\ DO\ c,s) \Rightarrow s$ |
WhileTrue:
$[\![\ bval\ b\ s_1;\ pe \vdash (c,s_1) \Rightarrow s_2;\ pe \vdash (WHILE\ b\ DO\ c,\ s_2) \Rightarrow s_3\ ]\!] \Longrightarrow$
$pe \vdash (WHILE\ b\ DO\ c,\ s_1) \Rightarrow s_3$ |

Var: $pe \vdash (c,s) \Rightarrow t \Longrightarrow pe \vdash (\{VAR\ x;;\ c\},\ s) \Rightarrow t(x := s\ x)$ |

Call: $pe \vdash (pe\ p,\ s) \Rightarrow t \Longrightarrow pe \vdash (CALL\ p,\ s) \Rightarrow t$ |

Proc: $pe(p := cp) \vdash (c,s) \Rightarrow t \Longrightarrow pe \vdash (\{PROC\ p = cp;;\ c\},\ s) \Rightarrow t$

code_pred *big_step* **.**

values $\{map\ t\ [''x'',''y'']\ |t.\ (\lambda p.\ SKIP) \vdash (test\_com,\ <>) \Rightarrow t\}$

end

theory *Procs_Stat_Vars_Dyn* **imports** *Procs*
begin

13.1.2 Static Scoping of Procedures, Dynamic of Variables

type_synonym *penv* = (*pname* × *com*) *list*

inductive
  big_step :: *penv* ⇒ *com* × *state* ⇒ *state* ⇒ *bool* (_ ⊢ _ ⇒ _ [60,0,60] 55)
where

Skip: $pe \vdash (SKIP,s) \Rightarrow s$ |
Assign: $pe \vdash (x ::= a,s) \Rightarrow s(x := aval\ a\ s)$ |
Seq: $[\![\ pe \vdash (c_1,s_1) \Rightarrow s_2;\ \ pe \vdash (c_2,s_2) \Rightarrow s_3\ ]\!] \Longrightarrow$
$pe \vdash (c_1;c_2,\ s_1) \Rightarrow s_3$ |

IfTrue: $[\![\ bval\ b\ s;\ \ pe \vdash (c_1,s) \Rightarrow t\ ]\!] \Longrightarrow$
$pe \vdash (IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \Rightarrow t$ |
IfFalse: $[\![\ \neg bval\ b\ s;\ \ pe \vdash (c_2,s) \Rightarrow t\ ]\!] \Longrightarrow$
$pe \vdash (IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \Rightarrow t$ |

WhileFalse: $\neg bval\ b\ s \Longrightarrow pe \vdash (WHILE\ b\ DO\ c,s) \Rightarrow s$ |
WhileTrue:
$[\![\ bval\ b\ s_1;\ \ pe \vdash (c,s_1) \Rightarrow s_2;\ \ pe \vdash (WHILE\ b\ DO\ c,\ s_2) \Rightarrow s_3\ ]\!] \Longrightarrow$
$pe \vdash (WHILE\ b\ DO\ c,\ s_1) \Rightarrow s_3$ |

Var: $pe \vdash (c,s) \Rightarrow t \Longrightarrow pe \vdash (\{VAR\ x;;\ c\},\ s) \Rightarrow t(x := s\ x)$ |

Call1: $(p,c)\#pe \vdash (c,\ s) \Rightarrow t \Longrightarrow (p,c)\#pe \vdash (CALL\ p,\ s) \Rightarrow t$ |
Call2: $[\![\ p' \neq p;\ \ pe \vdash (CALL\ p,\ s) \Rightarrow t\ ]\!] \Longrightarrow$
$(p',c)\#pe \vdash (CALL\ p,\ s) \Rightarrow t$ |

Proc: $(p,cp)\#pe \vdash (c,s) \Rightarrow t \Longrightarrow pe \vdash (\{PROC\ p = cp;;\ c\},\ s) \Rightarrow t$

code_pred *big_step* **.**

values $\{map\ t\ [''x'', ''y'']\ |t.\ [] \vdash (test\_com, <>) \Rightarrow t\}$

end

theory *Procs_Stat_Vars_Stat* **imports** *Procs*
begin

13.1.3 Static Scoping of Procedures and Variables

type_synonym $addr = nat$
type_synonym $venv = vname \Rightarrow addr$
type_synonym $store = addr \Rightarrow val$
type_synonym $penv = (pname \times com \times venv)\ list$

fun $venv :: penv \times venv \times nat \Rightarrow venv$ **where**
$venv(\_,ve,\_) = ve$

inductive
$big\_step :: penv \times venv \times nat \Rightarrow com \times store \Rightarrow store \Rightarrow bool$

$(\_ \vdash \_ \Rightarrow \_\ [60,0,60]\ 55)$
where
Skip: $e \vdash (SKIP,s) \Rightarrow s$ |
Assign: $(pe,ve,f) \vdash (x ::= a,s) \Rightarrow s(ve\ x := aval\ a\ (s\ o\ ve))$ |
Seq: $[\![\ e \vdash (c_1,s_1) \Rightarrow s_2;\ \ e \vdash (c_2,s_2) \Rightarrow s_3\ ]\!] \Longrightarrow$
$e \vdash (c_1;c_2,\ s_1) \Rightarrow s_3$ |

IfTrue: $[\![\ bval\ b\ (s \circ venv\ e);\ \ e \vdash (c_1,s) \Rightarrow t\ ]\!] \Longrightarrow$
$e \vdash (IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \Rightarrow t$ |
IfFalse: $[\![\ \neg bval\ b\ (s \circ venv\ e);\ \ e \vdash (c_2,s) \Rightarrow t\ ]\!] \Longrightarrow$
$e \vdash (IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \Rightarrow t$ |

WhileFalse: $\neg bval\ b\ (s \circ venv\ e) \Longrightarrow e \vdash (WHILE\ b\ DO\ c,s) \Rightarrow s$ |
WhileTrue:
$[\![\ bval\ b\ (s_1 \circ venv\ e);\ \ e \vdash (c,s_1) \Rightarrow s_2;$
$e \vdash (WHILE\ b\ DO\ c,\ s_2) \Rightarrow s_3\ ]\!] \Longrightarrow$
$e \vdash (WHILE\ b\ DO\ c,\ s_1) \Rightarrow s_3$ |

Var: $(pe,ve(x{:=}f),f{+}1) \vdash (c,s) \Rightarrow t \ \Longrightarrow$
$(pe,ve,f) \vdash (\{VAR\ x;;\ c\},\ s) \Rightarrow t$ |

Call1: $((p,c,ve)\#pe,ve,f) \vdash (c,\ s) \Rightarrow t \ \Longrightarrow$
$((p,c,ve)\#pe,ve',f) \vdash (CALL\ p,\ s) \Rightarrow t$ |
Call2: $[\![\ p' \neq p;\ \ (pe,ve,f) \vdash (CALL\ p,\ s) \Rightarrow t\ ]\!] \Longrightarrow$
$((p',c,ve')\#pe,ve,f) \vdash (CALL\ p,\ s) \Rightarrow t$ |

Proc: $((p,cp,ve)\#pe,ve,f) \vdash (c,s) \Rightarrow t$
$\Longrightarrow\ (pe,ve,f) \vdash (\{PROC\ p = cp;;\ c\},\ s) \Rightarrow t$

code_pred *big_step* **.**

values $\{map\ t\ [10,11]\ |t.$
$([],\ <''x'' := 10,\ ''y'' := 11>,\ 12)$
$\vdash (test\_com,\ <>) \Rightarrow t\}$

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.