Concrete Semantics

Tobias Nipkow & Gerwin Klein

August 15, 2018

Abstract

This document presents formalizations of the semantics of a simple imperative programming language together with a number of applications: a compiler, type systems, various program analyses and abstract interpreters. These theories form the basis of the book *Concrete Semantics with Isabelle/HOL* by Nipkow and Klein [2].

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* a_1 a_2) *s* = *aval* a_1 *s* + *aval* a_2 *s*

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

The same state more concisely:

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

A little syntax magic to write larger states compactly:

definition *null_state* (<>) **where**
null_state ≡ λ*x*. *0*
syntax
State :: *updbinds* => *'a* (<>)
translations
_State ms == *_Update* <> *ms*
_State (*_updbinds b bs*) <= *_Update* (*_State b*) *bs*

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

lemma <*a* := *1*, *b* := *2*> = (<> (*a* := *1*)) (*b* := (*2*::*int*))
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 a₁ a₂) =
  (case (asimp_const a₁, asimp_const a₂) of
    (N n₁, N n₂) ⇒ N(n₁+n₂) |
    (b₁,b₂) ⇒ Plus b₁ b₂)
```

```
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 i₁) (N i₂) = N(i₁+i₂) |
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 a₁ a₂ = Plus a₁ a₂
```

```
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 a₁ a₂) = plus (asimp a₁) (asimp a₂)
```

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**
$bval\ (Bc\ v)\ s = v$ |
$bval\ (Not\ b)\ s = (\neg\ bval\ b\ s)$ |
$bval\ (And\ b_1\ b_2)\ s = (bval\ b_1\ s \wedge bval\ b_2\ s)$ |
$bval\ (Less\ a_1\ a_2)\ s = (aval\ a_1\ s < aval\ a_2\ s)$

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

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*

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* _ (*j* # *i* # *stk*) = (*i* + *j*) # *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 e1 e2) = comp e1 @ comp e2 @ [ADD]

value comp (Plus (Plus (V ''x'') (N 1)) (V ''z''))

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

end
theory Star imports Main
begin

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

hide_fact (open) refl step  — names too generic

lemma star_trans:
  star r x y ⟹ star r y z ⟹ star r x z
proof(induction rule: star.induct)
  case refl thus ?case .
next
```

case *step* **thus** *?case* **by** (*metis star.step*)
qed

lemmas *star_induct* =
star.induct[*of r*:: $'a*'b \Rightarrow 'a*'b \Rightarrow$ *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

The big-step semantics is a straight-forward inductive definition with concrete syntax. Note that the first parameter is a tuple, so the syntax becomes $(c,s) \Rightarrow s'$.

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_goal *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*, λ_. *0*) ⇒ *t*}

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

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

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

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

Proof automation:

The introduction rules are good for automatically construction small program executions. The recursive cases may require backtracking, so we declare the set as unsafe intro rules.

declare *big_step.intros* [*intro*]

The standard induction rule

$[\![$*x1* ⇒ *x2*; $\bigwedge$*s*. *P* (*SKIP*, *s*) *s*; $\bigwedge$*x a s*. *P* (*x* ::= *a*, *s*) (*s*(*x* := *aval a s*));
$\bigwedge c_1\ s_1\ s_2\ c_2\ s_3$.
 $[\![(c_1, s_1) \Rightarrow s_2;\ P\ (c_1, s_1)\ s_2;\ (c_2, s_2) \Rightarrow s_3;\ P\ (c_2, s_2)\ s_3]\!]$
 $\Longrightarrow P\ (c_1;;\ c_2, s_1)\ s_3$;
$\bigwedge b\ s\ c_1\ t\ c_2$.
 $[\![$*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*;

$\bigwedge b\ s\ c_2\ t\ c_1.$
$[\![\neg\ \mathit{bval}\ b\ s;\ (c_2,\ s) \Rightarrow t;\ P\ (c_2,\ s)\ t]\!] \Longrightarrow P\ (\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2,\ s)\ t;$
$\bigwedge b\ s\ c.\ \neg\ \mathit{bval}\ b\ s \Longrightarrow P\ (\mathit{WHILE}\ b\ \mathit{DO}\ c,\ s)\ s;$
$\bigwedge b\ s_1\ c\ s_2\ s_3.$
$[\![\mathit{bval}\ b\ s_1;\ (c,\ s_1) \Rightarrow s_2;\ P\ (c,\ s_1)\ s_2;\ (\mathit{WHILE}\ b\ \mathit{DO}\ c,\ s_2) \Rightarrow s_3;$
$P\ (\mathit{WHILE}\ b\ \mathit{DO}\ c,\ s_2)\ s_3]\!]$
$\Longrightarrow P\ (\mathit{WHILE}\ b\ \mathit{DO}\ c,\ s_1)\ s_3]\!]$
$\Longrightarrow P\ \mathit{x1}\ \mathit{x2}$

thm *big_step.induct*

This induction schema is almost perfect for our purposes, but our trick for reusing the tuple syntax means that the induction schema has two parameters instead of the c, s, and s' that we are likely to encounter. Splitting the tuple parameter fixes this:

lemmas *big_step_induct* = *big_step.induct*[*split_format*(*complete*)]
thm *big_step_induct*

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

3.2 Rule inversion

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

inductive_cases *SkipE*[*elim*!]: $(\mathit{SKIP},s) \Rightarrow t$
thm *SkipE*

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

Similarly for the other commands:

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

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

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

An automatic example:

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

Rule inversion by hand via the "cases" method:

lemma assumes $(IF\ b\ THEN\ SKIP\ ELSE\ SKIP,\ s) \Rightarrow t$
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*:
  $(x ::= a,s) \Rightarrow s' \longleftrightarrow (s' = s(x := aval\ a\ s))$
  by *auto*

An example combining rule inversion and derivations

lemma *Seq_assoc*:
  $(c1;;\ c2;;\ c3,\ s) \Rightarrow s' \longleftrightarrow (c1;;\ (c2;;\ c3),\ s) \Rightarrow s'$
proof
  assume $(c1;;\ c2;;\ c3,\ s) \Rightarrow s'$
  then obtain *s1 s2* **where**

c1: (*c1*, *s*) ⇒ *s1* **and**
c2: (*c2*, *s1*) ⇒ *s2* **and**
c3: (*c3*, *s2*) ⇒ *s′* **by** *auto*
from *c2 c3*
have (*c2*;; *c3*, *s1*) ⇒ *s′* **by** (*rule Seq*)
with *c1*
show (*c1*;; (*c2*;; *c3*), *s*) ⇒ *s′* **by** (*rule Seq*)
next
— The other direction is analogous
assume (*c1*;; (*c2*;; *c3*), *s*) ⇒ *s′*
thus (*c1*;; *c2*;; *c3*, *s*) ⇒ *s′* **by** *auto*
qed

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**
$c \sim c' \equiv (\forall s\ t.\ (c,s) \Rightarrow t\ =\ (c',s) \Rightarrow t)$

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
have (*?iw*, *s*) ⇒ *t* **if** *assm*: (*?w*, *s*) ⇒ *t* **for** *s t*
proof –
from *assm* **show** *?thesis*
proof *cases* — rule inversion on (*?w*, *s*) ⇒ *t*
case *WhileFalse*
thus *?thesis* **by** *blast*
next
case *WhileTrue*
from ‹*bval b s*› ‹(*?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 $(c;;\ ?w,\ s) \Rightarrow t$ **by** (*rule Seq*)
    — then the whole *IF*
    with ‹*bval b s*› **show** *?thesis* **by** (*rule IfTrue*)
  qed
qed
moreover
— now the other direction:
have $(?w,\ s) \Rightarrow t$ **if** *assm*: $(?iw,\ s) \Rightarrow t$ **for** *s t*
proof −
  from *assm* **show** *?thesis*
  proof *cases* — rule inversion on $(?iw,\ s) \Rightarrow t$
    case *IfFalse*
    hence $s = t$ **using** ‹$(?iw,\ s) \Rightarrow t$› **by** *blast*
    thus *?thesis* **using** ‹$\neg$*bval b s*› **by** *blast*
  next
    case *IfTrue*
    — and for this, only the Seq-rule is applicable:
    from ‹$(c;;\ ?w,\ s) \Rightarrow t$› **obtain** s' **where**
      $(c,\ s) \Rightarrow s'$ **and** $(?w,\ s') \Rightarrow t$ **by** *auto*
    — with this information, we can build a derivation tree for *WHILE*
    with ‹*bval b s*› **show** *?thesis* **by** (*rule WhileTrue*)
  qed
qed
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*

lemma *sim_while_cong_aux*:
 $(\mathit{WHILE}\ b\ \mathit{DO}\ c,s) \Rightarrow t \implies c \sim c' \implies (\mathit{WHILE}\ b\ \mathit{DO}\ c',s) \Rightarrow t$
apply(*induction WHILE b DO c s t arbitrary*: *b c rule*: *big_step_induct*)
 apply *blast*
apply *blast*
done

lemma *sim_while_cong*: $c \sim c' \implies \mathit{WHILE}\ b\ \mathit{DO}\ c \sim \mathit{WHILE}\ b\ \mathit{DO}\ c'$
by (*metis sim_while_cong_aux*)

Command equivalence is an equivalence relation, i.e. it is reflexive, symmetric, and transitive. Because we used an abbreviation above, Isabelle derives this automatically.

lemma *sim_refl*: $c \sim c$ **by** *simp*
lemma *sim_sym*: $(c \sim c') = (c' \sim c)$ **by** *auto*
lemma *sim_trans*: $c \sim c' \implies c' \sim c'' \implies c \sim c''$ **by** *auto*

3.4 Execution is deterministic

This proof is automatic.

theorem *big_step_determ*: $[\![(c,s) \Rightarrow t;\ (c,s) \Rightarrow u]\!] \implies u = t$
 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 $b\ c\ s\ s_1\ t\ t'$
 — The assumptions of the rule:
 assume *bval b s* **and** $(c,s) \Rightarrow s_1$ **and** $(\mathit{WHILE}\ b\ \mathit{DO}\ c,s_1) \Rightarrow t$
 — Ind.Hyp; note the $\bigwedge$ because of arbitrary:
 assume *IHc*: $\bigwedge t'.\ (c,s) \Rightarrow t' \implies t' = s_1$
 assume *IHw*: $\bigwedge t'.\ (\mathit{WHILE}\ b\ \mathit{DO}\ c,s_1) \Rightarrow t' \implies t' = t$
 — Premise of implication:
 assume $(\mathit{WHILE}\ b\ \mathit{DO}\ c,s) \Rightarrow t'$
 with ‹*bval b s*› **obtain** ${s_1}'$ **where**
  c: $(c,s) \Rightarrow {s_1}'$ **and**
  w: $(\mathit{WHILE}\ b\ \mathit{DO}\ c,{s_1}') \Rightarrow t'$
  by *auto*
 from *c IHc* **have** ${s_1}' = s_1$ **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* ⇒ *com* ∗ *state* ⇒ *bool* (**infix** → *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: $\neg bval\ b\ s \Longrightarrow (IF\ b\ THEN\ c_1\ ELSE\ c_2,s) \rightarrow (c_2,s)$ |

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

abbreviation
  small_steps :: *com* ∗ *state* ⇒ *com* ∗ *state* ⇒ *bool* (**infix** →∗ *55*)
where $x \rightarrow\!*\ y == star\ small\_step\ x\ y$

4.2 Executability

code_pred *small_step* **.**

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

4.3 Proof infrastructure

4.3.1 Induction rules

The default induction rule *small_step.induct* only works for lemmas of the form $a \rightarrow b \Longrightarrow \ldots$ where a and b are not already pairs $(DUMMY,DUMMY)$. We can generate a suitable variant of *small_step.induct* for pairs by "splitting" the arguments → 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) → ct
thm SkipE
inductive_cases AssignE[elim!]: (x::=a,s) → ct
thm AssignE
inductive_cases SeqE[elim]: (c1;;c2,s) → ct
thm SeqE
inductive_cases IfE[elim!]: (IF b THEN c1 ELSE c2,s) → ct
inductive_cases WhileE[elim]: (WHILE b DO c, s) → ct
```

A simple property:

```
lemma deterministic:
  cs → cs' ⟹ cs → cs'' ⟹ 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*:
  $cs \to cs' \Longrightarrow cs' \Rightarrow t \Longrightarrow cs \Rightarrow t$
apply (*induction arbitrary*: *t rule*: *small_step.induct*)
apply *auto*
done

lemma *small_to_big*:
  $cs \to\!* (SKIP,t) \Longrightarrow cs \Rightarrow t$
apply (*induction cs* (*SKIP*,*t*) *rule*: *star.induct*)
apply (*auto intro*: *small1_big_continue*)
done

Finally, the equivalence theorem:

theorem *big_iff_small*:
  $cs \Rightarrow t = cs \to\!* (SKIP,t)$
by(*metis big_to_small small_to_big*)

4.5 Final configurations and infinite reductions

definition *final cs* $\longleftrightarrow \neg(\exists\, cs'.\ cs \to cs')$

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*:
$(\exists\, t.\ cs \Rightarrow t) \longleftrightarrow (\exists\, cs'.\ cs \rightarrow{*}\ cs' \wedge \mathit{final}\ cs')$
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
theory *Finite_Reachable*
imports *Small_Step*
begin

4.6 Finite number of reachable commands

This theory shows that in the small-step semantics one can only reach a finite number of commands from any given command. Hence one can see the command component of a small-step configuration as a combination of the program to be executed and a pc.

definition *reachable* :: *com* $\Rightarrow$ *com set* **where**
reachable $c = \{c'.\ \exists\, s\ t.\ (c,s) \rightarrow{*}\ (c',t)\}$

Proofs need induction on the length of a small-step reduction sequence.

fun *small_stepsn* :: *com* $*$ *state* $\Rightarrow$ *nat* $\Rightarrow$ *com* $*$ *state* $\Rightarrow$ *bool*
($\_ \rightarrow'(\_') \_$ [55,0,55] 55) **where**
$(cs \rightarrow(0)\ cs') = (cs' = cs)$ |
$cs \rightarrow(Suc\ n)\ cs'' = (\exists\, cs'.\ cs \rightarrow cs' \wedge cs' \rightarrow(n)\ cs'')$

lemma *stepsn_if_star*: $cs \rightarrow{*}\ cs' \Longrightarrow \exists\, n.\ cs \rightarrow(n)\ cs'$
proof(*induction rule*: *star.induct*)
  case *refl* **show** *?case* **by** (*metis small_stepsn.simps*(*1*))
next
  case *step* **thus** *?case* **by** (*metis small_stepsn.simps*(*2*))
qed

lemma *star_if_stepsn*: $cs \rightarrow(n)\ cs' \Longrightarrow cs \rightarrow{*}\ cs'$
by(*induction n arbitrary*: *cs*) (*auto elim*: *star.step*)

lemma *SKIP_starD*: $(SKIP,\ s) \rightarrow{*}\ (c,t) \Longrightarrow c = SKIP$
by(*induction SKIP s c t rule*: *star_induct*) *auto*

lemma *reachable_SKIP*: *reachable SKIP* = {*SKIP*}
by(*auto simp*: *reachable_def dest*: *SKIP_starD*)

lemma *Assign_starD*: (*x*::=*a*, *s*) →∗ (*c*,*t*) ⟹ *c* ∈ {*x*::=*a*, *SKIP*}
by (*induction x*::=*a s c t rule*: *star_induct*) (*auto dest*: *SKIP_starD*)

lemma *reachable_Assign*: *reachable* (*x*::=*a*) = {*x*::=*a*, *SKIP*}
by(*auto simp*: *reachable_def dest*:*Assign_starD*)

lemma *Seq_stepsnD*: (*c1*;; *c2*, *s*) →(*n*) (*c*′, *t*) ⟹
(∃ *c1*′ *m*. *c*′ = *c1*′;; *c2* ∧ (*c1*, *s*) →(*m*) (*c1*′, *t*) ∧ *m* ≤ *n*) ∨
(∃ *s2 m1 m2*. (*c1*,*s*) →(*m1*) (*SKIP*,*s2*) ∧ (*c2*, *s2*) →(*m2*) (*c*′, *t*) ∧
m1+*m2* < *n*)
proof(*induction n arbitrary*: *c1 c2 s*)
case *0* **thus** *?case* **by** *auto*
next
case (*Suc n*)
from *Suc.prems* **obtain** *s*′ *c12*′ **where** (*c1*;;*c2*, *s*) → (*c12*′, *s*′)
and *n*: (*c12*′,*s*′) →(*n*) (*c*′,*t*) **by** *auto*
from *this*(*1*) **show** *?case*
proof
assume *c1* = *SKIP* (*c12*′, *s*′) = (*c2*, *s*)
hence (*c1*,*s*) →(*0*) (*SKIP*, *s*′) ∧ (*c2*, *s*′) →(*n*) (*c*′, *t*) ∧ *0* + *n* < *Suc n*
using *n* **by** *auto*
thus *?case* **by** *blast*
next
fix *c1*′ *s*′′ **assume** *1*: (*c12*′, *s*′) = (*c1*′;; *c2*, *s*′′) (*c1*, *s*) → (*c1*′, *s*′′)
hence *n*′: (*c1*′;;*c2*,*s*′) →(*n*) (*c*′,*t*) **using** *n* **by** *auto*
from *Suc.IH*[*OF n*′] **show** *?case*
proof
assume ∃ *c1*′′ *m*. *c*′ = *c1*′′;; *c2* ∧ (*c1*′, *s*′) →(*m*) (*c1*′′, *t*) ∧ *m* ≤ *n*
(**is** ∃ *a b*. *?P a b*)
then obtain *c1*′′ *m* **where** *2*: *?P c1*′′ *m* **by** *blast*
hence *c*′ = *c1*′′;;*c2* ∧ (*c1*, *s*) →(*Suc m*) (*c1*′′,*t*) ∧ *Suc m* ≤ *Suc n*
using *1* **by** *auto*
thus *?case* **by** *blast*
next
assume ∃ *s2 m1 m2*. (*c1*′,*s*′) →(*m1*) (*SKIP*,*s2*) ∧
(*c2*,*s2*) →(*m2*) (*c*′,*t*) ∧ *m1*+*m2* < *n* (**is** ∃ *a b c*. *?P a b c*)
then obtain *s2 m1 m2* **where** *?P s2 m1 m2* **by** *blast*
hence (*c1*,*s*) →(*Suc m1*) (*SKIP*,*s2*) ∧ (*c2*,*s2*) →(*m2*) (*c*′,*t*) ∧
Suc m1 + *m2* < *Suc n* **using** *1* **by** *auto*
thus *?case* **by** *blast*
qed

qed
qed

corollary *Seq_starD*: $(c1;;\ c2,\ s) \to* (c',\ t) \Longrightarrow$
$(\exists\, c1'.\ c' = c1';;\ c2 \wedge (c1,\ s) \to* (c1',\ t)) \vee$
$(\exists\, s2.\ (c1,s) \to* (SKIP,s2) \wedge (c2,\ s2) \to* (c',\ t))$
by(*metis Seq_stepsnD star_if_stepsn stepsn_if_star*)

lemma *reachable_Seq*: *reachable* $(c1;;c2) \subseteq$
$(\lambda c1'.\ c1';;c2)$ ' *reachable c1* $\cup$ *reachable c2*
by(*auto simp*: *reachable_def image_def dest*!: *Seq_starD*)

lemma *If_starD*: $(IF\ b\ THEN\ c1\ ELSE\ c2,\ s) \to* (c,t) \Longrightarrow$
$c = IF\ b\ THEN\ c1\ ELSE\ c2 \vee (c1,s) \to* (c,t) \vee (c2,s) \to* (c,t)$
by(*induction IF b THEN c1 ELSE c2 s c t rule*: *star_induct*) *auto*

lemma *reachable_If*: *reachable* $(IF\ b\ THEN\ c1\ ELSE\ c2) \subseteq$
$\{IF\ b\ THEN\ c1\ ELSE\ c2\} \cup$ *reachable c1* $\cup$ *reachable c2*
by(*auto simp*: *reachable_def dest*!: *If_starD*)

lemma *While_stepsnD*: $(WHILE\ b\ DO\ c,\ s) \to(n)\ (c2,t) \Longrightarrow$
$c2 \in \{WHILE\ b\ DO\ c,\ IF\ b\ THEN\ c\ ;;\ WHILE\ b\ DO\ c\ ELSE\ SKIP,$
$SKIP\}$
$\vee (\exists\, c1.\ c2 = c1\ ;;\ WHILE\ b\ DO\ c \wedge (\exists\ s1\ s2.\ (c,s1) \to* (c1,s2)))$
proof(*induction n arbitrary*: *s rule*: *less_induct*)
case (*less n1*)
show *?case*
proof(*cases n1*)
case *0* **thus** *?thesis* **using** *less.prems* **by** (*simp*)
next
case (*Suc n2*)
let $?w = WHILE\ b\ DO\ c$
let $?iw = IF\ b\ THEN\ c\ ;;\ ?w\ ELSE\ SKIP$
from *Suc less.prems* **have** *n2*: $(?iw,s) \to(n2)\ (c2,t)$ **by**(*auto elim*!:
WhileE)
show *?thesis*
proof(*cases n2*)
case *0* **thus** *?thesis* **using** *n2* **by** *auto*
next
case (*Suc n3*)
then obtain $iw'\ s'$ **where** $(?iw,s) \to (iw',s')$
and *n3*: $(iw',s') \to(n3)\ (c2,t)$ **using** *n2* **by** *auto*

```
      from this(1)
      show ?thesis
      proof
        assume (iw', s') = (c;; WHILE b DO c, s)
        with n3 have (c;;?w, s) →(n3) (c2,t) by auto
        from Seq_stepsnD[OF this] show ?thesis
        proof
          assume ∃ c1' m. c2 = c1';; ?w ∧ (c,s) →(m) (c1', t) ∧ m ≤ n3
          thus ?thesis by (metis star_if_stepsn)
        next
          assume ∃ s2 m1 m2. (c, s) →(m1) (SKIP, s2) ∧
            (WHILE b DO c, s2) →(m2) (c2, t) ∧ m1 + m2 < n3 (is ∃ x y
z. ?P x y z)
          then obtain s2 m1 m2 where ?P s2 m1 m2 by blast
          with ⟨n2 = Suc n3⟩ ⟨n1 = Suc n2⟩have m2 < n1 by arith
          from less.IH[OF this] ⟨?P s2 m1 m2⟩ show ?thesis by blast
        qed
      next
        assume (iw', s') = (SKIP, s)
       thus ?thesis using star_if_stepsn[OF n3] by(auto dest!: SKIP_starD)
      qed
    qed
  qed
qed

lemma reachable_While: reachable (WHILE b DO c) ⊆
  {WHILE b DO c, IF b THEN c ;; WHILE b DO c ELSE SKIP, SKIP} ∪
  (λc'. c' ;; WHILE b DO c) ` reachable c
apply(auto simp: reachable_def image_def)
by (metis While_stepsnD insertE singletonE stepsn_if_star)


theorem finite_reachable: finite(reachable c)
apply(induction c)
apply(auto simp: reachable_SKIP reachable_Assign
  finite_subset[OF reachable_Seq] finite_subset[OF reachable_If]
  finite_subset[OF reachable_While])
done


end
```

5 Denotational Semantics of Commands

theory *Denotational* **imports** *Big_Step* **begin**

type_synonym *com_den* = (*state* × *state*) *set*

definition *W* :: (*state* ⇒ *bool*) ⇒ *com_den* ⇒ (*com_den* ⇒ *com_den*)
where
W db dc = (λ*dw*. {(*s*,*t*). *if db s then* (*s*,*t*) ∈ *dc O dw else s*=*t*})

fun *D* :: *com* ⇒ *com_den* **where**
D SKIP = *Id* |
D (*x* ::= *a*) = {(*s*,*t*). *t* = *s*(*x* := *aval a s*)} |
D (*c1*;;*c2*) = *D*(*c1*) *O D*(*c2*) |
D (*IF b THEN c1 ELSE c2*)
= {(*s*,*t*). *if bval b s then* (*s*,*t*) ∈ *D c1 else* (*s*,*t*) ∈ *D c2*} |
D (*WHILE b DO c*) = *lfp* (*W* (*bval b*) (*D c*))

lemma *W_mono*: *mono* (*W b r*)
by (*unfold W_def mono_def*) *auto*

lemma *D_While_If*:
D(*WHILE b DO c*) = *D*(*IF b THEN c*;;*WHILE b DO c ELSE SKIP*)
proof−
let *?w* = *WHILE b DO c* **let** *?f* = *W* (*bval b*) (*D c*)
have *D ?w* = *lfp ?f* **by** *simp*
also have ... = *?f* (*lfp ?f*) **by**(*rule lfp_unfold* [*OF W_mono*])
also have ... = *D*(*IF b THEN c*;;*?w ELSE SKIP*) **by** (*simp add*: *W_def*)
finally show *?thesis* **.**
qed

Equivalence of denotational and big-step semantics:

lemma *D_if_big_step*: (*c*,*s*) ⇒ *t* ⟹ (*s*,*t*) ∈ *D*(*c*)
proof (*induction rule*: *big_step_induct*)
case *WhileFalse*
with *D_While_If* **show** *?case* **by** *auto*
next
case *WhileTrue*
show *?case* **unfolding** *D_While_If* **using** *WhileTrue* **by** *auto*
qed *auto*

abbreviation *Big_step* :: *com* ⇒ *com_den* **where**
Big_step c ≡ {(*s*,*t*). (*c*,*s*) ⇒ *t*}

lemma *Big_step_if_D*: $(s,t) \in D(c) \Longrightarrow (s,t) \in Big\_step\ c$
proof (*induction c arbitrary*: *s t*)
  case *Seq* **thus** *?case* **by** *fastforce*
next
  case (*While b c*)
  let $?B = Big\_step\ (WHILE\ b\ DO\ c)$ **let** $?f = W\ (bval\ b)\ (D\ c)$
  have $?f\ ?B \subseteq ?B$ **using** *While.IH* **by** (*auto simp*: *W_def*)
  from *lfp_lowerbound*[**where** $?f = ?f$, *OF this*] *While.prems*
  show *?case* **by** *auto*
qed (*auto split*: *if_splits*)

theorem *denotational_is_big_step*:
  $(s,t) \in D(c) \ = \ ((c,s) \Rightarrow t)$
by (*metis D_if_big_step Big_step_if_D*[*simplified*])

corollary *equiv_c_iff_equal_D*: $(c1 \sim c2) \longleftrightarrow D\ c1 = D\ c2$
by(*simp add*: *denotational_is_big_step*[*symmetric*] *set_eq_iff*)

5.1 Continuity

definition *chain* :: $(nat \Rightarrow 'a\ set) \Rightarrow bool$ **where**
$chain\ S = (\forall i.\ S\ i \subseteq S(Suc\ i))$

lemma *chain_total*: $chain\ S \Longrightarrow S\ i \le S\ j \lor S\ j \le S\ i$
by (*metis chain_def le_cases lift_Suc_mono_le*)

definition *cont* :: $('a\ set \Rightarrow 'b\ set) \Rightarrow bool$ **where**
$cont\ f = (\forall S.\ chain\ S \longrightarrow f(UN\ n.\ S\ n) = (UN\ n.\ f(S\ n)))$

lemma *mono_if_cont*: **fixes** $f :: 'a\ set \Rightarrow 'b\ set$
  assumes *cont f* **shows** *mono f*
proof
  fix $a\ b :: 'a\ set$ **assume** $a \subseteq b$
  let $?S = \lambda n{::}nat.\ if\ n{=}0\ then\ a\ else\ b$
  have *chain ?S* **using** $\langle a \subseteq b \rangle$ **by**(*auto simp*: *chain_def*)
  hence $f(UN\ n.\ ?S\ n) = (UN\ n.\ f(?S\ n))$
    using *assms* **by**(*simp add*: *cont_def*)
  moreover have $(UN\ n.\ ?S\ n) = b$ **using** $\langle a \subseteq b \rangle$ **by** (*auto split*: *if_splits*)
  moreover have $(UN\ n.\ f(?S\ n)) = f\ a \cup f\ b$ **by** (*auto split*: *if_splits*)
  ultimately show $f\ a \subseteq f\ b$ **by** (*metis Un_upper1*)
qed

lemma *chain_iterates*: **fixes** $f :: 'a\ set \Rightarrow 'a\ set$
  assumes *mono f* **shows** $chain(\lambda n.\ (f\hat{}\hat{}n)\ \{\})$

```
proof-
  have (f ^^ n) {} ⊆ (f ^^ Suc n) {} for n
  proof (induction n)
    case 0 show ?case by simp
  next
    case (Suc n) thus ?case using assms by (auto simp: mono_def)
  qed
  thus ?thesis by(auto simp: chain_def assms)
qed

theorem lfp_if_cont:
  assumes cont f shows lfp f = (UN n. (f^^n) {}) (is _ = ?U)
proof
  from assms mono_if_cont
  have mono: (f ^^ n) {} ⊆ (f ^^ Suc n) {} for n
    using funpow_decreasing [of n Suc n] by auto
  show lfp f ⊆ ?U
  proof (rule lfp_lowerbound)
    have f ?U = (UN n. (f^^Suc n){})
      using chain_iterates[OF mono_if_cont[OF assms]] assms
      by(simp add: cont_def)
    also have ... = (f^^0){} ∪ ... by simp
    also have ... = ?U
        using mono by auto (metis funpow_simps_right(2) funpow_swap1
o_apply)
    finally show f ?U ⊆ ?U by simp
  qed
next
  have (f^^n){} ⊆ p if f p ⊆ p for n p
  proof -
    show ?thesis
    proof(induction n)
      case 0 show ?case by simp
    next
      case Suc
      from monoD[OF mono_if_cont[OF assms] Suc] ‹f p ⊆ p›
      show ?case by simp
    qed
  qed
  thus ?U ⊆ lfp f by(auto simp: lfp_def)
qed

lemma cont_W: cont(W b r)
by(auto simp: cont_def W_def)
```

5.2 The denotational semantics is deterministic

lemma *single_valued_UN_chain*:
  assumes *chain S* ($\bigwedge n$. *single_valued* (*S n*))
  shows *single_valued*(*UN n. S n*)
proof(*auto simp*: *single_valued_def*)
  fix *m n x y z* **assume** $(x, y) \in S\ m$ $(x, z) \in S\ n$
  with *chain_total*[*OF assms*(*1*), *of m n*] *assms*(*2*)
  show $y = z$ **by** (*auto simp*: *single_valued_def*)
qed

lemma *single_valued_lfp*: **fixes** *f* :: *com_den* $\Rightarrow$ *com_den*
assumes *cont f* $\bigwedge r$. *single_valued r* $\Longrightarrow$ *single_valued* (*f r*)
shows *single_valued*(*lfp f*)
unfolding *lfp_if_cont*[*OF assms*(*1*)]
proof(*rule single_valued_UN_chain*[*OF chain_iterates*[*OF mono_if_cont*[*OF assms*(*1*)]]])
  fix *n* **show** *single_valued* ((*f* ^^ *n*) {})
  by(*induction n*)(*auto simp*: *assms*(*2*))
qed

lemma *single_valued_D*: *single_valued* (*D c*)
proof(*induction c*)
  case *Seq* **thus** *?case* **by**(*simp add*: *single_valued_relcomp*)
next
  case (*While b c*)
  let *?f* = *W* (*bval b*) (*D c*)
  have *single_valued* (*lfp ?f*)
  proof(*rule single_valued_lfp*[*OF cont_W*])
    show $\bigwedge r$. *single_valued r* $\Longrightarrow$ *single_valued* (*?f r*)
      using *While.IH* **by**(*force simp*: *single_valued_def W_def*)
  qed
  thus *?case* **by** *simp*
qed (*auto simp add*: *single_valued_def*)

end

6 Compiler for IMP

theory *Compiler* **imports** *Big_Step Star*
begin

6.1 List setup

In the following, we use the length of lists as integers instead of natural numbers. Instead of converting *nat* to *int* explicitly, we tell Isabelle to coerce *nat* automatically when necessary.

declare [[*coercion_enabled*]]
declare [[*coercion int* :: *nat* ⇒ *int*]]

Similarly, we will want to access the ith element of a list, where i is an *int*.

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

The only additional lemma we need about this function is indexing over append:

lemma *inth_append* [*simp*]:
  0 ≤ *i* ⟹
  (*xs* @ *ys*) !! *i* = (*if i* < *size xs then xs* !! *i else ys* !! (*i* − *size xs*))
by (*induction xs arbitrary*: *i*) (*auto simp*: *algebra_simps*)

We hide coercion *int* applied to *length*:

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

notation *isize* (*size*)

6.2 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 ⇒ (*i*+*1*,*s*(*x* := *hd stk*),*tl stk*) |
  JMP n ⇒ (*i*+*1*+*n*,*s*,*stk*) |
  JMPLESS n ⇒ (*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
$exec1 :: instr\ list \Rightarrow config \Rightarrow config \Rightarrow bool$
$((\_/ \vdash (\_ \rightarrow/\ \_))\ [59,0,59]\ 60)$
where
$P \vdash c \rightarrow c' =$
$(\exists\, i\ s\ stk.\ c = (i,s,stk) \wedge c' = iexec(P!!i)\ (i,s,stk) \wedge 0 \leq i \wedge i < size\ P)$

lemma $exec1I$ [$intro$, $code\_pred\_intro$]:
$c' = iexec\ (P!!i)\ (i,s,stk) \Longrightarrow 0 \leq i \Longrightarrow i < size\ P$
$\Longrightarrow P \vdash (i,s,stk) \rightarrow c'$
by ($simp\ add$: $exec1\_def$)

abbreviation
$exec :: instr\ list \Rightarrow config \Rightarrow config \Rightarrow bool\ ((\_/ \vdash (\_ \rightarrow{*}/\ \_))\ 50)$
where
$exec\ P \equiv star\ (exec1\ P)$

lemmas $exec\_induct = star.induct$ [$of\ exec1\ P$, $split\_format(complete)$]

code_pred $exec1$ **by** ($metis\ exec1\_def$)

values
$\{(i,map\ t\ [''x'',''y''],stk) \mid i\ t\ stk.$
$[LOAD\ ''y'',\ STORE\ ''x''] \vdash$
$(0,\ <''x'' := 3,\ ''y'' := 4>,\ [])\ \rightarrow{*}\ (i,t,stk)\}$

6.3 Verification infrastructure

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 ($auto\ simp$: $exec1\_def$)

lemma $exec\_appendR$: $P \vdash c \rightarrow{*}\ c' \Longrightarrow P@P' \vdash c \rightarrow{*}\ c'$
by ($induction\ rule$: $star.induct$) ($fastforce\ intro$: $star.step\ exec1\_appendR$)+

lemma $exec1\_appendL$:

fixes $i\ i' :: int$
shows
$P \vdash (i,s,stk) \to (i',s',stk') \Longrightarrow$
$P' \mathbin{@} P \vdash (size(P')+i,s,stk) \to (size(P')+i',s',stk')$
unfolding *exec1_def*
by (*auto simp del*: *iexec.simps*)

lemma *exec_appendL*:
fixes $i\ i' :: int$
shows
$P \vdash (i,s,stk) \to* (i',s',stk') \Longrightarrow$
$P' \mathbin{@} P \vdash (size(P')+i,s,stk) \to* (size(P')+i',s',stk')$
by (*induction rule*: *exec_induct*) (*blast intro*: *star.step exec1_appendL*)+

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

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

lemma *exec_Cons_1* [*intro*]:
$P \vdash (0,s,stk) \to* (j,t,stk') \Longrightarrow$
$instr\#P \vdash (1,s,stk) \to* (1+j,t,stk')$
by (*drule exec_appendL*[**where** P'=[*instr*]]) *simp*

lemma *exec_appendL_if* [*intro*]:
fixes $i\ i'\ j :: int$
shows
$size\ P' <= i$
$\Longrightarrow P \vdash (i - size\ P',s,stk) \to* (j,s',stk')$
$\Longrightarrow i' = size\ P' + j$
$\Longrightarrow P' \mathbin{@} P \vdash (i,s,stk) \to* (i',s',stk')$
by (*drule exec_appendL*[**where** P'=P']) *simp*

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

lemma *exec_append_trans*[*intro*]:
fixes $i'\ i''\ j'' :: int$
shows
$P \vdash (0,s,stk) \to* (i',s',stk') \Longrightarrow$
$size\ P \le i' \Longrightarrow$
$P' \vdash (i' - size\ P,s',stk') \to* (i'',s'',stk'') \Longrightarrow$
$j'' = size\ P + i''$
$\Longrightarrow$

```
 P @ P' ⊢ (0,s,stk) →* (j'',s'',stk'')
by(metis star_trans[OF exec_appendR exec_appendL_if])


declare Let_def[simp]
```

6.4 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) →* (size(acomp a),s,aval a s#stk)
by (induction a arbitrary: stk) fastforce+

fun bcomp :: bexp ⇒ bool ⇒ int ⇒ instr list where
bcomp (Bc v) f n = (if v=f then [JMP n] else []) |
bcomp (Not b) f n = bcomp b (¬f) n |
bcomp (And b1 b2) f n =
 (let cb2 = bcomp b2 f n;
        m = if f then size cb2 else (size cb2::int)+n;
      cb1 = bcomp b1 False m
  in cb1 @ cb2) |
bcomp (Less a1 a2) f n =
 acomp a1 @ acomp a2 @ (if f 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]:
  fixes n :: int
  shows
  0 ≤ n ⟹
  bcomp b f n ⊢
 (0,s,stk)  →*  (size(bcomp b f n) + (if f = bval b s then n else 0),s,stk)
proof(induction b arbitrary: f n)
  case Not
  from Not(1)[where f=~f] Not(2) show ?case by fastforce
next
  case (And b1 b2)
  from And(1)[of if f then size(bcomp b2 f n) else size(bcomp b2 f n) + n
```

```
              False]
      And(2)[of n f] 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 (size cc₁ + 1)
   in cb @ cc₁ @ JMP (size cc₂) # cc₂) |
ccomp (WHILE b DO c) =
 (let cc = ccomp c; cb = bcomp b False (size cc + 1)
  in cb @ cc @ [JMP (−(size cb + size 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)))
```

6.5 Preservation of semantics

```
lemma ccomp_bigstep:
  (c,s) ⇒ t ⟹ ccomp c ⊢ (0,s,stk) →* (size(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) →* (size ?cc1,s2,stk)
    using Seq.IH(1) by fastforce
  moreover
  have ?cc1 @ ?cc2 ⊢ (size ?cc1,s2,stk) →* (size(?cc1 @ ?cc2),s3,stk)
    using Seq.IH(2) by fastforce
  ultimately show ?case by simp (blast intro: star_trans)
next
  case (WhileTrue b s1 c s2 s3)
  let ?cc = ccomp c
  let ?cb = bcomp b False (size ?cc + 1)
```

```
  let ?cw = ccomp(WHILE b DO c)
  have ?cw ⊢ (0,s1,stk) →* (size ?cb,s1,stk)
    using ‹bval b s1› by fastforce
  moreover
  have ?cw ⊢ (size ?cb,s1,stk) →* (size ?cb + size ?cc,s2,stk)
    using WhileTrue.IH(1) by fastforce
  moreover
  have ?cw ⊢ (size ?cb + size ?cc,s2,stk) →* (0,s2,stk)
    by fastforce
  moreover
  have ?cw ⊢ (0,s2,stk) →* (size ?cw,s3,stk) by(rule WhileTrue.IH(2))
  ultimately show ?case by(blast intro: star_trans)
qed fastforce+

end
```

```
theory Compiler2
imports Compiler
begin
```

The preservation of the source code semantics is already shown in the parent theory *HOL−IMP.Compiler*. This here shows the second direction.

7 Compiler Correctness, Reverse Direction

7.1 Definitions

Execution in n steps for simpler induction

```
primrec
  exec_n :: instr list ⇒ config ⇒ nat ⇒ config ⇒ bool
  (_/ ⊢ (_ →^_/ _) [65,0,1000,55] 55)
where
  P ⊢ c →^0 c' = (c'=c) |
  P ⊢ c →^(Suc n) c'' = (∃ c'. (P ⊢ c → c') ∧ P ⊢ c' →^n c'')
```

The possible successor PCs of an instruction at position n

```
definition isuccs :: instr ⇒ int ⇒ int set where
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 PCs of an instruction list

definition *succs* :: *instr list* ⇒ *int* ⇒ *int set* **where**
$succs\ P\ n = \{s.\ \exists\, i{::}int.\ 0 \le i \wedge i < size\ P \wedge s \in isuccs\ (P!!i)\ (n{+}i)\}$

Possible exit PCs of a program

definition *exits* :: *instr list* ⇒ *int set* **where**
$exits\ P = succs\ P\ 0 - \{0..<\ size\ P\}$

7.2 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 intro*: *star.step*)

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*: *star.induct*) (*auto 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 simp*: *exec1_def*)

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

7.3 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:
  size P <= fst c ⟹ ¬ P ⊢ c → c'
  by (auto simp: exec1_def)

lemma exec_n_end:
  size P <= (n::int) ⟹
  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
```

7.4 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::int. 0 ≤ i ∧ i < size P ∧ p ∈ isuccs (P!!i) (n+i)
  have p ∈ ?x ∪ ?xs if assm: p ∈ succs (x#xs) n for p
  proof −
    from assm obtain i::int where isuccs: ?isuccs p (x#xs) n i
      unfolding succs_def by auto
    show ?thesis
    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
```

```
  qed
  thus succs (x#xs) n ⊆ ?x ∪ ?xs ..

  have p ∈ succs (x#xs) n if assm: p ∈ ?x ∨ p ∈ ?xs for p
  proof −
    from assm show ?thesis
    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
  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 < size 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 ((+) (i::int))
  by (metis add_minus_cancel inj_on_inverseI)

lemma succs_set_shift [simp]:
  (+) 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 + size xs)
  by (induct xs arbitrary: n) (auto simp: succs_Cons algebra_simps)


lemma exits_append [simp]:
  exits (xs @ ys) = exits xs ∪ ((+) (size xs)) ` exits ys −
```

$\{0..<size\ xs + size\ ys\}$
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 $(x \# xs) = (isuccs\ x\ 0 - \{0\}) \cup ((+)\ 1)$ ` *exits* xs −
$\{0..<1 + size\ xs\}$
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*}
$i \neq -1 \Longrightarrow$ *exits* [*JMP i*] = $\{1 + i\}$
$i \neq -1 \Longrightarrow$ *exits* [*JMPGE i*] = $\{1 + i, 1\}$
$i \neq -1 \Longrightarrow$ *exits* [*JMPLESS i*] = $\{1 + i, 1\}$
by (*auto simp*: *exits_def*)

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

lemma *acomp_size*:
$(1::int) \leq size\ (acomp\ a)$
by (*induct a*) *auto*

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

lemma *bcomp_succs*:
$0 \leq i \Longrightarrow$
succs (*bcomp b f i*) $n \subseteq \{n\ ..\ n + size\ (bcomp\ b\ f\ i)\}$
$\cup \{n + i + size\ (bcomp\ b\ f\ i)\}$
proof (*induction b arbitrary*: *f i n*)
case (*And b1 b2*)
from *And.prems*

```
  show ?case
    by (cases f)
       (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:
  fixes i :: int
  shows
  0 ≤ i ⟹
  exits (bcomp b f i) ⊆ {size (bcomp b f i), i + size (bcomp b f i)}
  by (auto simp: exits_def)

lemma bcomp_exitsD [dest!]:
  p ∈ exits (bcomp b f i) ⟹ 0 ≤ i ⟹
  p = size (bcomp b f i) ∨ p = i + size (bcomp b f i)
  using bcomp_exits by auto

lemma ccomp_succs:
  succs (ccomp c) n ⊆ {n..n + size (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) ⊆ {size (ccomp c)}
```

using *ccomp_succs* [*of c 0*] **by** (*auto simp*: *exits_def*)

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

7.5 Splitting up machine executions

lemma *exec1_split*:
fixes *i j* :: *int*
shows
$P\ @\ c\ @\ P' \vdash (size\ P + i,\ s) \to (j,s') \Longrightarrow 0 \le i \Longrightarrow i < size\ c \Longrightarrow$
$c \vdash (i,s) \to (j - size\ P,\ s')$
by (*auto split*: *instr.splits simp*: *exec1_def*)

lemma *exec_n_split*:
fixes *i j* :: *int*
assumes $P\ @\ c\ @\ P' \vdash (size\ P + i,\ s) \to\hat{}n\ (j,\ s')$
$0 \le i$ $i < size\ c$
$j \notin \{size\ P\ ..{<}\ size\ P + size\ c\}$
shows $\exists\, s''\ (i'{::}int)\ k\ m.$
$c \vdash (i,\ s) \to\hat{}k\ (i',\ s'') \wedge$
$i' \in exits\ c \wedge$
$P\ @\ c\ @\ P' \vdash (size\ P + i',\ s'') \to\hat{}m\ (j,\ s') \wedge$
$n = k + m$
using *assms* **proof** (*induction n arbitrary*: *i j s*)
case *0*
thus *?case* **by** *simp*
next
case (*Suc n*)
have *i*: $0 \le i$ $i < size\ c$ **by** *fact+*
from *Suc.prems*
have *j*: $\neg\ (size\ P \le j \wedge j < size\ P + size\ c)$ **by** *simp*
from *Suc.prems*
obtain *i0 s0* **where**
step: $P\ @\ c\ @\ P' \vdash (size\ P + i,\ s) \to (i0,s0)$ **and**
rest: $P\ @\ c\ @\ P' \vdash (i0,s0) \to\hat{}n\ (j,\ s')$
by *clarsimp*

from *step i*
have *c*: $c \vdash (i,s) \to (i0 - size\ P,\ s0)$ **by** (*rule exec1_split*)

have $i0 = size\ P + (i0 - size\ P)$ **by** *simp*
then obtain *j0*::*int* **where** *j0*: $i0 = size\ P + j0$ **..**

note *split_paired_Ex* [*simp del*]

have *?case* **if** *assm*: $j0 \in \{0 ..< size\ c\}$
proof −
from *assm j0 j rest c* **show** *?case*
by (*fastforce dest*!: *Suc.IH intro*!: *exec_Suc*)
qed
moreover
have *?case* **if** *assm*: $j0 \notin \{0 ..< size\ c\}$
proof −
from *c j0* **have** $j0 \in succs\ c\ 0$
by (*auto dest*: *succs_iexec1 simp*: *exec1_def simp del*: *iexec.simps*)
with *assm* **have** $j0 \in exits\ c$ **by** (*simp add*: *exits_def*)
with *c j0 rest* **show** *?case* **by** *fastforce*
qed
ultimately
show *?case* **by** *cases*
qed

lemma *exec_n_drop_right*:
fixes $j :: int$
assumes $c\ @\ P' \vdash (0, s) \rightarrow\hat{}n\ (j, s')\ j \notin \{0..<size\ c\}$
shows $\exists s''\ i'\ k\ m.$
$(if\ c = []\ then\ s'' = s \wedge i' = 0 \wedge k = 0$
$else\ c \vdash (0, s) \rightarrow\hat{}k\ (i', s'') \wedge$
$i' \in exits\ c) \wedge$
$c\ @\ P' \vdash (i', s'') \rightarrow\hat{}m\ (j, s') \wedge$
$n = k + m$
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*:
fixes $i\ n :: int$
assumes $P1\ @\ P2 \vdash (i, s, stk) \rightarrow (n, s', stk')$ **and** $size\ P1 \leq i$
shows $P2 \vdash (i - size\ P1, s, stk) \rightarrow (n - size\ P1, s', stk')$
proof −
have $i = size\ P1 + (i - size\ P1)$ **by** *simp*
then obtain $i' :: int$ **where** $i = size\ P1 + i'$ **..**
moreover
have $n = size\ P1 + (n - size\ P1)$ **by** *simp*
then obtain $n' :: int$ **where** $n = size\ P1 + n'$ **..**

ultimately
show *?thesis* **using** *assms*
by (*clarsimp simp*: *exec1_def simp del*: *iexec.simps*)
qed

lemma *exec_n_drop_left*:
fixes *i n* :: *int*
assumes P @ $P' \vdash (i, s, stk) \to\hat{\ }k\ (n, s', stk')$
$size\ P \leq i\ exits\ P' \subseteq \{0..\}$
shows $P' \vdash (i - size\ P, s, stk) \to\hat{\ }k\ (n - size\ P, s', stk')$
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 @ $P' \vdash (i, s, stk) \to (i', s'', stk'')$ **and**
rest: P @ $P' \vdash (i', s'', stk'') \to\hat{\ }k\ (n, s', stk')$
by *auto*
from *step* ‹$size\ P \leq i$›
have $*$: $P' \vdash (i - size\ P, s, stk) \to (i' - size\ P, s'', stk'')$
by (*rule exec1_drop_left*)
then have $i' - size\ P \in succs\ P'\ 0$
by (*fastforce dest*!: *succs_iexec1 simp*: *exec1_def simp del*: *iexec.simps*)
with ‹$exits\ P' \subseteq \{0..\}$›
have $size\ P \leq i'$ **by** (*auto simp*: *exits_def*)
from *rest this* ‹$exits\ P' \subseteq \{0..\}$›
have $P' \vdash (i' - size\ P, s'', stk'') \to\hat{\ }k\ (n - size\ P, s', stk')$
by (*rule Suc.IH*)
with $*$ **show** *?case* **by** *auto*
qed

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

definition
$closed\ P \longleftrightarrow exits\ P \subseteq \{size\ P\}$

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:
  fixes j :: int
  assumes exec: P @ P' ⊢ (0,s,stk) →^k (j, s', stk')
  assumes P: size P ≤ j
  assumes closed: closed P
  assumes exits: exits P' ⊆ {0..}
  shows ∃ k1 k2 s'' stk''. P ⊢ (0,s,stk) →^k1 (size P, s'', stk'') ∧
                           P' ⊢ (0,s'',stk'') →^k2 (j − size P, s', stk')
proof (cases P)
  case Nil with exec
  show ?thesis by fastforce
next
  case Cons
  hence 0 < size P by simp
  with exec P closed
  obtain k1 k2 s'' stk'' where
    1: P ⊢ (0,s,stk) →^k1 (size P, s'', stk'') and
    2: P @ P' ⊢ (size P,s'',stk'') →^k2 (j, s', stk')
    by (auto dest!: exec_n_split [where P=[] and i=0, simplified]
             simp: closed_def)
  moreover
  have j = size P + (j − size P) by simp
  then obtain j0 :: int where j = size P + j0 ..
  ultimately
  show ?thesis using exits
    by (fastforce dest: exec_n_drop_left)
qed
```

7.6 Correctness theorem

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

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

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

  thus ?case by (fastforce dest: Plus.IH simp: exec_n_simps exec1_def)
qed (auto simp: exec_n_simps exec1_def)

lemma bcomp_split:
  fixes i j :: int
  assumes bcomp b f i @ P' ⊢ (0, s, stk) →^n (j, s', stk')
          j ∉ {0..<size (bcomp b f i)} 0 ≤ i
  shows ∃ s'' stk'' (i'::int) k m.
           bcomp b f i ⊢ (0, s, stk) →^k (i', s'', stk'') ∧
           (i' = size (bcomp b f i) ∨ i' = i + size (bcomp b f i)) ∧
           bcomp b f i @ P' ⊢ (i', s'', stk'') →^m (j, s', stk') ∧
           n = k + m
 using assms by (cases bcomp b f i = []) (fastforce dest!: exec_n_drop_right)+

lemma bcomp_exec_n [dest]:
  fixes i j :: int
  assumes bcomp b f j ⊢ (0, s, stk) →^n (i, s', stk')
          size (bcomp b f j) ≤ i 0 ≤ j
  shows i = size(bcomp b f j) + (if f = bval b s then j else 0) ∧
        s' = s ∧ stk' = stk
using assms proof (induction b arbitrary: f j i n s' stk')
  case Bc thus ?case
    by (simp split: if_split_asm add: exec_n_simps exec1_def)
next
  case (Not b)
  from Not.prems show ?case
    by (fastforce dest!: Not.IH)
next
  case (And b1 b2)

  let ?b2 = bcomp b2 f j
  let ?m  = if f then size ?b2 else size ?b2 + j
  let ?b1 = bcomp b1 False ?m

  have j: size (bcomp (And b1 b2) f j) ≤ i 0 ≤ j by fact+

  from And.prems
  obtain s'' stk'' and i'::int and k m where
```

    b1: *?b1* ⊢ (*0*, *s*, *stk*) →^*k* (*i'*, *s''*, *stk''*)
        i' = *size ?b1* ∨ *i'* = *?m* + *size ?b1* **and**
    b2: *?b2* ⊢ (*i'* − *size ?b1*, *s''*, *stk''*) →^*m* (*i* − *size ?b1*, *s'*, *stk'*)
    by (*auto dest*!: *bcomp_split dest*: *exec_n_drop_left*)
  from *b1 j*
  have *i'* = *size ?b1* + (*if* ¬*bval b1 s then ?m else 0*) ∧ *s''* = *s* ∧ *stk''* = *stk*
    by (*auto dest*!: *And.IH*)
  with *b2 j*
  show *?case*
    by (*fastforce dest*!: *And.IH simp*: *exec_n_end split*: *if_split_asm*)
next
  case *Less*
  thus *?case* **by** (*auto dest*!: *exec_n_split_full simp*: *exec_n_simps exec1_def*)

qed

lemma *ccomp_empty* [*elim*!]:
  ccomp c = [] ⟹ (*c*,*s*) ⇒ *s*
  by (*induct c*) *auto*

declare *assign_simp* [*simp*]

lemma *ccomp_exec_n*:
  ccomp c ⊢ (*0*,*s*,*stk*) →^*n* (*size*(*ccomp c*),*t*,*stk'*)
  ⟹ (*c*,*s*) ⇒ *t* ∧ *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 exec1_def*)
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* (*size* (*ccomp c1*) + *1*)

```
  from ‹?cs ⊢ (0,s,stk) →^n (size ?cs,t,stk')›
  obtain i' :: int and k m s'' stk'' where
    cs: ?cs ⊢ (i',s'',stk'') →^m (size ?cs,t,stk') and
        ?bcomp ⊢ (0,s,stk) →^k (i', s'', stk'')
        i' = size ?bcomp ∨ i' = size ?bcomp + size (ccomp c1) + 1
    by (auto dest!: bcomp_split)

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

  with cs have cs':
    ccomp c1@JMP (size (ccomp c2))#ccomp c2 ⊢
       (if bval b s then 0 else size (ccomp c1)+1, s, stk) →^m
       (1 + size (ccomp c1) + size (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: if_split_asm
                  simp: exec_n_simps exec1_def)
  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)

    have ?case if assm: ¬ bval b s
    proof −
      from assm 1.prems
```

```
  show ?case
    by simp (fastforce dest!: bcomp_split simp: exec_n_simps)
qed
moreover
have ?case if b: bval b s
proof -
  let ?c0 = WHILE b DO c
  let ?cs = ccomp ?c0
  let ?bs = bcomp b False (size (ccomp c) + 1)
  let ?jmp = [JMP (-((size ?bs + size (ccomp c) + 1)))]

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

  show ?case
  proof cases
    assume ccomp c = []
    with cs k
    obtain m where
      ?cs ⊢ (0,s,stk) →^m (size (ccomp ?c0), t, stk')
      m < n
      by (auto simp: exec_n_step [where k=k] exec1_def)
    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' (size (ccomp c), s'', stk'') and
      rest: ?cs ⊢ (size ?bs + size (ccomp c), s'', stk'') →^m
                  (size ?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' (size ?cs, t, stk')
      k' < n
```

by (*auto simp*: *exec_n_step* [**where** $k{=}m$] *exec1_def*)
with *1.IH*
have (*?c0*, s'') $\Rightarrow$ t $\wedge$ $stk' = stk$ **by** *blast*
ultimately
show *?case* **using** *b* **by** *blast*
qed
qed
ultimately show *?case* **by** *cases*
qed
qed

theorem *ccomp_exec*:
ccomp c $\vdash$ (*0*,*s*,*stk*) $\rightarrow*$ (*size*(*ccomp c*),*t*,stk') $\Longrightarrow$ (*c*,*s*) $\Rightarrow$ *t*
by (*auto dest*: *exec_exec_n ccomp_exec_n*)

corollary *ccomp_sound*:
ccomp c $\vdash$ (*0*,*s*,*stk*) $\rightarrow*$ (*size*(*ccomp c*),*t*,*stk*) $\longleftrightarrow$ (*c*,*s*) $\Rightarrow$ *t*
by (*blast intro*!: *ccomp_exec ccomp_bigstep*)

end

8 A Typed Language

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

We build on *Complex_Main* instead of *Main* to access the real numbers.

8.1 Arithmetic Expressions

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

type_synonym *vname* = *string*
type_synonym *state* = *vname* $\Rightarrow$ *val***datatype** *aexp* = *Ic int* | *Rc real* |
V vname | *Plus aexp aexp*

inductive *taval* :: *aexp* $\Rightarrow$ *state* $\Rightarrow$ *val* $\Rightarrow$ *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*) $\Longrightarrow$ *taval a2 s* (*Iv i2*)
$\Longrightarrow$ *taval* (*Plus a1 a2*) *s* (*Iv*(*i1*+*i2*)) |
taval a1 s (*Rv r1*) $\Longrightarrow$ *taval a2 s* (*Rv r2*)
$\Longrightarrow$ *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*

8.2 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)

8.3 Syntax of Commands

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*)

8.4 Small-Step Semantics of Commands

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

Seq1: (*SKIP*;;*c*,*s*) → (*c*,*s*) |
Seq2: (*c1*,*s*) → (*c1'*,*s'*) ⟹ (*c1*;;*c2*,*s*) → (*c1'*;;*c2*,*s'*) |

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

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

lemmas *small_step_induct* = *small_step.induct*[*split_format*(*complete*)]

8.5 The Type System

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

type_synonym *tyenv* = *vname* ⇒ *ty*

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

declare *atyping.intros* [*intro*!]
inductive_cases [*elim*!]:
  $\Gamma \vdash$ *V x* : τ $\Gamma \vdash$ *Ic i* : τ $\Gamma \vdash$ *Rc r* : τ $\Gamma \vdash$ *Plus a1 a2* : τ

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

inductive *btyping* :: *tyenv* ⇒ *bexp* ⇒ *bool* (**infix** ⊢ *50*)
where
B_ty: $\Gamma \vdash$ *Bc v* |
Not_ty: $\Gamma \vdash b \Longrightarrow \Gamma \vdash$ *Not b* |
And_ty: $\Gamma \vdash$ *b1* $\Longrightarrow \Gamma \vdash$ *b2* $\Longrightarrow \Gamma \vdash$ *And b1 b2* |
Less_ty: $\Gamma \vdash$ *a1* : $\tau \Longrightarrow \Gamma \vdash$ *a2* : $\tau \Longrightarrow \Gamma \vdash$ *Less a1 a2*

declare *btyping.intros* [*intro*!]
inductive_cases [*elim*!]: $\Gamma \vdash$ *Not b* $\Gamma \vdash$ *And b1 b2* $\Gamma \vdash$ *Less a1 a2*

inductive *ctyping* :: *tyenv* ⇒ *com* ⇒ *bool* (**infix** ⊢ *50*) **where**
Skip_ty: $\Gamma \vdash$ *SKIP* |
Assign_ty: $\Gamma \vdash a : \Gamma(x) \Longrightarrow \Gamma \vdash x$ ::= *a* |
Seq_ty: $\Gamma \vdash$ *c1* $\Longrightarrow \Gamma \vdash$ *c2* $\Longrightarrow \Gamma \vdash$ *c1*;;*c2* |
If_ty: $\Gamma \vdash b \Longrightarrow \Gamma \vdash$ *c1* $\Longrightarrow \Gamma \vdash$ *c2* $\Longrightarrow \Gamma \vdash$ *IF b THEN c1 ELSE c2* |
While_ty: $\Gamma \vdash b \Longrightarrow \Gamma \vdash c \Longrightarrow \Gamma \vdash$ *WHILE b DO c*

declare *ctyping.intros* [*intro*!]
inductive_cases [*elim*!]:
  $\Gamma \vdash$ *x* ::= *a* $\Gamma \vdash$ *c1*;;*c2*
  $\Gamma \vdash$ *IF b THEN c1 ELSE c2*
  $\Gamma \vdash$ *WHILE b DO c*

8.6 Well-typed Programs Do Not Get Stuck

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

lemma type_eq_Ity[simp]: type v = Ity ⟷ (∃ i. v = Iv i)
by (cases v) simp_all

lemma type_eq_Rty[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*:
 $(c,s) \to\!* (c',s') \Longrightarrow \Gamma \vdash c \Longrightarrow \Gamma \vdash s \Longrightarrow c' \neq SKIP$
 $\Longrightarrow \exists\, cs''.\ (c',s') \to cs''$
apply(*induction rule*:*star_induct*)
apply (*metis progress*)
by (*metis styping_preservation ctyping_preservation*)

end
theory *Poly_Types* **imports** *Types* **begin**

8.7 Type Variables

datatype *ty* = *Ity* | *Rty* | *TV nat*

Everything else remains the same.

type_synonym *tyenv* = *vname* ⇒ *ty*

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

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

inductive *ctyping* :: *tyenv* ⇒ *com* ⇒ *bool* (**infix** ⊢p *50*) **where**
Γ ⊢p *SKIP* |
Γ ⊢p *a* : Γ(*x*) ⟹ Γ ⊢p *x* ::= *a* |
Γ ⊢p *c1* ⟹ Γ ⊢p *c2* ⟹ Γ ⊢p *c1*;;*c2* |
Γ ⊢p *b* ⟹ Γ ⊢p *c1* ⟹ Γ ⊢p *c2* ⟹ Γ ⊢p *IF b THEN c1 ELSE c2* |
Γ ⊢p *b* ⟹ Γ ⊢p *c* ⟹ Γ ⊢p *WHILE b DO c*

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

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

fun *tsubst* :: (*nat* ⇒ *ty*) ⇒ *ty* ⇒ *ty* **where**
tsubst S (*TV n*) = *S n* |
tsubst S t = *t*

8.8 Typing is Preserved by Substitution

lemma *subst_atyping*: *E* ⊢*p* *a* : *t* ⟹ *tsubst S* ∘ *E* ⊢*p* *a* : *tsubst S t*
apply(*induction rule*: *atyping.induct*)
apply(*auto intro*: *atyping.intros*)
done

lemma *subst_btyping*: *E* ⊢*p* (*b*::*bexp*) ⟹ *tsubst S* ∘ *E* ⊢*p* *b*
apply(*induction rule*: *btyping.induct*)
apply(*auto intro*: *btyping.intros*)
apply(*drule subst_atyping*[**where** *S*=*S*])
apply(*drule subst_atyping*[**where** *S*=*S*])
apply(*simp add*: *o_def btyping.intros*)
done

lemma *subst_ctyping*: *E* ⊢*p* (*c*::*com*) ⟹ *tsubst S* ∘ *E* ⊢*p* *c*
apply(*induction rule*: *ctyping.induct*)
apply(*auto intro*: *ctyping.intros*)
apply(*drule subst_atyping*[**where** *S*=*S*])
apply(*simp add*: *o_def ctyping.intros*)
apply(*drule subst_btyping*[**where** *S*=*S*])
apply(*simp add*: *o_def ctyping.intros*)
apply(*drule subst_btyping*[**where** *S*=*S*])
apply(*simp add*: *o_def ctyping.intros*)
done

end

9 Security Type Systems

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

9.1 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

9.2 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:
  ⟦ *max* (*sec b*) *l* ⊢ c_1; *max* (*sec b*) *l* ⊢ c_2 ⟧ ⟹ *l* ⊢ *IF b THEN* c_1 *ELSE* c_2 |
While:
  max (*sec b*) *l* ⊢ *c* ⟹ *l* ⊢ *WHILE b DO c*

code_pred (*expected_modes*: *i* => *i* => *bool*) *sec_type* **.**

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

```
inductive_cases [elim!]:
  l ⊢ x ::= a   l ⊢ c₁;;c₂   l ⊢ IF b THEN c₁ ELSE c₂   l ⊢ WHILE b DO c
```

An important property: anti-monotonicity.

```
lemma anti_mono: ⟦ l ⊢ c;  l' ≤ l ⟧ ⟹ l' ⊢ c
apply(induction arbitrary: l' rule: sec_type.induct)
apply (metis sec_type.intros(1))
apply (metis le_trans sec_type.intros(2))
apply (metis sec_type.intros(3))
apply (metis If le_refl sup_mono sup_nat_def)
apply (metis While le_refl sup_mono sup_nat_def)
done

lemma confinement: ⟦ (c,s) ⇒ t;  l ⊢ c ⟧ ⟹ s = t (< l)
proof(induction rule: big_step_induct)
  case Skip thus ?case by simp
next
  case Assign thus ?case by auto
next
  case Seq thus ?case by auto
next
  case (IfTrue b s c1)
  hence max (sec b) l ⊢ c1 by auto
  hence l ⊢ c1 by (metis max.cobounded2 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 max.cobounded2 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 max.cobounded2 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 ‹0 ⊢ IF b THEN c1 ELSE c2› 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 ‹(c1, s) ⇒ s'› ‹sec b ⊢ c1›] ‹¬ sec b ≤ l›
    have s = s' (≤ l) by auto
    moreover
     from confinement[OF ‹(IF b THEN c1 ELSE c2, t) ⇒ t'› 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 ‹0 ⊢ IF b THEN c1 ELSE c2› 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* ⟨(*IF b THEN c1 ELSE c2*, *t*) ⇒ *t'*⟩ *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* ⟨(*WHILE b DO c*, *t*) ⇒ *t'*⟩ *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** ⟨*0* ⊢ *WHILE b DO c*⟩ **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** $(c,t1) \Rightarrow t2$ $(?w,t2) \Rightarrow t3$
using $\langle(?w,t1) \Rightarrow t3\rangle$ **by** *auto*
from *WhileTrue.IH(2)[OF* $\langle(?w,t2) \Rightarrow t3\rangle$ $\langle 0 \vdash ?w\rangle$
WhileTrue.IH(1)[OF $\langle(c,t1) \Rightarrow t2\rangle$ *anti_mono[OF* $\langle sec\ b \vdash c\rangle$]
$\langle s1 = t1\ (\leq l)\rangle$]]
show *?thesis* **by** *simp*
next
assume $\neg$ *sec b* $\leq$ *l*
have *1*: *sec b* $\vdash$ *?w* **by**(*rule sec_type.intros*)(*simp_all add*: $\langle sec\ b \vdash c\rangle$)
from *confinement[OF big_step.WhileTrue[OF WhileTrue.hyps] 1]* $\langle\neg$ *sec b* $\leq l\rangle$
have $s1 = s3\ (\leq l)$ **by** *auto*
moreover
from *confinement[OF* $\langle(WHILE\ b\ DO\ c,\ t1) \Rightarrow t3\rangle$ *1]* $\langle\neg\ sec\ b \leq l\rangle$
have $t1 = t3\ (\leq l)$ **by** *auto*
ultimately show $s3 = t3\ (\leq l)$ **using** $\langle s1 = t1\ (\leq l)\rangle$ **by** *auto*
qed
qed

9.3 The Standard Typing System

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

inductive *sec_type'* :: *nat* $\Rightarrow$ *com* $\Rightarrow$ *bool* ((_/ $\vdash''$ _) [0,0] 50) **where**
Skip':
$l \vdash'$ *SKIP* |
Assign':
$[\![$ *sec x* $\geq$ *sec a*; *sec x* $\geq l$ $]\!] \Longrightarrow l \vdash' x ::= a$ |
Seq':
$[\![\ l \vdash' c_1;\ \ l \vdash' c_2\ ]\!] \Longrightarrow l \vdash' c_1;;c_2$ |
If':
$[\![$ *sec b* $\leq l$; $l \vdash' c_1$; $l \vdash' c_2$ $]\!] \Longrightarrow l \vdash'$ *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 max.commute max.absorb_iff2 nat_le_linear If′ anti_mono′*)
by (*metis less_or_eq_imp_le max.absorb1 max.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 max.absorb2 If*)
apply (*metis max.absorb2 While*)
by (*metis anti_mono*)

9.4 A Bottom-Up Typing System

inductive *sec_type2* :: *com* ⇒ *level* ⇒ *bool* ((⊢ _ : _) [*0*,*0*] *50*) **where**
Skip2:
$\vdash SKIP : l$ |
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.cobounded1 min.cobounded2*)
apply (*metis If′ anti_mono′ min.absorb2 min.absorb_iff1 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.boundedI*)
apply (*metis If2 inf_greatest inf_nat_def le_trans*)

apply (*metis While2 le_trans*)
by (*metis le_trans*)

end
theory *Sec_TypingT* **imports** *Sec_Type_Expr*
begin

9.5 A Termination-Sensitive Syntax Directed System

inductive *sec_type* :: *nat* ⇒ *com* ⇒ *bool* ((_/ ⊢ _) [*0*,*0*] *50*) **where**
Skip:
  $l \vdash SKIP$ |
Assign:
  ⟦ $sec\ x \geq sec\ a$; $sec\ x \geq l$ ⟧ $\Longrightarrow l \vdash x ::= a$ |
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*: $(c,s) \Rightarrow t \Longrightarrow l \vdash c \Longrightarrow s = t\ (< l)$
proof(*induction rule*: *big_step_induct*)
  case *Skip* **thus** *?case* **by** *simp*
next
  case *Assign* **thus** *?case* **by** *auto*
next
  case *Seq* **thus** *?case* **by** *auto*

```
next
  case (IfTrue b s c1)
  hence max (sec b) l ⊢ c1 by auto
  hence l ⊢ c1 by (metis max.cobounded2 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 max.cobounded2 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 max.cobounded2)
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 ‹0 ⊢ IF b THEN c1 ELSE c2› by auto
  obtain t' where t': (c1, t) ⇒ t' s' = t' (≤ l)
    using IfTrue.IH[OF anti_mono[OF ‹sec b ⊢ c1›] ‹s = t (≤ l)›] 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
      t': (IF b THEN c1 ELSE c2,t) ⇒ t' ..
    moreover
    from confinement[OF t' 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 ‹0 ⊢ IF b THEN c1 ELSE c2› by auto
  obtain t' where t': (c2, t) ⇒ t' s' = t' (≤ l)
    using IfFalse.IH[OF anti_mono[OF ‹sec b ⊢ c2›] ‹s = t (≤ l)›] 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
      t': (IF b THEN c1 ELSE c2,t) ⇒ t' ..
    moreover
    from confinement[OF t' 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 ‹0 ⊢ WHILE b DO c› by auto
  from WhileTrue.IH(1)[OF this ‹s = t (≤ l)›]
  obtain t'' where (c,t) ⇒ t'' and s'' = t'' (≤l) by blast
  from WhileTrue.IH(2)[OF ‹0 ⊢ ?w› this(2)]
  obtain t' where (?w,t'') ⇒ t' and s' = t' (≤l) by blast
  from ‹bval b s› have bval b t
    using bval_eq_if_eq_le[OF ‹s = t (≤l)›] by auto
  show ?case
    using big_step.WhileTrue[OF ‹bval b t› ‹(c,t) ⇒ t''› ‹(?w,t'') ⇒ t'›]
    by (metis ‹s' = t' (≤ l)›)
qed
```

9.6 The Standard Termination-Sensitive System

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

```
inductive sec_type' :: nat ⇒ com ⇒ bool ((_/ ⊢'' _) [0,0] 50) where
```

Skip′:
  $l \vdash' SKIP$ |
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′:
  $[\![sec\ b \leq l;\ \ l \vdash' c_1;\ \ l \vdash' c_2]\!] \Longrightarrow l \vdash' IF\ b\ THEN\ c_1\ ELSE\ c_2$ |
While′:
  $[\![sec\ b = 0;\ \ 0 \vdash' c]\!] \Longrightarrow 0 \vdash' WHILE\ b\ DO\ c$ |
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 max.commute max.absorb_iff2 nat_le_linear If′ anti_mono′*)
by (*metis While′*)

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 max.absorb2 If*)
apply (*metis While*)
by (*metis anti_mono*)

corollary *sec_type_eq*: $l \vdash c \longleftrightarrow l \vdash' c$
by (*metis sec_type′_sec_type sec_type_sec_type′*)

end

10 Definite Initialization Analysis

theory *Vars* **imports** *Com*
begin

10.1 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* ⇒ *vname set* **where**
vars (*N n*) = {} |
vars (*V x*) = {*x*} |
vars (*Plus* a_1 a_2) = *vars* a_1 ∪ *vars* a_2

instance ..

end

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

instantiation *bexp* :: *vars*
begin

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

instance ..

end

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

abbreviation
eq_on :: ($'a \Rightarrow 'b$) ⇒ ($'a \Rightarrow 'b$) ⇒ $'a$ *set* ⇒ *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*

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

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

instantiation *com* :: *vars*
begin

definition *vars_com c* = *lvars c* ∪ *rvars c*

instance ..

end

lemma *vars_com_simps*[*simp*]:
  vars 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*

$vars\ (WHILE\ b\ DO\ c) = vars\ b \cup vars\ c$
by(*auto simp*: *vars_com_def*)

lemma *finite_avars*[*simp*]: *finite*(*vars*(*a*::*aexp*))
by(*induction a*) *simp_all*

lemma *finite_bvars*[*simp*]: *finite*(*vars*(*b*::*bexp*))
by(*induction b*) *simp_all*

lemma *finite_lvars*[*simp*]: *finite*(*lvars*(*c*))
by(*induction c*) *simp_all*

lemma *finite_rvars*[*simp*]: *finite*(*rvars*(*c*))
by(*induction c*) *simp_all*

lemma *finite_cvars*[*simp*]: *finite*(*vars*(*c*::*com*))
by(*simp add*: *vars_com_def*)

end

theory *Def_Init_Exp*
imports *Vars*
begin

10.2 Initialization-Sensitive Expressions Evaluation

type_synonym $state = vname \Rightarrow val\ option$

fun $aval :: aexp \Rightarrow state \Rightarrow val\ option$ **where**
$aval\ (N\ i)\ s = Some\ i\ |$
$aval\ (V\ x)\ s = s\ x\ |$
$aval\ (Plus\ a_1\ a_2)\ s =$
$(case\ (aval\ a_1\ s,\ aval\ a_2\ s)\ of$
$(Some\ i_1, Some\ i_2) \Rightarrow Some(i_1+i_2)\ |\ \_ \Rightarrow None)$

fun $bval :: bexp \Rightarrow state \Rightarrow bool\ option$ **where**
$bval\ (Bc\ v)\ s = Some\ v\ |$
$bval\ (Not\ b)\ s = (case\ bval\ b\ s\ of\ None \Rightarrow None\ |\ Some\ bv \Rightarrow Some(\neg\ bv))$
$|$
$bval\ (And\ b_1\ b_2)\ s = (case\ (bval\ b_1\ s,\ bval\ b_2\ s)\ of$
$(Some\ bv_1,\ Some\ bv_2) \Rightarrow Some(bv_1\ \&\ bv_2)\ |\ \_ \Rightarrow None)\ |$

$bval\ (Less\ a_1\ a_2)\ s = (case\ (aval\ a_1\ s,\ aval\ a_2\ s)\ of$
$(Some\ i_1,\ Some\ i_2) \Rightarrow Some(i_1 < i_2) \mid \_ \Rightarrow None)$

lemma *aval_Some*: $vars\ a \subseteq dom\ s \Longrightarrow \exists\ i.\ aval\ a\ s = Some\ i$
by (*induct a*) *auto*

lemma *bval_Some*: $vars\ b \subseteq dom\ s \Longrightarrow \exists\ bv.\ bval\ b\ s = Some\ bv$
by (*induct b*) (*auto dest*!: *aval_Some*)

end
theory *Def_Init*
imports *Vars Com*
begin

10.3 Definite Initialization Analysis

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

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

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

end

theory *Def_Init_Big*
imports *Def_Init_Exp Def_Init*
begin

10.4 Initialization-Sensitive Big Step Semantics

inductive
$\mathit{big\_step} :: (\mathit{com} \times \mathit{state\ option}) \Rightarrow \mathit{state\ option} \Rightarrow \mathit{bool}$ (**infix** $\Rightarrow$ *55*)
where
None: $(c,\mathit{None}) \Rightarrow \mathit{None}$ |
Skip: $(\mathit{SKIP},s) \Rightarrow s$ |
$\mathit{AssignNone}$: $\mathit{aval}\ a\ s = \mathit{None} \Longrightarrow (x ::= a,\ \mathit{Some}\ s) \Rightarrow \mathit{None}$ |
Assign: $\mathit{aval}\ a\ s = \mathit{Some}\ i \Longrightarrow (x ::= a,\ \mathit{Some}\ s) \Rightarrow \mathit{Some}(s(x := \mathit{Some}\ i))$
|
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: $\mathit{bval}\ b\ s = \mathit{None} \Longrightarrow (\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2,\mathit{Some}\ s) \Rightarrow \mathit{None}$ |
IfTrue: $[\![\ \mathit{bval}\ b\ s = \mathit{Some}\ \mathit{True};\ (c_1,\mathit{Some}\ s) \Rightarrow s'\ ]\!] \Longrightarrow$
$(\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2,\mathit{Some}\ s) \Rightarrow s'$ |
$\mathit{IfFalse}$: $[\![\ \mathit{bval}\ b\ s = \mathit{Some}\ \mathit{False};\ (c_2,\mathit{Some}\ s) \Rightarrow s'\ ]\!] \Longrightarrow$
$(\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2,\mathit{Some}\ s) \Rightarrow s'$ |

$\mathit{WhileNone}$: $\mathit{bval}\ b\ s = \mathit{None} \Longrightarrow (\mathit{WHILE}\ b\ \mathit{DO}\ c,\mathit{Some}\ s) \Rightarrow \mathit{None}$ |
$\mathit{WhileFalse}$: $\mathit{bval}\ b\ s = \mathit{Some}\ \mathit{False} \Longrightarrow (\mathit{WHILE}\ b\ \mathit{DO}\ c,\mathit{Some}\ s) \Rightarrow \mathit{Some}$
s |
$\mathit{WhileTrue}$:
$[\![\ \mathit{bval}\ b\ s = \mathit{Some}\ \mathit{True};\ (c,\mathit{Some}\ s) \Rightarrow s';\ (\mathit{WHILE}\ b\ \mathit{DO}\ c,s') \Rightarrow s''\ ]\!]$
$\Longrightarrow$
$(\mathit{WHILE}\ b\ \mathit{DO}\ c,\mathit{Some}\ s) \Rightarrow s''$

lemmas $\mathit{big\_step\_induct} = \mathit{big\_step.induct}[\mathit{split\_format}(\mathit{complete})]$

10.5 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*:
$[\![\ (c,\mathit{Some}\ s) \Rightarrow s';\ D\ A\ c\ A';\ A \subseteq \mathit{dom}\ s\ ]\!]$
$\Longrightarrow \exists\ t.\ s' = \mathit{Some}\ t \wedge A' \subseteq \mathit{dom}\ t$
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

theory *Def_Init_Small*
imports *Star Def_Init_Exp Def_Init*
begin

10.6 Initialization-Sensitive Small Step Semantics

inductive
  small_step :: $(com \times state) \Rightarrow (com \times state) \Rightarrow bool$ (**infix** → *55*)
where
Assign: $aval\ a\ s = Some\ i \Longrightarrow (x ::= a,\ s) \rightarrow (SKIP,\ s(x := Some\ i))$ |

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

IfTrue: $bval\ b\ s = Some\ True \Longrightarrow (IF\ b\ THEN\ c_1\ ELSE\ c_2,s) \rightarrow (c_1,s)$ |
IfFalse: $bval\ b\ s = Some\ False \Longrightarrow (IF\ b\ THEN\ c_1\ ELSE\ c_2,s) \rightarrow (c_2,s)$ |

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

lemmas *small_step_induct* = *small_step.induct*[*split_format*(*complete*)]

abbreviation *small_steps* :: $com * state \Rightarrow com * state \Rightarrow bool$ (**infix** →∗ *55*)

where $x \to\!* \; y == star\ small\_step\ x\ y$

10.7 Soundness wrt Small Steps

theorem *progress*:
$D\ (dom\ s)\ c\ A' \Longrightarrow c \neq SKIP \Longrightarrow \exists\, cs'.\ (c,s) \to cs'$
proof (*induction c arbitrary*: *s A'*)
case *Assign* **thus** *?case* **by** *auto* (*metis aval_Some small_step.Assign*)
next
case (*If b c1 c2*)
then obtain *bv* **where** $bval\ b\ s = Some\ bv$ **by** (*auto dest*!:*bval_Some*)
then show *?case*
by(*cases bv*)(*auto intro*: *small_step.IfTrue small_step.IfFalse*)
qed (*fastforce intro*: *small_step.intros*)+

lemma *D_mono*: $D\ A\ c\ M \Longrightarrow A \subseteq A' \Longrightarrow \exists\, M'.\ D\ A'\ c\ M'\ \&\ M <= M'$
proof (*induction c arbitrary*: $A\ A'\ M$)
case *Seq* **thus** *?case* **by** *auto* (*metis D.intros*(*3*))
next
case (*If b c1 c2*)
then obtain *M1 M2* **where** $vars\ b \subseteq A\ \ D\ A\ c1\ M1\ \ D\ A\ c2\ M2\ \ M = M1 \cap M2$
by *auto*
with *If.IH* ‹$A \subseteq A'$› **obtain** $M1'\ M2'$
where $D\ A'\ c1\ M1'\ \ D\ A'\ c2\ M2'$ **and** $M1 \subseteq M1'\ \ M2 \subseteq M2'$ **by** *metis*
hence $D\ A'\ (IF\ b\ THEN\ c1\ ELSE\ c2)\ (M1' \cap M2')$ **and** $M \subseteq M1' \cap M2'$
using ‹$vars\ b \subseteq A$› ‹$A \subseteq A'$› ‹$M = M1 \cap M2$› **by**(*fastforce intro*: *D.intros*)+
thus *?case* **by** *metis*
next
case *While* **thus** *?case* **by** *auto* (*metis D.intros*(*5*) *subset_trans*)
qed (*auto intro*: *D.intros*)

theorem *D_preservation*:
$(c,s) \to (c',s') \Longrightarrow D\ (dom\ s)\ c\ A \Longrightarrow \exists\, A'.\ D\ (dom\ s')\ c'\ A'\ \&\ A <= A'$
proof (*induction arbitrary*: *A rule*: *small_step_induct*)
case (*While b c s*)
then obtain A' **where** A': $vars\ b \subseteq dom\ s\ \ A = dom\ s\ \ D\ (dom\ s)\ c\ A'$ **by** *blast*
then obtain A'' **where** $D\ A'\ c\ A''$ **by** (*metis D_incr D_mono*)
with A' **have** $D\ (dom\ s)\ (IF\ b\ THEN\ c;;\ WHILE\ b\ DO\ c\ ELSE\ SKIP)\ (dom\ s)$
by (*metis D.If*[*OF* ‹$vars\ b \subseteq dom\ s$› *D.Seq*[*OF* ‹$D\ (dom\ s)\ c\ A'$›

$D.While[OF\ \_\ \langle D\ A'\ c\ A''\rangle]]\ D.Skip]\ D\_incr\ Int\_absorb1\ subset\_trans)$
  thus *?case* **by** (*metis D_incr* ⟨$A = dom\ s$⟩)
next
  case *Seq2* **thus** *?case* **by** *auto* (*metis D_mono D.intros*(*3*))
qed (*auto intro*: *D.intros*)

theorem *D_sound*:
  $(c,s) \rightarrow* (c',s') \Longrightarrow D\ (dom\ s)\ c\ A'$
  $\Longrightarrow (\exists\, cs''.\ (c',s') \rightarrow cs'') \vee c' = SKIP$
apply(*induction arbitrary*: A' *rule*:*star_induct*)
apply (*metis progress*)
by (*metis D_preservation*)

end

11 Constant Folding

theory *Sem_Equiv*
imports *Big_Step*
begin

11.1 Semantic Equivalence up to a Condition

type_synonym $assn = state \Rightarrow bool$

definition
  $equiv\_up\_to :: assn \Rightarrow com \Rightarrow com \Rightarrow bool\ (\_ \models \_ \sim \_\ [50,0,10]\ 50)$
where
  $(P \models c \sim c') = (\forall\, s\ s'.\ P\ s \longrightarrow (c,s) \Rightarrow s' \longleftrightarrow (c',s) \Rightarrow s')$

definition
  $bequiv\_up\_to :: assn \Rightarrow bexp \Rightarrow bexp \Rightarrow bool\ (\_ \models \_ <\sim> \_\ [50,0,10]\ 50)$
where
  $(P \models b <\sim> b') = (\forall\, s.\ P\ s \longrightarrow bval\ b\ s = bval\ b'\ s)$

lemma *equiv_up_to_True*:
  $((\lambda\_.\ True) \models c \sim c') = (c \sim c')$
  by (*simp add*: *equiv_def equiv_up_to_def*)

lemma *equiv_up_to_weaken*:
  $P \models c \sim c' \Longrightarrow (\bigwedge s.\ P'\ s \Longrightarrow P\ s) \Longrightarrow P' \models c \sim c'$
  by (*simp add*: *equiv_up_to_def*)

lemma *equiv_up_toI*:

$(\bigwedge s\ s'.\ P\ s \Longrightarrow (c,\ s) \Rightarrow s' = (c',\ s) \Rightarrow s') \Longrightarrow P \models c \sim c'$
by (*unfold equiv_up_to_def*) *blast*

lemma *equiv_up_toD1*:
$P \models c \sim c' \Longrightarrow (c,\ s) \Rightarrow s' \Longrightarrow P\ s \Longrightarrow (c',\ s) \Rightarrow s'$
by (*unfold equiv_up_to_def*) *blast*

lemma *equiv_up_toD2*:
$P \models c \sim c' \Longrightarrow (c',\ s) \Rightarrow s' \Longrightarrow P\ s \Longrightarrow (c,\ s) \Rightarrow s'$
by (*unfold equiv_up_to_def*) *blast*

lemma *equiv_up_to_refl* [*simp*, *intro*!]:
$P \models c \sim c$
by (*auto simp*: *equiv_up_to_def*)

lemma *equiv_up_to_sym*:
$(P \models c \sim c') = (P \models c' \sim c)$
by (*auto simp*: *equiv_up_to_def*)

lemma *equiv_up_to_trans*:
$P \models c \sim c' \Longrightarrow P \models c' \sim c'' \Longrightarrow P \models c \sim c''$
by (*auto simp*: *equiv_up_to_def*)

lemma *bequiv_up_to_refl* [*simp*, *intro*!]:
$P \models b <\sim> b$
by (*auto simp*: *bequiv_up_to_def*)

lemma *bequiv_up_to_sym*:
$(P \models b <\sim> b') = (P \models b' <\sim> b)$
by (*auto simp*: *bequiv_up_to_def*)

lemma *bequiv_up_to_trans*:
$P \models b <\sim> b' \Longrightarrow P \models b' <\sim> b'' \Longrightarrow P \models b <\sim> b''$
by (*auto simp*: *bequiv_up_to_def*)

lemma *bequiv_up_to_subst*:
$P \models b <\sim> b' \Longrightarrow P\ s \Longrightarrow bval\ b\ s = bval\ b'\ s$
by (*simp add*: *bequiv_up_to_def*)

lemma *equiv_up_to_seq*:
$P \models c \sim c' \Longrightarrow Q \models d \sim d' \Longrightarrow$

```
  (⋀s s'. (c,s) ⇒ s' ⟹ P s ⟹ Q s') ⟹
  P ⊨ (c;; d) ∼ (c';; d')
  by (clarsimp simp: equiv_up_to_def) blast

lemma equiv_up_to_while_lemma_weak:
  shows (d,s) ⇒ s' ⟹
        P ⊨ b <∼> b' ⟹
        P ⊨ c ∼ c' ⟹
        (⋀s s'. (c, s) ⇒ s' ⟹ P s ⟹ bval b s ⟹ P s') ⟹
        P s ⟹
        d = WHILE b DO c ⟹
        (WHILE b' DO c', s) ⇒ s'
proof (induction rule: big_step_induct)
  case (WhileTrue b s1 c s2 s3)
  hence IH: P s2 ⟹ (WHILE b' DO c', s2) ⇒ s3 by auto
  from WhileTrue.prems
  have P ⊨ b <∼> b' by simp
  with ‹bval b s1› ‹P s1›
  have bval b' s1 by (simp add: bequiv_up_to_def)
  moreover
  from WhileTrue.prems
  have P ⊨ c ∼ c' by simp
  with ‹bval b s1› ‹P s1› ‹(c, s1) ⇒ s2›
  have (c', s1) ⇒ s2 by (simp add: equiv_up_to_def)
  moreover
  from WhileTrue.prems
  have ⋀s s'. (c,s) ⇒ s' ⟹ P s ⟹ bval b s ⟹ P s' by simp
  with ‹P s1› ‹bval b s1› ‹(c, s1) ⇒ s2›
  have P s2 by simp
  hence (WHILE b' DO c', s2) ⇒ s3 by (rule IH)
  ultimately
  show ?case by blast
next
  case WhileFalse
  thus ?case by (auto simp: bequiv_up_to_def)
qed (fastforce simp: equiv_up_to_def bequiv_up_to_def)+

lemma equiv_up_to_while_weak:
  assumes b: P ⊨ b <∼> b'
  assumes c: P ⊨ c ∼ c'
  assumes I: ⋀s s'. (c, s) ⇒ s' ⟹ P s ⟹ bval b s ⟹ P s'
  shows P ⊨ WHILE b DO c ∼ WHILE b' DO c'
proof -
  from b have b': P ⊨ b' <∼> b by (simp add: bequiv_up_to_sym)
```

from c b **have** c': $P \models c' \sim c$ **by** $(simp\ add{:}\ equiv\_up\_to\_sym)$

from I
have I': $\bigwedge s\ s'.\ (c',\ s) \Rightarrow s' \Longrightarrow P\ s \Longrightarrow bval\ b'\ s \Longrightarrow P\ s'$
by $(auto\ dest!{:}\ equiv\_up\_toD1\ [OF\ c']\ simp{:}\ bequiv\_up\_to\_subst\ [OF\ b'])$

note $equiv\_up\_to\_while\_lemma\_weak\ [OF\ \_\ b\ c]$
$equiv\_up\_to\_while\_lemma\_weak\ [OF\ \_\ b'\ c']$
thus $?thesis$ **using** $I\ I'$ **by** $(auto\ intro!{:}\ equiv\_up\_toI)$
qed

lemma $equiv\_up\_to\_if\_weak$:
$P \models b <\sim> b' \Longrightarrow P \models c \sim c' \Longrightarrow P \models d \sim d' \Longrightarrow$
$P \models IF\ b\ THEN\ c\ ELSE\ d \sim IF\ b'\ THEN\ c'\ ELSE\ d'$
by $(auto\ simp{:}\ bequiv\_up\_to\_def\ equiv\_up\_to\_def)$

lemma $equiv\_up\_to\_if\_True\ [intro!]$:
$(\bigwedge s.\ P\ s \Longrightarrow bval\ b\ s) \Longrightarrow P \models IF\ b\ THEN\ c1\ ELSE\ c2 \sim c1$
by $(auto\ simp{:}\ equiv\_up\_to\_def)$

lemma $equiv\_up\_to\_if\_False\ [intro!]$:
$(\bigwedge s.\ P\ s \Longrightarrow \neg\ bval\ b\ s) \Longrightarrow P \models IF\ b\ THEN\ c1\ ELSE\ c2 \sim c2$
by $(auto\ simp{:}\ equiv\_up\_to\_def)$

lemma $equiv\_up\_to\_while\_False\ [intro!]$:
$(\bigwedge s.\ P\ s \Longrightarrow \neg\ bval\ b\ s) \Longrightarrow P \models WHILE\ b\ DO\ c \sim SKIP$
by $(auto\ simp{:}\ equiv\_up\_to\_def)$

lemma $while\_never$: $(c,\ s) \Rightarrow u \Longrightarrow c \neq WHILE\ (Bc\ True)\ DO\ c'$
by $(induct\ rule{:}\ big\_step\_induct)\ auto$

lemma $equiv\_up\_to\_while\_True\ [intro!,simp]$:
$P \models WHILE\ Bc\ True\ DO\ c \sim WHILE\ Bc\ True\ DO\ SKIP$
unfolding $equiv\_up\_to\_def$
by $(blast\ dest{:}\ while\_never)$

end
theory *Fold* **imports** *Sem_Equiv Vars* **begin**

11.2 Simple folding of arithmetic expressions

type_synonym

$tab = vname \Rightarrow val\ option$

fun *afold* :: *aexp* ⇒ *tab* ⇒ *aexp* **where**
afold (*N n*) _ = *N n* |
afold (*V x*) *t* = (*case t x of None* ⇒ *V x* | *Some k* ⇒ *N k*) |
afold (*Plus e1 e2*) *t* = (*case* (*afold e1 t*, *afold e2 t*) *of*
(*N n1*, *N n2*) ⇒ *N*(*n1*+*n2*) | (*e1'*,*e2'*) ⇒ *Plus e1' e2'*)

definition *approx t s* ⟷ (∀ *x k*. *t x* = *Some k* ⟶ *s x* = *k*)

theorem *aval_afold*[*simp*]:
assumes *approx t s*
shows *aval* (*afold a t*) *s* = *aval a s*
using *assms*
by (*induct a*) (*auto simp*: *approx_def split*: *aexp.split option.split*)

theorem *aval_afold_N*:
assumes *approx t s*
shows *afold a t* = *N n* ⟹ *aval a s* = *n*
by (*metis assms aval.simps*(*1*) *aval_afold*)

definition
merge t1 t2 = (λ*m*. *if t1 m* = *t2 m then t1 m else None*)

primrec *defs* :: *com* ⇒ *tab* ⇒ *tab* **where**
defs SKIP t = *t* |
defs (*x* ::= *a*) *t* =
(*case afold a t of N k* ⇒ *t*(*x* ↦ *k*) | _ ⇒ *t*(*x*:=*None*)) |
defs (*c1*;;*c2*) *t* = (*defs c2 o defs c1*) *t* |
defs (*IF b THEN c1 ELSE c2*) *t* = *merge* (*defs c1 t*) (*defs c2 t*) |
defs (*WHILE b DO c*) *t* = *t* |‘ (−*lvars c*)

primrec *fold* **where**
fold SKIP _ = *SKIP* |
fold (*x* ::= *a*) *t* = (*x* ::= (*afold a t*)) |
fold (*c1*;;*c2*) *t* = (*fold c1 t*;; *fold c2* (*defs c1 t*)) |
fold (*IF b THEN c1 ELSE c2*) *t* = *IF b THEN fold c1 t ELSE fold c2 t* |
fold (*WHILE b DO c*) *t* = *WHILE b DO fold c* (*t* |‘ (−*lvars c*))

lemma *approx_merge*:
approx t1 s ∨ *approx t2 s* ⟹ *approx* (*merge t1 t2*) *s*
by (*fastforce simp*: *merge_def approx_def*)

lemma *approx_map_le*:

approx t2 s $\Longrightarrow$ *t1* $\subseteq_m$ *t2* $\Longrightarrow$ *approx t1 s*
by (*clarsimp simp*: *approx_def map_le_def dom_def*)

lemma *restrict_map_le* [*intro*!, *simp*]: *t* |' *S* $\subseteq_m$ *t*
by (*clarsimp simp*: *restrict_map_def map_le_def*)

lemma *merge_restrict*:
assumes *t1* |' *S* = *t* |' *S*
assumes *t2* |' *S* = *t* |' *S*
shows *merge t1 t2* |' *S* = *t* |' *S*
proof −
from *assms*
have $\forall x$. (*t1* |' *S*) *x* = (*t* |' *S*) *x*
and $\forall x$. (*t2* |' *S*) *x* = (*t* |' *S*) *x* **by** *auto*
thus *?thesis*
by (*auto simp*: *merge_def restrict_map_def*
split: *if_splits*)
qed

lemma *defs_restrict*:
defs c t |' (− *lvars c*) = *t* |' (− *lvars c*)
proof (*induction c arbitrary*: *t*)
case (*Seq c1 c2*)
hence *defs c1 t* |' (− *lvars c1*) = *t* |' (− *lvars c1*)
by *simp*
hence *defs c1 t* |' (− *lvars c1*) |' (−*lvars c2*) =
t |' (− *lvars c1*) |' (−*lvars c2*) **by** *simp*
moreover
from *Seq*
have *defs c2* (*defs c1 t*) |' (− *lvars c2*) =
defs c1 t |' (− *lvars c2*)
by *simp*
hence *defs c2* (*defs c1 t*) |' (− *lvars c2*) |' (− *lvars c1*) =
defs c1 t |' (− *lvars c2*) |' (− *lvars c1*)
by *simp*
ultimately
show *?case* **by** (*clarsimp simp*: *Int_commute*)
next
case (*If b c1 c2*)
hence *defs c1 t* |' (− *lvars c1*) = *t* |' (− *lvars c1*) **by** *simp*
hence *defs c1 t* |' (− *lvars c1*) |' (−*lvars c2*) =
t |' (− *lvars c1*) |' (−*lvars c2*) **by** *simp*
moreover

from *If*
have *defs c2 t* $|`$ $(-$ *lvars c2*$)$ $=$ *t* $|`$ $(-$ *lvars c2*$)$ **by** *simp*
hence *defs c2 t* $|`$ $(-$ *lvars c2*$)$ $|`$ $(-$*lvars c1*$)$ $=$
t $|`$ $(-$ *lvars c2*$)$ $|`$ $(-$*lvars c1*$)$ **by** *simp*
ultimately
show *?case* **by** (*auto simp*: *Int_commute intro*: *merge_restrict*)
qed (*auto split*: *aexp.split*)

lemma *big_step_pres_approx*:
$(c,s) \Rightarrow s' \Longrightarrow$ *approx t s* $\Longrightarrow$ *approx* (*defs c t*) s'
proof (*induction arbitrary*: *t rule*: *big_step_induct*)
case *Skip* **thus** *?case* **by** *simp*
next
case *Assign*
thus *?case*
by (*clarsimp simp*: *aval_afold_N approx_def split*: *aexp.split*)
next
case (*Seq c1 s1 s2 c2 s3*)
have *approx* (*defs c1 t*) *s2* **by** (*rule Seq.IH*(*1*)[*OF Seq.prems*])
hence *approx* (*defs c2* (*defs c1 t*)) *s3* **by** (*rule Seq.IH*(*2*))
thus *?case* **by** *simp*
next
case (*IfTrue b s c1* s')
hence *approx* (*defs c1 t*) s' **by** *simp*
thus *?case* **by** (*simp add*: *approx_merge*)
next
case (*IfFalse b s c2* s')
hence *approx* (*defs c2 t*) s' **by** *simp*
thus *?case* **by** (*simp add*: *approx_merge*)
next
case *WhileFalse*
thus *?case* **by** (*simp add*: *approx_def restrict_map_def*)
next
case (*WhileTrue b s1 c s2 s3*)
hence *approx* (*defs c t*) *s2* **by** *simp*
with *WhileTrue*
have *approx* (*defs c t* $|`$ $(-$*lvars c*$)$) *s3* **by** *simp*
thus *?case* **by** (*simp add*: *defs_restrict*)
qed

lemma *big_step_pres_approx_restrict*:
$(c,s) \Rightarrow s' \Longrightarrow$ *approx* (*t* $|`$ $(-$*lvars c*$)$) *s* $\Longrightarrow$ *approx* (*t* $|`$ $(-$*lvars c*$)$) s'

```
proof (induction arbitrary: t rule: big_step_induct)
  case Assign
  thus ?case by (clarsimp simp: approx_def)
next
  case (Seq c1 s1 s2 c2 s3)
  hence approx (t |` (−lvars c2) |` (−lvars c1)) s1
    by (simp add: Int_commute)
  hence approx (t |` (−lvars c2) |` (−lvars c1)) s2
    by (rule Seq)
  hence approx (t |` (−lvars c1) |` (−lvars c2)) s2
    by (simp add: Int_commute)
  hence approx (t |` (−lvars c1) |` (−lvars c2)) s3
    by (rule Seq)
  thus ?case by simp
next
  case (IfTrue b s c1 s′ c2)
  hence approx (t |` (−lvars c2) |` (−lvars c1)) s
    by (simp add: Int_commute)
  hence approx (t |` (−lvars c2) |` (−lvars c1)) s′
    by (rule IfTrue)
  thus ?case by (simp add: Int_commute)
next
  case (IfFalse b s c2 s′ c1)
  hence approx (t |` (−lvars c1) |` (−lvars c2)) s
    by simp
  hence approx (t |` (−lvars c1) |` (−lvars c2)) s′
    by (rule IfFalse)
  thus ?case by simp
qed auto


declare assign_simp [simp]

lemma approx_eq:
  approx t ⊨ c ∼ fold c t
proof (induction c arbitrary: t)
  case SKIP show ?case by simp
next
  case Assign
  show ?case by (simp add: equiv_up_to_def)
next
  case Seq
  thus ?case by (auto intro!: equiv_up_to_seq big_step_pres_approx)
next
```

case *If*
thus *?case* **by** (*auto intro*!: *equiv_up_to_if_weak*)
next
case (*While b c*)
hence *approx* (t |' ($-$ *lvars c*)) $\models$
WHILE b DO c $\sim$ *WHILE b DO fold c* (t |' ($-$ *lvars c*))
by (*auto intro*: *equiv_up_to_while_weak big_step_pres_approx_restrict*)
thus *?case*
by (*auto intro*: *equiv_up_to_weaken approx_map_le*)
qed

lemma *approx_empty* [*simp*]:
approx Map.empty = (λ_. *True*)
by (*auto simp*: *approx_def*)

theorem *constant_folding_equiv*:
fold c Map.empty $\sim$ *c*
using *approx_eq* [*of Map.empty c*]
by (*simp add*: *equiv_up_to_True sim_sym*)

end

12 Live Variable Analysis

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

12.1 Liveness Analysis

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

$kill\ SKIP = \{\}$ |
$kill\ (x ::= a) = \{x\}$ |
$kill\ (c_1;;\ c_2) = kill\ c_1 \cup kill\ c_2$ |
$kill\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) = kill\ c_1 \cap kill\ c_2$ |
$kill\ (WHILE\ b\ DO\ c) = \{\}$

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

lemma *L_gen_kill*: $L\ c\ X = gen\ c \cup (X - kill\ c)$
by(*induct c arbitrary*:*X*) *auto*

lemma *L_While_pfp*: $L\ c\ (L\ (WHILE\ b\ DO\ c)\ X) \subseteq L\ (WHILE\ b\ DO\ c)$
X
by(*auto simp add*:*L_gen_kill*)

lemma *L_While_lpfp*:
  $vars\ b \cup X \cup L\ c\ P \subseteq P \Longrightarrow L\ (WHILE\ b\ DO\ c)\ X \subseteq P$
by(*simp add*: *L_gen_kill*)

lemma *L_While_vars*: $vars\ b \subseteq L\ (WHILE\ b\ DO\ c)\ X$
by *auto*

lemma *L_While_X*: $X \subseteq L\ (WHILE\ b\ DO\ c)\ X$
by *auto*

Disable L WHILE equation and reason only with L WHILE constraints

declare *L.simps*(*5*)[*simp del*]

12.2 Correctness

theorem *L_correct*:
  $(c,s) \Rightarrow s' \Longrightarrow s = t\ on\ L\ c\ X \Longrightarrow$
  $\exists\ t'.\ (c,t) \Rightarrow t'\ \&\ s' = t'\ on\ X$
proof (*induction arbitrary*: *X t rule*: *big_step_induct*)
  case *Skip* **then show** *?case* **by** *auto*
next
  case *Assign* **then show** *?case*
    by (*auto simp*: *ball_Un*)
next

```
  case (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 (metis L_While_vars bval_eq_if_eq_on_vars set_mp)
   thus ?case by(metis WhileFalse.prems L_While_X big_step.WhileFalse
set_mp)
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_vars bval_eq_if_eq_on_vars set_mp)
  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
```

12.3 Program Optimization

Burying assignments to dead variables:

fun *bury* :: *com* ⇒ *vname set* ⇒ *com* **where**
bury SKIP X = *SKIP* |
bury (x ::= a) X = (*if* $x \in X$ *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 (L (*WHILE* b *DO* c) X)

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

theorem *bury_correct*:
  $(c,s) \Rightarrow s' \Longrightarrow s = t$ *on* L c $X \Longrightarrow$
  $\exists\ t'.\ (bury\ c\ X,t) \Rightarrow t'\ \&\ s' = t'$ *on* X
proof (*induction arbitrary*: *X t rule*: *big_step_induct*)
  case *Skip* **then show** *?case* **by** *auto*
next
  case *Assign* **then show** *?case*
    by (*auto simp*: *ball_Un*)
next
  case (*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 (metis L_While_vars bval_eq_if_eq_on_vars set_mp)
  thus ?case
   by simp (metis L_While_X WhileFalse.prems big_step.WhileFalse set_mp)
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_vars bval_eq_if_eq_on_vars set_mp)
  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_correct: (c,s) ⇒ s′ ⟹ (bury c UNIV,s) ⇒ s′
using bury_correct[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 | (∃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]: bc₁;;bc₂ = bury c X ⟷
  (∃c₁ c₂. c = c₁;;c₂ & bc₂ = bury c₂ X & bc₁ = bury c₁ (L c₂ X))
by (cases c) auto

lemma If_bury[simp]: IF b THEN bc1 ELSE bc2 = bury c X ⟷
  (∃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 ⟷
  (∃c′. c = WHILE b DO c′ & bc′ = bury c′ (L (WHILE b DO c′) X))
by (cases c) auto
```

```
theorem bury_correct2:
  (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 (metis L_While_vars bval_eq_if_eq_on_vars set_rev_mp)
```

thus *?case* **using** *WhileFalse*
by *auto* (*metis L_While_X big_step.WhileFalse set_mp*)
next
case (*WhileTrue b s1 bc′ s2 s3 w X t1*)
then obtain *c′* **where** *w*: *w* = *WHILE b DO c′*
and *bc′*: *bc′* = *bury c′* (*L* (*WHILE b DO c′*) *X*) **by** *auto*
from ‹*bval b s1*› *WhileTrue.prems w* **have** *bval b t1*
by *auto* (*metis L_While_vars bval_eq_if_eq_on_vars set_mp*)
note *IH* = *WhileTrue.hyps*(*3*,*5*)
have *s1* = *t1 on L c′* (*L w X*)
using *L_While_pfp WhileTrue.prems w* **by** *blast*
with *IH*(*1*)[*OF bc′*, *of t1*] *w* **obtain** *t2* **where**
(*c′*, *t1*) ⇒ *t2 s2* = *t2 on L w X* **by** *auto*
from *IH*(*2*)[*OF WhileTrue.hyps*(*6*), *of t2*] *w this*(*2*) **obtain** *t3*
where (*w*,*t2*) ⇒ *t3 s3* = *t3 on X*
by *auto*
with ‹*bval b t1*› ‹(*c′*, *t1*) ⇒ *t2*› *w* **show** *?case* **by** *auto*
qed

corollary *final_bury_correct2*: (*bury c UNIV*,*s*) ⇒ *s′* ⟹ (*c*,*s*) ⇒ *s′*
using *bury_correct2*[*of c UNIV*]
by (*auto simp*: *fun_eq_iff*[*symmetric*])

corollary *bury_sim*: *bury c UNIV* ∼ *c*
by(*metis final_bury_correct final_bury_correct2*)

end

theory *Live_True*
imports *HOL−Library.While_Combinator Vars Big_Step*
begin

12.4 True Liveness Analysis

fun *L* :: *com* ⇒ *vname set* ⇒ *vname set* **where**
L SKIP X = *X* |
L (*x* ::= *a*) *X* = (*if x* ∈ *X then vars a* ∪ (*X* − {*x*}) *else X*) |
L (c_1;; c_2) *X* = *L* c_1 (*L* c_2 *X*) |
L (*IF b THEN* c_1 *ELSE* c_2) *X* = *vars b* ∪ *L* c_1 *X* ∪ *L* c_2 *X* |
L (*WHILE b DO c*) *X* = *lfp*(λ*Y*. *vars b* ∪ *X* ∪ *L c Y*)

lemma *L_mono*: *mono* (*L c*)
proof−

have $X \subseteq Y \Longrightarrow L\ c\ X \subseteq L\ c\ Y$ **for** $X\ Y$
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 ($\lambda Y.\ X \cup L\ c\ Y$)
by (*metis* (*no_types*) *L_mono mono_def order_eq_iff set_eq_subset sup_mono*)

lemma *L_While_unfold*:
L (*WHILE b DO c*) X = *vars* $b \cup X \cup L\ c$ (L (*WHILE b DO c*) X)
by(*metis lfp_unfold*[*OF mono_union_L*] *L.simps*(*5*))

lemma *L_While_pfp*: $L\ c$ (L (*WHILE b DO c*) X) $\subseteq L$ (*WHILE b DO c*) X
using *L_While_unfold* **by** *blast*

lemma *L_While_vars*: *vars* $b \subseteq L$ (*WHILE b DO c*) X
using *L_While_unfold* **by** *blast*

lemma *L_While_X*: $X \subseteq L$ (*WHILE b DO c*) X
using *L_While_unfold* **by** *blast*

Disable *L WHILE* equation and reason only with *L WHILE* constraints:

declare *L.simps*(*5*)[*simp del*]

12.5 Correctness

theorem *L_correct*:
$(c,s) \Rightarrow s' \Longrightarrow s = t$ *on* $L\ c\ X \Longrightarrow$
$\exists\ t'.\ (c,t) \Rightarrow t'\ \&\ s' = t'$ *on* X
proof (*induction arbitrary*: $X\ t$ *rule*: *big_step_induct*)
case *Skip* **then show** *?case* **by** *auto*
next
case *Assign* **then show** *?case*

```
    by (auto simp: ball_Un)
next
  case (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_vars bval_eq_if_eq_on_vars set_mp)
  thus ?case using WhileFalse.prems L_While_X[of X b c] 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_vars bval_eq_if_eq_on_vars set_mp)
  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
```

12.6 Executability

```
lemma L_subset_vars: L c X ⊆ rvars c ∪ X
proof(induction c arbitrary: X)
  case (While b c)
  have lfp(λY. vars b ∪ X ∪ L c Y) ⊆ vars b ∪ rvars c ∪ X
    using While.IH[of vars b ∪ rvars c ∪ X]
    by (auto intro!: lfp_lowerbound)
  thus ?case by (simp add: L.simps(5))
qed auto
```

Make L executable by replacing *lfp* with the *while* combinator from theory *HOL−Library.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 == λY. vars b ∪ X ∪ L c Y
shows L (WHILE b DO c) X = while (λY. f Y ≠ Y) f {} (is _ = ?r)
proof −
  let ?V = vars b ∪ rvars 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 L.simps(5))
qed
```

```
lemma L_While_let: finite X ⟹ L (WHILE b DO c) X =
  (let f = (λY. vars b ∪ X ∪ L c Y)
   in while (λY. f Y ≠ Y) f {})
by(simp add: L_While)
```

```
lemma L_While_set: L (WHILE b DO c) (set xs) =
  (let f = (λY. vars b ∪ set xs ∪ L c Y)
   in while (λY. f Y ≠ Y) f {})
by(rule L_While_let, simp)
```

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.

A test:

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*

12.7 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* (λ*A*. *vars b* ∪ *X* ∪ *Lb c A*) *2* {} (*vars b* ∪ *rvars 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* = λ*A*. *vars b* ∪ *X* ∪ *L c A*
  let *?fb* = λ*A*. *vars b* ∪ *X* ∪ *Lb c A*
  show *?case*

proof (*simp add*: *L.simps*(*5*), *rule lfp_subset_iter*[*OF mono_union_L*])
show !!*X*. *?f X* ⊆ *?fb X* **using** *While.IH* **by** *blast*
show *lfp ?f* ⊆ *vars b* ∪ *rvars c* ∪ *X*
by (*metis* (*full_types*) *L.simps*(*5*) *L_subset_vars rvars.simps*(*5*))
qed
next
case *Seq* **thus** *?case* **by** *simp* (*metis* (*full_types*) *L_mono monoD subset_trans*)
qed *auto*

end

13 Hoare Logic

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

13.1 Hoare Logic for Partial Correctness

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

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

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: ⊢ {λ*s*. *P*(*s*[*a*/*x*])} *x*::=*a* {*P*} |

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

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

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

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

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

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

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

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

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

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

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

end

theory *Hoare_Examples* **imports** *Hoare* **begin**

hide_const (**open**) *sum*

Summing up the first x natural numbers in variable y.

fun *sum* :: *int* ⇒ *int* **where**
sum i = (*if i* ≤ *0 then 0 else sum* (*i* − *1*) + *i*)

lemma *sum_simps*[*simp*]:
$0 < i \Longrightarrow sum\ i = sum\ (i - 1) + i$
$i \le 0 \Longrightarrow sum\ i = 0$
by(*simp_all*)

declare *sum.simps*[*simp del*]

abbreviation *wsum* ==
  WHILE Less (*N 0*) (*V* ''*x*'')
  DO (''*y*'' ::= *Plus* (*V* ''*y*'') (*V* ''*x*'');;
      ''*x*'' ::= *Plus* (*V* ''*x*'') (*N* (− *1*)))

13.1.1 Proof by Operational Semantics

The behaviour of the loop is proved by induction:

lemma *while_sum*:
  (*wsum*, *s*) ⇒ *t* ⟹ *t* ''*y*'' = *s* ''*y*'' + *sum*(*s* ''*x*'')
apply(*induction wsum s t rule*: *big_step_induct*)
apply(*auto*)
done

We were lucky that the proof was 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 (''*y*'' ::= *N 0*;; *wsum*, *s*) ⇒ *t*
  shows *t* ''*y*'' = *sum* (*s* ''*x*'')
proof −
  from *assms* **have** (*wsum*,*s*(''*y*'':=*0*)) ⇒ *t* **by** *auto*
  from *while_sum*[*OF this*] **show** *?thesis* **by** *simp*
qed

13.1.2 Proof by Hoare Logic

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

lemma ⊢ {λ*s*. *s* ''*x*'' = *n*} ''*y*'' ::= *N 0*;; *wsum* {λ*s*. *s* ''*y*'' = *sum n*}
apply(*rule Seq*)
 prefer *2*
 apply(*rule While'* [**where** *P* = λ*s*. (*s* ''*y*'' = *sum n* − *sum*(*s* ''*x*''))])
  apply(*rule Seq*)
   prefer *2*
   apply(*rule Assign*)
  apply(*rule Assign'*)
  apply *simp*
 apply *simp*
apply(*rule Assign'*)
apply *simp*

done

The proof is intentionally an apply script 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

13.2 Soundness and Completeness

theory *Hoare_Sound_Complete*
imports *Hoare*
begin

13.2.1 Soundness

lemma *hoare_sound*: $\vdash \{P\}c\{Q\} \implies \models \{P\}c\{Q\}$
proof(*induction rule*: *hoare.induct*)
  case (*While P b c*)
  have $(\mathit{WHILE}\ b\ \mathit{DO}\ c,s) \Rightarrow t \implies P\ s \implies P\ t \wedge \neg\ \mathit{bval}\ b\ t$ **for** *s 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.IH* **unfolding** *hoare_valid_def* **by** *blast*
  qed
  thus *?case* **unfolding** *hoare_valid_def* **by** *blast*
qed (*auto simp*: *hoare_valid_def*)

13.2.2 Weakest Precondition

definition *wp* :: *com* ⇒ *assn* ⇒ *assn* **where**
$wp\ c\ Q = (\lambda s.\ \forall t.\ (c,s) \Rightarrow t \longrightarrow Q\ t)$

lemma *wp_SKIP*[*simp*]: $wp\ \mathit{SKIP}\ Q = Q$
by (*rule ext*) (*auto simp*: *wp_def*)

lemma *wp_Ass*[*simp*]: $wp\ (x{::=}a)\ Q = (\lambda s.\ Q(s[a/x]))$
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 =
(λs. *if bval* b s *then wp* c_1 Q s *else 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 $\Longrightarrow$
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*]: $\neg$ *bval* b s $\Longrightarrow$ *wp* (*WHILE* b *DO* c) Q s =
Q s
by(*simp add*: *wp_While_If*)

13.2.3 Completeness

lemma *wp_is_pre*: $\vdash \{wp\ c\ Q\}\ c\ \{Q\}$
proof(*induction* c *arbitrary*: Q)
  case *If* **thus** *?case* **by**(*auto intro*: *conseq*)
next
  case (*While* b c)
  let *?w* = *WHILE* b *DO* c
  show $\vdash \{wp\ ?w\ Q\}\ ?w\ \{Q\}$
  proof(*rule While'*)
    show $\vdash \{\lambda s.\ wp\ ?w\ Q\ s \wedge bval\ b\ s\}\ c\ \{wp\ ?w\ Q\}$
    proof(*rule strengthen_pre*[*OF* _ *While.IH*])
      show $\forall s.\ wp\ ?w\ Q\ s \wedge bval\ b\ s \longrightarrow wp\ c\ (wp\ ?w\ Q)\ s$ **by** *auto*
    qed
    show $\forall s.\ wp\ ?w\ Q\ s \wedge \neg\ bval\ b\ s \longrightarrow Q\ s$ **by** *auto*
  qed
qed *auto*

lemma *hoare_complete*: **assumes** $\models \{P\}c\{Q\}$ **shows** $\vdash \{P\}c\{Q\}$
proof(*rule strengthen_pre*)
  show $\forall s.\ P\ s \longrightarrow wp\ c\ Q\ s$ **using** *assms*
    by (*auto simp*: *hoare_valid_def wp_def*)
  show $\vdash \{wp\ c\ Q\}\ c\ \{Q\}$ **by**(*rule wp_is_pre*)
qed

corollary *hoare_sound_complete*: $\vdash \{P\}c\{Q\} \longleftrightarrow\ \models \{P\}c\{Q\}$

by (*metis hoare_complete hoare_sound*)

end

theory *VCG* **imports** *Hoare* **begin**

13.3 Verification Conditions

Annotated commands: commands where loops are annotated with invariants.

datatype *acom* =
  Askip   (*SKIP*) |
  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*)

notation *com.SKIP* (*SKIP*)

Strip annotations:

fun *strip* :: *acom* ⇒ *com* **where**
strip SKIP = *SKIP* |
strip (*x* ::= *a*) = (*x* ::= *a*) |
strip (C_1;; C_2) = (*strip* C_1;; *strip* C_2) |
strip (*IF b THEN* C_1 *ELSE* C_2) = (*IF b THEN strip* C_1 *ELSE strip* C_2) |
strip ({_} *WHILE b DO C*) = (*WHILE b DO strip C*)

Weakest precondition from annotated commands:

fun *pre* :: *acom* ⇒ *assn* ⇒ *assn* **where**
pre SKIP Q = *Q* |
pre (*x* ::= *a*) *Q* = (λ*s*. *Q*(*s*(*x* := *aval a s*))) |
pre (C_1;; C_2) *Q* = *pre* C_1 (*pre* C_2 *Q*) |
pre (*IF b THEN* C_1 *ELSE* C_2) *Q* =
  (λ*s*. *if bval b s then pre* C_1 *Q s else pre* C_2 *Q s*) |
pre ({*I*} *WHILE b DO C*) *Q* = *I*

Verification condition:

fun *vc* :: *acom* ⇒ *assn* ⇒ *bool* **where**
vc SKIP Q = *True* |
vc (*x* ::= *a*) *Q* = *True* |
vc (C_1;; C_2) *Q* = (*vc* C_1 (*pre* C_2 *Q*) ∧ *vc* C_2 *Q*) |
vc (*IF b THEN* C_1 *ELSE* C_2) *Q* = (*vc* C_1 *Q* ∧ *vc* C_2 *Q*) |
vc ({*I*} *WHILE b DO C*) *Q* =

$((\forall\, s.\ (I\ s \wedge bval\ b\ s \longrightarrow pre\ C\ I\ s) \wedge$
$\quad (I\ s \wedge \neg\ bval\ b\ s \longrightarrow Q\ s)) \wedge$
$vc\ C\ I)$

Soundness:

lemma vc_sound: $vc\ C\ Q \Longrightarrow \vdash \{pre\ C\ Q\}\ strip\ C\ \{Q\}$
proof$(induction\ C\ arbitrary{:}\ Q)$
case $(Awhile\ I\ b\ C)$
show $?case$
proof$(simp,\ rule\ While')$
from $\langle vc\ (Awhile\ I\ b\ C)\ Q\rangle$
have vc: $vc\ C\ I$ **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':
$[\![\ vc\ C\ Q;\ \forall\, s.\ P\ s \longrightarrow pre\ C\ Q\ s\ ]\!] \Longrightarrow \vdash \{P\}\ strip\ C\ \{Q\}$
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:
$\forall\, s.\ P\ s \longrightarrow P'\ s \Longrightarrow vc\ C\ P \Longrightarrow vc\ C\ P'$
proof$(induction\ C\ arbitrary{:}\ P\ P')$
case $Aseq$ **thus** $?case$ **by** $simp\ (metis\ pre\_mono)$
qed $simp\_all$

lemma $vc\_complete$:
$\vdash \{P\}c\{Q\} \Longrightarrow \exists\, C.\ strip\ C = c \wedge vc\ C\ Q \wedge (\forall\, s.\ P\ s \longrightarrow pre\ C\ Q\ s)$
(**is** $\_ \Longrightarrow \exists\, C.\ ?G\ P\ c\ Q\ C)$
proof $(induction\ rule{:}\ hoare.induct)$
case $Skip$
show $?case$ (**is** $\exists\, C.\ ?C\ C)$
proof show $?C\ Askip$ **by** $simp$ **qed**

```
next
  case (Assign P a x)
  show ?case (is ∃ C. ?C C)
  proof show ?C(Aassign x a) by simp qed
next
  case (Seq P c1 Q c2 R)
  from Seq.IH obtain C1 where ih1: ?G P c1 Q C1 by blast
  from Seq.IH obtain C2 where ih2: ?G Q c2 R C2 by blast
  show ?case (is ∃ C. ?C C)
  proof
    show ?C(Aseq C1 C2)
      using ih1 ih2 by (fastforce elim!: pre_mono vc_mono)
  qed
next
  case (If P b c1 Q c2)
  from If.IH obtain C1 where ih1: ?G (λs. P s ∧ bval b s) c1 Q C1
    by blast
  from If.IH obtain C2 where ih2: ?G (λs. P s ∧ ¬bval b s) c2 Q C2
    by blast
  show ?case (is ∃ C. ?C C)
  proof
    show ?C(Aif b C1 C2) using ih1 ih2 by simp
  qed
next
  case (While P b c)
  from While.IH obtain C where ih: ?G (λs. P s ∧ bval b s) c P C by
blast
  show ?case (is ∃ C. ?C C)
  proof show ?C(Awhile P b C) using ih by simp qed
next
  case conseq thus ?case by(fast elim!: pre_mono vc_mono)
qed

end
```

13.4 Hoare Logic for Total Correctness

```
theory Hoare_Total
imports Hoare_Examples
begin
```

13.4.1 Hoare Logic for Total Correctness — Separate Termination Relation

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* $\Rightarrow$ *com* $\Rightarrow$ *assn* $\Rightarrow$ *bool*
  ($\models_t$ {(*1*_)}/ (_)/ {(*1*_)} *50*) **where**
$\models_t \{P\}c\{Q\} \longleftrightarrow (\forall s.\ P\ s \longrightarrow (\exists t.\ (c,s) \Rightarrow t \wedge Q\ t))$

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

inductive
  hoaret :: *assn* $\Rightarrow$ *com* $\Rightarrow$ *assn* $\Rightarrow$ *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:
  $(\bigwedge n{::}nat.$
    $\vdash_t \{\lambda s.\ P\ s \wedge bval\ b\ s \wedge T\ s\ n\}\ c\ \{\lambda s.\ P\ s \wedge (\exists n'{<}n.\ T\ s\ n')\})$
  $\Longrightarrow \vdash_t \{\lambda s.\ P\ s \wedge (\exists n.\ T\ s\ n)\}\ WHILE\ b\ DO\ c\ \{\lambda s.\ P\ s \wedge \neg bval\ b\ s\}$ |

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 relation T :: *state* $\Rightarrow$ *nat* $\Rightarrow$ *bool* decreases. The following functional version is more intuitive:

lemma *While_fun*:
  $[\![\bigwedge n{::}nat.\ \vdash_t \{\lambda s.\ P\ s \wedge bval\ b\ s \wedge n = f\ s\}\ c\ \{\lambda s.\ P\ s \wedge f\ s < n\}]\!]$
  $\Longrightarrow \vdash_t \{P\}\ WHILE\ b\ DO\ c\ \{\lambda s.\ P\ s \wedge \neg bval\ b\ s\}$
  by (*rule While* [**where** $T{=}\lambda s\ n.\ n = f\ s$, *simplified*])

Building in the consequence rule:

lemma *strengthen_pre*:

$\llbracket\ \forall s.\ P'\ s \longrightarrow P\ s;\ \ \vdash_t \{P\}\ c\ \{Q\}\ \rrbracket \Longrightarrow \vdash_t \{P'\}\ c\ \{Q\}$
by (*metis conseq*)

lemma *weaken_post*:
$\llbracket \vdash_t \{P\}\ c\ \{Q\};\ \ \forall s.\ Q\ s \longrightarrow Q'\ s\ \rrbracket \Longrightarrow\ \vdash_t \{P\}\ c\ \{Q'\}$
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_fun'*:
assumes $\bigwedge n{::}nat.\ \vdash_t \{\lambda s.\ P\ s \wedge bval\ b\ s \wedge n = f\ s\}\ c\ \{\lambda s.\ P\ s \wedge f\ s < n\}$
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_fun assms*(*2*)])

Our standard example:

lemma $\vdash_t \{\lambda s.\ s\ ''x'' = i\}\ ''y'' ::= N\ 0;;\ wsum\ \{\lambda s.\ s\ ''y'' = sum\ i\}$
apply(*rule Seq*)
prefer *2*
apply(*rule While_fun'* [**where** $P = \lambda s.\ (s\ ''y'' = sum\ i\ -\ sum(s\ ''x''))$
and $f = \lambda s.\ nat(s\ ''x'')$])
apply(*rule Seq*)
prefer *2*
apply(*rule Assign*)
apply(*rule Assign'*)
apply *simp*
apply(*simp*)
apply(*rule Assign'*)
apply *simp*
done

The soundness theorem:

theorem *hoaret_sound*: $\vdash_t \{P\}c\{Q\}\ \ \Longrightarrow\ \ \models_t \{P\}c\{Q\}$
proof(*unfold hoare_tvalid_def*, *induction rule*: *hoaret.induct*)
case (*While P b T c*)
have $\llbracket\ P\ s;\ T\ s\ n\ \rrbracket \Longrightarrow \exists t.\ (WHILE\ b\ DO\ c,\ s) \Rightarrow t \wedge P\ t \wedge \neg\ bval\ b\ t$
for *s n*
proof(*induction n arbitrary*: *s rule*: *less_induct*)
case (*less n*) **thus** *?case* **by** (*metis While.IH WhileFalse WhileTrue*)
qed
thus *?case* **by** *auto*
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* $\Rightarrow$ *assn* $\Rightarrow$ *assn* (wp_t) **where**
$wp_t\ c\ Q\ =\ (\lambda s.\ \exists\, t.\ (c,s) \Rightarrow t \wedge Q\ t)$

lemma [*simp*]: wp_t *SKIP* $Q = Q$
by(*auto intro*!: *ext simp*: *wpt_def*)

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

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

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

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

inductive *Its* :: *bexp* $\Rightarrow$ *com* $\Rightarrow$ *state* $\Rightarrow$ *nat* $\Rightarrow$ *bool* **where**
Its_0: $\neg$ *bval b s* $\Longrightarrow$ *Its b c s 0* |
Its_Suc: $[\![$ *bval b s*; $(c,s) \Rightarrow s'$; *Its b c* s' n $]\!] \Longrightarrow$ *Its b c s* (*Suc n*)

The relation is in fact a function:

lemma *Its_fun*: *Its b c s n* $\Longrightarrow$ *Its b c s* $n' \Longrightarrow n=n'$
proof(*induction arbitrary*: n' *rule*:*Its.induct*)
 case *Its_0* **thus** *?case* **by**(*metis Its.cases*)
next
 case *Its_Suc* **thus** *?case* **by**(*metis Its.cases big_step_determ*)
qed

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

lemma *WHILE_Its*: (*WHILE b DO c*,*s*) $\Rightarrow t \Longrightarrow \exists\, n.$ *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

lemma wpt_is_pre: ⊢t {wpt c Q} c {Q}
proof (induction c arbitrary: Q)
  case SKIP show ?case by (auto intro:hoaret.Skip)
next
  case Assign show ?case by (auto intro:hoaret.Assign)
next
  case Seq thus ?case by (auto intro:hoaret.Seq)
next
  case If thus ?case by (auto intro:hoaret.If hoaret.conseq)
next
  case (While b c)
  let ?w = WHILE b DO c
  let ?T = Its b c
  have 1: ∀ s. wpt ?w Q s ⟶ wpt ?w Q s ∧ (∃ n. Its b c s n)
    unfolding wpt_def by (metis WHILE_Its)
  let ?R = λn s'. wpt ?w Q s' ∧ (∃ n'<n. ?T s' n')
  have ∀ s. wpt ?w Q s ∧ bval b s ∧ ?T s n ⟶ wpt c (?R n) s for n
  proof -
    have wpt c (?R n) s if bval b s and ?T s n and (?w, s) ⇒ t and Q t
for s t
    proof -
      from ‹bval b s› and ‹(?w, s) ⇒ t› obtain s' where
        (c,s) ⇒ s' (?w,s') ⇒ t by auto
      from ‹(?w, s') ⇒ t› obtain n' where ?T s' n'
        by (blast dest: WHILE_Its)
     with ‹bval b s› and ‹(c, s) ⇒ s'› have ?T s (Suc n') by (rule Its_Suc)
      with ‹?T s n› have n = Suc n' by (rule Its_fun)
      with ‹(c,s) ⇒ s'› and ‹(?w,s') ⇒ t› and ‹Q t› and ‹?T s' n'›
      show ?thesis by (auto simp: wpt_def)
    qed
    thus ?thesis
      unfolding wpt_def by auto

  qed
  note 2 = hoaret.While[OF strengthen_pre[OF this While.IH]]
  have ∀ s. wpt ?w Q s ∧ ¬ bval b s ⟶ Q s
    by (auto simp add:wpt_def)
  with 1 2 show ?case by (rule conseq)
```

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*: $\models_t \{P\}c\{Q\} \Longrightarrow \vdash_t \{P\}c\{Q\}$
apply(*rule strengthen_pre*[*OF _ wpt_is_pre*])
apply(*auto simp*: *hoare_tvalid_def wpt_def*)
done

corollary *hoaret_sound_complete*: $\vdash_t \{P\}c\{Q\} \longleftrightarrow \models_t \{P\}c\{Q\}$
by (*metis hoaret_sound hoaret_complete*)

end

theory *Hoare_Total_EX*
imports *Hoare*
begin

13.4.2 Hoare Logic for Total Correctness — *nat*-Indexed Invariant

This is the standard set of rules that you find in many publications. The While-rule is different from the one in Concrete Semantics in that the invariant is indexed by natural numbers and goes down by 1 with every iteration. The completeness proof is easier but the rule is harder to apply in program proofs.

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

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:
$\llbracket \bigwedge n{::}nat.\ \vdash_t \{P\ (Suc\ n)\}\ c\ \{P\ n\};$
$\forall n\ s.\ P\ (Suc\ n)\ s \longrightarrow bval\ b\ s;\ \ \forall s.\ P\ 0\ s \longrightarrow \neg\ bval\ b\ s\ \rrbracket$
$\Longrightarrow \vdash_t \{\lambda s.\ \exists n.\ P\ n\ s\}\ WHILE\ b\ DO\ c\ \{P\ 0\}\ \ |$

conseq: $\llbracket\ \forall s.\ P'\ s \longrightarrow P\ s;\ \vdash_t \{P\}c\{Q\};\ \forall s.\ Q\ s \longrightarrow Q'\ s\ \ \rrbracket \Longrightarrow$
$\vdash_t \{P'\}c\{Q'\}$

Building in the consequence rule:

lemma *strengthen_pre*:
$\llbracket\ \forall s.\ P'\ s \longrightarrow P\ s;\ \ \vdash_t \{P\}\ c\ \{Q\}\ \rrbracket \Longrightarrow \vdash_t \{P'\}\ c\ \{Q\}$
by (*metis conseq*)

lemma *weaken_post*:
$\llbracket \vdash_t \{P\}\ c\ \{Q\};\ \ \forall s.\ Q\ s \longrightarrow Q'\ s\ \rrbracket \Longrightarrow\ \vdash_t \{P\}\ c\ \{Q'\}$
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*])

The soundness theorem:

theorem *hoaret_sound*: $\vdash_t \{P\}c\{Q\}\ \ \Longrightarrow\ \ \models_t \{P\}c\{Q\}$
proof(*unfold hoare_tvalid_def*, *induction rule*: *hoaret.induct*)
  case (*While P c b*)
  have $P\ n\ s \Longrightarrow \exists t.\ (WHILE\ b\ DO\ c,\ s) \Rightarrow t \wedge P\ 0\ t$ **for** *n s*
  proof(*induction n arbitrary*: *s*)
    case *0* **thus** *?case* **using** *While.hyps*(*3*) *WhileFalse* **by** *blast*
  next
    case *Suc*
    thus *?case* **by** (*meson While.IH While.hyps*(*2*) *WhileTrue*)
  qed
  thus *?case* **by** *auto*
next
  case *If* **thus** *?case* **by** *auto blast*
qed *fastforce+*

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

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

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

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

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

Function *wpw* computes the weakest precondition of a While-loop that is unfolded a fixed number of times.

fun *wpw* :: *bexp* ⇒ *com* ⇒ *nat* ⇒ *assn* ⇒ *assn* **where**
$wpw\ b\ c\ 0\ Q\ s = (\neg\ bval\ b\ s \wedge Q\ s)$ |
$wpw\ b\ c\ (Suc\ n)\ Q\ s = (bval\ b\ s \wedge (\exists s'.\ (c,s) \Rightarrow s' \wedge\ wpw\ b\ c\ n\ Q\ s'))$

lemma *WHILE_Its*: $(WHILE\ b\ DO\ c,s) \Rightarrow t \Longrightarrow Q\ t \Longrightarrow \exists n.\ wpw\ b\ c\ n\ Q\ s$
proof(*induction WHILE b DO c s t rule*: *big_step_induct*)
  case *WhileFalse* **thus** *?case* **using** *wpw.simps*(*1*) **by** *blast*
next
  case *WhileTrue* **thus** *?case* **using** *wpw.simps*(*2*) **by** *blast*
qed

lemma *wpt_is_pre*: $\vdash_t \{wp_t\ c\ Q\}\ c\ \{Q\}$
proof (*induction c arbitrary*: *Q*)
  case *SKIP* **show** *?case* **by** (*auto intro*:*hoaret.Skip*)
next
  case *Assign* **show** *?case* **by** (*auto intro*:*hoaret.Assign*)
next
  case *Seq* **thus** *?case* **by** (*auto intro*:*hoaret.Seq*)
next
  case *If* **thus** *?case* **by** (*auto intro*:*hoaret.If hoaret.conseq*)
next
  case (*While b c*)
  let *?w* = *WHILE b DO c*
  have *c1*: $\forall s.\ wp_t\ ?w\ Q\ s \longrightarrow (\exists n.\ wpw\ b\ c\ n\ Q\ s)$

unfolding *wpt_def* **by** (*metis WHILE_Its*)
have *c3*: $\forall s.$ *wpw b c 0 Q s* $\longrightarrow$ *Q s* **by** *simp*
have *w2*: $\forall n\ s.$ *wpw b c* (*Suc n*) *Q s* $\longrightarrow$ *bval b s* **by** *simp*
have *w3*: $\forall s.$ *wpw b c 0 Q s* $\longrightarrow \neg$ *bval b s* **by** *simp*
have $\vdash_t$ {*wpw b c* (*Suc n*) *Q*} *c* {*wpw b c n Q*} **for** *n*
proof −
have $*$: $\forall s.$ *wpw b c* (*Suc n*) *Q s* $\longrightarrow$ ($\exists t.$ (*c*, *s*) $\Rightarrow$ *t* $\wedge$ *wpw b c n Q t*)
by *simp*
show *?thesis* **by**(*rule strengthen_pre*[*OF* $*$ *While.IH*[*of wpw b c n Q*, *unfolded wpt_def*]])
qed
from *conseq*[*OF c1 hoaret.While*[*OF this w2 w3*] *c3*]
show *?case* **.**
qed

theorem *hoaret_complete*: $\models_t$ $\{P\}c\{Q\} \Longrightarrow \vdash_t$ $\{P\}c\{Q\}$
apply(*rule strengthen_pre*[*OF _ wpt_is_pre*])
apply(*auto simp*: *hoare_tvalid_def wpt_def*)
done

corollary *hoaret_sound_complete*: $\vdash_t$ $\{P\}c\{Q\} \longleftrightarrow \models_t$ $\{P\}c\{Q\}$
by (*metis hoaret_sound hoaret_complete*)

end

theory *VCG_Total_EX*
imports *Hoare_Total_EX*
begin

13.5 Verification Conditions for Total Correctness

Annotated commands: commands where loops are annotated with invariants.

datatype *acom* =
Askip (*SKIP*) |
Aassign vname aexp ((_ ::= _) [*1000*, *61*] *61*) |
Aseq acom acom (_;;/ _ [*60*, *61*] *60*) |
Aif bexp acom acom ((*IF* _/ *THEN* _/ *ELSE* _) [*0*, *0*, *61*] *61*) |
Awhile nat $\Rightarrow$ *assn bexp acom*
(({_}/ *WHILE* _/ *DO* _) [*0*, *0*, *61*] *61*)

notation *com.SKIP* (*SKIP*)

Strip annotations:

fun *strip* :: *acom* ⇒ *com* **where**
strip SKIP = *SKIP* |
strip (x ::= a) = (x ::= a) |
strip (C_1;; C_2) = (*strip* C_1;; *strip* C_2) |
strip (*IF b THEN* C_1 *ELSE* C_2) = (*IF b THEN strip* C_1 *ELSE strip* C_2) |
strip ({_} *WHILE b DO C*) = (*WHILE b DO strip C*)

Weakest precondition from annotated commands:

fun *pre* :: *acom* ⇒ *assn* ⇒ *assn* **where**
pre SKIP Q = *Q* |
pre (x ::= a) Q = ($\lambda s.$ $Q(s(x$:= *aval a s*))) |
pre (C_1;; C_2) Q = *pre* C_1 (*pre* C_2 Q) |
pre (*IF b THEN* C_1 *ELSE* C_2) Q =
  ($\lambda s.$ *if bval b s then pre* C_1 *Q s else pre* C_2 *Q s*) |
pre ({I} *WHILE b DO C*) Q = ($\lambda s.$ $\exists n.$ I n s)

Verification condition:

fun *vc* :: *acom* ⇒ *assn* ⇒ *bool* **where**
vc SKIP Q = *True* |
vc (x ::= a) Q = *True* |
vc (C_1;; C_2) Q = (*vc* C_1 (*pre* C_2 Q) ∧ *vc* C_2 Q) |
vc (*IF b THEN* C_1 *ELSE* C_2) Q = (*vc* C_1 Q ∧ *vc* C_2 Q) |
vc ({I} *WHILE b DO C*) Q =
  (∀ *s n*. (*I* (*Suc n*) *s* ⟶ *pre C* (*I n*) *s*) ∧
      (*I* (*Suc n*) *s* ⟶ *bval b s*) ∧
      (*I 0 s* ⟶ ¬ *bval b s* ∧ *Q s*) ∧
      vc C (*I n*))

lemma *vc_sound*: *vc C Q* ⟹ $\vdash_t$ {*pre C Q*} *strip C* {*Q*}
proof(*induction C arbitrary*: *Q*)
  case (*Awhile I b C*)
  show *?case*
  proof(*simp*, *rule conseq*[*OF _ While*[*of I*]], *goal_cases*)
    case (*2 n*) **show** *?case*
      using *Awhile.IH*[*of I n*] *Awhile.prems*
      by (*auto intro*: *strengthen_pre*)
  qed (*insert Awhile.prems*, *auto*)
qed (*auto intro*: *conseq Seq If simp*: *Skip Assign*)

When trying to extend the completeness proof of the VCG for partial correctness to total correctness one runs into the following problem. In the case of the while-rule, the universally quantified n in the first premise means that for that premise the induction hypothesis does not yield a single annotated command C but merely that for every n such a C exists.

end

theory *Hoare_Total_EX2*
imports *Hoare*
begin

13.5.1 Hoare Logic for Total Correctness — With Logical Variables

This is the standard set of rules that you find in many publications. In the while-rule, a logical variable is needed to remember the pre-value of the variant (an expression that decreases by one with each iteration). In this theory, logical variables are modeled explicitly. A simpler (but not quite as flexible) approach is found in theory *Hoare_Total_EX*: pre and post-condition are connected via a universally quantified HOL variable.

type_synonym *lvname* = *string*
type_synonym *assn2* = (*lvname* ⇒ *nat*) ⇒ *state* ⇒ *bool*

definition *hoare_tvalid* :: *assn2* ⇒ *com* ⇒ *assn2* ⇒ *bool*
 $(\models_t\ \{(1\_)\}/\ (\_)/\ \{(1\_)\}\ 50)$ **where**
$\models_t \{P\}c\{Q\} \longleftrightarrow (\forall\, l\ s.\ P\ l\ s \longrightarrow (\exists\, t.\ (c,s) \Rightarrow t \wedge Q\ l\ t))$

inductive
 $hoaret :: assn2 \Rightarrow com \Rightarrow assn2 \Rightarrow bool\ (\vdash_t\ (\{(1\_)\}/\ (\_)/\ \{(1\_)\})\ 50)$
where

Skip: $\vdash_t \{P\}\ SKIP\ \{P\}$ |

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

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

While:
 $[\![\vdash_t \{\lambda l.\ P\ (l(x := Suc(l(x))))\}\ c\ \{P\};$
 $\forall\, l\ s.\ l\ x > 0 \wedge P\ l\ s \longrightarrow bval\ b\ s;$
 $\forall\, l\ s.\ l\ x = 0 \wedge P\ l\ s \longrightarrow \neg\ bval\ b\ s]\!]$
 $\Longrightarrow \vdash_t \{\lambda l\ s.\ \exists\, n.\ P\ (l(x:=n))\ s\}\ WHILE\ b\ DO\ c\ \{\lambda l\ s.\ P\ (l(x := 0))$
$s\}$ |

$conseq$: $\llbracket\ \forall\, l\ s.\ P'\ l\ s \longrightarrow P\ l\ s; \vdash_t \{P\}c\{Q\}; \forall\, l\ s.\ Q\ l\ s \longrightarrow Q'\ l\ s\ \rrbracket \Longrightarrow$
$\vdash_t \{P'\}c\{Q'\}$

Building in the consequence rule:

lemma *strengthen_pre*:
$\llbracket\ \forall\, l\ s.\ P'\ l\ s \longrightarrow P\ l\ s;\ \vdash_t \{P\}\ c\ \{Q\}\ \rrbracket \Longrightarrow \vdash_t \{P'\}\ c\ \{Q\}$
by (*metis conseq*)

lemma *weaken_post*:
$\llbracket \vdash_t \{P\}\ c\ \{Q\};\ \forall\, l\ s.\ Q\ l\ s \longrightarrow Q'\ l\ s\ \rrbracket \Longrightarrow\ \vdash_t \{P\}\ c\ \{Q'\}$
by (*metis conseq*)

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

The soundness theorem:

theorem *hoaret_sound*: $\vdash_t \{P\}c\{Q\}\ \Longrightarrow\ \models_t \{P\}c\{Q\}$
proof(*unfold hoare_tvalid_def*, *induction rule*: *hoaret.induct*)
  case (*While P x c b*)
  have $\llbracket\ l\ x = n;\ P\ l\ s\ \rrbracket \Longrightarrow \exists\, t.\ (\mathit{WHILE}\ b\ \mathit{DO}\ c,\ s) \Rightarrow t \wedge P\ (l(x := 0))$
t **for** $n\ l\ s$
  proof(*induction n arbitrary*: *l s*)
    case *0* **thus** *?case* **using** *While.hyps*(*3*) *WhileFalse*
      by (*metis fun_upd_triv*)
  next
    case *Suc*
    thus *?case* **using** *While.IH While.hyps*(*2*) *WhileTrue*
      by (*metis fun_upd_same fun_upd_triv fun_upd_upd zero_less_Suc*)
  qed
  thus *?case* **by** *fastforce*
next
  case *If* **thus** *?case* **by** *auto blast*
qed *fastforce+*

definition *wpt* :: *com* ⇒ *assn2* ⇒ *assn2* (wp_t) **where**
$wp_t\ c\ Q\ =\ (\lambda l\ s.\ \exists\, t.\ (c,s) \Rightarrow t \wedge Q\ l\ t)$

lemma [*simp*]: wp_t *SKIP* $Q = Q$
by(*auto intro*!: *ext simp*: *wpt_def*)

lemma [*simp*]: $wp_t\ (x ::= e)\ Q = (\lambda l\ s.\ Q\ l\ (s(x := \mathit{aval}\ e\ s)))$
by(*auto intro*!: *ext simp*: *wpt_def*)

lemma *wpt_Seq*[*simp*]: wp_t $(c_1;;c_2)$ $Q = wp_t\ c_1\ (wp_t\ c_2\ Q)$
by (*auto simp*: *wpt_def fun_eq_iff*)

lemma [*simp*]:
wp_t (*IF b THEN* c_1 *ELSE* c_2) $Q = (\lambda l\ s.\ wp_t$ (*if bval b s then* c_1 *else* c_2) $Q\ l\ s)$
by (*auto simp*: *wpt_def fun_eq_iff*)

Function *wpw* computes the weakest precondition of a While-loop that is unfolded a fixed number of times.

fun *wpw* :: *bexp* ⇒ *com* ⇒ *nat* ⇒ *assn2* ⇒ *assn2* **where**
wpw b c 0 Q l s = (¬ *bval b s* ∧ *Q l s*) |
wpw b c (*Suc n*) *Q l s* = (*bval b s* ∧ (∃ *s'*. (*c*,*s*) ⇒ *s'* ∧ *wpw b c n Q l s'*))

lemma *WHILE_Its*:
(*WHILE b DO c*,*s*) ⇒ *t* ⟹ *Q l t* ⟹ ∃ *n*. *wpw b c n Q l s*
proof(*induction WHILE b DO c s t arbitrary*: *l rule*: *big_step_induct*)
case *WhileFalse* **thus** *?case* **using** *wpw.simps*(*1*) **by** *blast*
next
case *WhileTrue* **show** *?case*
using *wpw.simps*(*2*) *WhileTrue*(*1*,*2*) *WhileTrue*(*5*)[*OF WhileTrue*(*6*)]
by *blast*
qed

definition *support* :: *assn2* ⇒ *string set* **where**
support P = {*x*. ∃ *l1 l2 s*. (∀ *y*. *y* ≠ *x* ⟶ *l1 y* = *l2 y*) ∧ *P l1 s* ≠ *P l2 s*}

lemma *support_wpt*: *support* (wp_t *c Q*) ⊆ *support Q*
by(*simp add*: *support_def wpt_def*) *blast*

lemma *support_wpw0*: *support* (*wpw b c n Q*) ⊆ *support Q*
proof(*induction n*)
case *0* **show** *?case* **by** (*simp add*: *support_def*) *blast*
next
case *Suc*
have *1*: *support* (λ*l s*. *A s* ∧ *B l s*) ⊆ *support B* **for** *A B*
by(*auto simp*: *support_def*)
have *2*: *support* (λ*l s*. ∃ *s'*. *A s s'* ∧ *B l s'*) ⊆ *support B* **for** *A B*
by(*auto simp*: *support_def*) *blast+*
from *Suc 1 2* **show** *?case* **by** *simp* (*meson order_trans*)
qed

lemma *support_wpw_Un*:

```
  support (%l. wpw b c (l x) Q l) ⊆ insert x (UN n. support(wpw b c n Q))
using support_wpw0[of b c _ Q]
apply(auto simp add: support_def subset_iff)
apply metis
apply metis
done

lemma support_wpw: support (%l. wpw b c (l x) Q l) ⊆ insert x (support
Q)
using support_wpw0[of b c _ Q] support_wpw_Un[of b c _ Q]
by blast

lemma assn2_lupd: x ∉ support Q ⟹ Q (l(x:=n)) = Q l
by(simp add: support_def fun_upd_other fun_eq_iff)
  (metis (no_types, lifting) fun_upd_def)

abbreviation new Q ≡ SOME x. x ∉ support Q

lemma wpw_lupd: x ∉ support Q ⟹ wpw b c n Q (l(x := u)) = wpw b c
n Q l
by(induction n) (auto simp: assn2_lupd fun_eq_iff)

lemma wpt_is_pre: finite(support Q) ⟹ ⊢t {wpt c Q} c {Q}
proof (induction c arbitrary: Q)
  case SKIP show ?case by (auto intro:hoaret.Skip)
next
  case Assign show ?case by (auto intro:hoaret.Assign)
next
  case (Seq c1 c2) show ?case
    by (auto intro:hoaret.Seq Seq finite_subset[OF support_wpt])
next
  case If thus ?case by (auto intro:hoaret.If hoaret.conseq)
next
  case (While b c)
  let ?x = new Q
  have ∃ x. x ∉ support Q using While.prems infinite_UNIV_listI
    using ex_new_if_finite by blast
  hence [simp]: ?x ∉ support Q by (rule someI_ex)
  let ?w = WHILE b DO c
  have fsup: finite (support (λl. wpw b c (l x) Q l)) for x
    using finite_subset[OF support_wpw] While.prems by simp
  have c1: ∀ l s. wpt ?w Q l s ⟶ (∃ n. wpw b c n Q l s)
    unfolding wpt_def by (metis WHILE_Its)
  have c2: ∀ l s. l ?x = 0 ∧ wpw b c (l ?x) Q l s ⟶ ¬ bval b s
```

```
    by (simp cong: conj_cong)
  have w2: ∀ l s. 0 < l ?x ∧ wpw b c (l ?x) Q l s ⟶ bval b s
    by (auto simp: gr0_conv_Suc cong: conj_cong)
  have 1: ∀ l s. wpw b c (Suc(l ?x)) Q l s ⟶
               (∃ t. (c, s) ⇒ t ∧ wpw b c (l ?x) Q l t)
    by simp
  have ∗: ⊢t {λl. wpw b c (Suc (l ?x)) Q l} c {λl. wpw b c (l ?x) Q l}
    by(rule strengthen_pre[OF 1
          While.IH[of λl. wpw b c (l ?x) Q l, unfolded wpt_def, OF fsup]])
  show ?case
  apply(rule conseq[OF _ hoaret.While[OF _ w2 c2]])
    apply (simp_all add: c1 ∗ assn2_lupd wpw_lupd del: wpw.simps(2))
  done
qed

theorem hoaret_complete: finite(support Q) ⟹ ⊨t {P}c{Q} ⟹ ⊢t {P}c{Q}
apply(rule strengthen_pre[OF _ wpt_is_pre])
apply(auto simp: hoare_tvalid_def wpt_def)
done

end


theory VCG_Total_EX2
imports Hoare_Total_EX2
begin
```

13.6 Verification Conditions for Total Correctness

Theory *VCG_Total_EX* conatins a VCG built on top of a Hoare logic without logical variables. As a result the completeness proof runs into a problem. This theory uses a Hoare logic with logical variables and proves soundness and completeness.

Annotated commands: commands where loops are annotated with invariants.

```
datatype acom =
  Askip                  (SKIP) |
  Aassign vname aexp     ((_ ::= _) [1000, 61] 61) |
  Aseq   acom acom       (_;;/ _  [60, 61] 60) |
  Aif bexp acom acom     ((IF _/ THEN _/ ELSE _)  [0, 0, 61] 61) |
  Awhile assn2 lvname bexp acom
    (({_'/_}/ WHILE _/ DO _)  [0, 0, 0, 61] 61)
```

notation *com.SKIP* (*SKIP*)

Strip annotations:

fun *strip* :: *acom* ⇒ *com* **where**
strip SKIP = *SKIP* |
strip (x ::= a) = (x ::= a) |
strip (C_1;; C_2) = (*strip* C_1;; *strip* C_2) |
strip (*IF b THEN* C_1 *ELSE* C_2) = (*IF b THEN strip* C_1 *ELSE strip* C_2) |
strip ({_/_} *WHILE b DO C*) = (*WHILE b DO strip C*)

Weakest precondition from annotated commands:

fun *pre* :: *acom* ⇒ *assn2* ⇒ *assn2* **where**
pre SKIP Q = *Q* |
pre (x ::= a) Q = ($\lambda l\ s.\ Q\ l\ (s(x := aval\ a\ s))$) |
pre (C_1;; C_2) Q = *pre* C_1 (*pre* C_2 Q) |
pre (*IF b THEN* C_1 *ELSE* C_2) Q =
  ($\lambda l\ s.$ *if bval b s then pre* C_1 $Q\ l\ s$ *else pre* C_2 $Q\ l\ s$) |
pre ({I/x} *WHILE b DO C*) Q = ($\lambda l\ s.\ \exists\, n.\ I\ (l(x{:=}n))\ s$)

Verification condition:

fun *vc* :: *acom* ⇒ *assn2* ⇒ *bool* **where**
vc SKIP Q = *True* |
vc (x ::= a) Q = *True* |
vc (C_1;; C_2) Q = (*vc* C_1 (*pre* C_2 Q) ∧ *vc* C_2 Q) |
vc (*IF b THEN* C_1 *ELSE* C_2) Q = (*vc* C_1 Q ∧ *vc* C_2 Q) |
vc ({I/x} *WHILE b DO C*) Q =
  ($\forall\, l\ s.$ ($I\ (l(x{:=}Suc(l\ x)))\ s \longrightarrow$ *pre* $C\ I\ l\ s$) ∧
    ($l\ x > 0 \wedge I\ l\ s \longrightarrow$ *bval b s*) ∧
    ($I\ (l(x := 0))\ s \longrightarrow \neg$ *bval b s* ∧ $Q\ l\ s$) ∧
    vc C I)

lemma *vc_sound*: *vc C Q* $\Longrightarrow \vdash_t$ {*pre C Q*} *strip C* {Q}
proof(*induction C arbitrary*: *Q*)
  case (*Awhile I x b C*)
  show *?case*
  proof(*simp*, *rule weaken_post*[*OF While*[*of I x*]], *goal_cases*)
    case *1* **show** *?case*
      using *Awhile.IH*[*of I*] *Awhile.prems* **by** (*auto intro*: *strengthen_pre*)
  next
    case *3* **show** *?case*
      using *Awhile.prems* **by** (*simp*) (*metis fun_upd_triv*)
  qed (*insert Awhile.prems*, *auto*)
qed (*auto intro*: *conseq Seq If simp*: *Skip Assign*)

Completeness:

```
lemma pre_mono:
  ∀ l s. P l s ⟶ P' l s ⟹ pre C P l s ⟹ pre C P' l s
proof (induction C arbitrary: P P' l s)
  case Aseq thus ?case by simp metis
qed simp_all

lemma vc_mono:
  ∀ l s. P l s ⟶ P' l s ⟹ vc C P ⟹ vc C P'
proof(induction C arbitrary: P P')
  case Aseq thus ?case by simp (metis pre_mono)
qed simp_all

lemma vc_complete:
⊢t {P}c{Q} ⟹ ∃ C. strip C = c ∧ vc C Q ∧ (∀ l s. P l s ⟶ pre C Q l
s)
  (is _ ⟹ ∃ C. ?G P c Q C)
proof (induction rule: hoaret.induct)
  case Skip
  show ?case (is ∃ C. ?C C)
  proof show ?C Askip by simp qed
next
  case (Assign P a x)
  show ?case (is ∃ C. ?C C)
  proof show ?C(Aassign x a) by simp qed
next
  case (Seq P c1 Q c2 R)
  from Seq.IH obtain C1 where ih1: ?G P c1 Q C1 by blast
  from Seq.IH obtain C2 where ih2: ?G Q c2 R C2 by blast
  show ?case (is ∃ C. ?C C)
  proof
    show ?C(Aseq C1 C2)
      using ih1 ih2 by (fastforce elim!: pre_mono vc_mono)
  qed
next
  case (If P b c1 Q c2)
  from If.IH obtain C1 where ih1: ?G (λl s. P l s ∧ bval b s) c1 Q C1
    by blast
  from If.IH obtain C2 where ih2: ?G (λl s. P l s ∧ ¬bval b s) c2 Q C2
    by blast
  show ?case (is ∃ C. ?C C)
  proof
    show ?C(Aif b C1 C2) using ih1 ih2 by simp
  qed
next
```

```
  case (While P x c b)
  from While.IH obtain C where
    ih: ?G (λl s. P (l(x:=Suc(l x))) s ∧ bval b s) c P C
    by blast
  show ?case (is ∃ C. ?C C)
  proof
    have vc ({P/x} WHILE b DO C) (λl. P (l(x := 0)))
      using ih While.hyps(2,3)
      by simp (metis fun_upd_same zero_less_Suc)
    thus ?C(Awhile P x b C) using ih by simp
 qed
next
  case conseq thus ?case by(fast elim!: pre_mono vc_mono)
qed

end
```

14 Abstract Interpretation

```
theory Complete_Lattice
imports Main
begin

locale Complete_Lattice =
fixes L :: 'a::order set and Glb :: 'a set ⇒ 'a
assumes Glb_lower: A ⊆ L ⟹ a ∈ A ⟹ Glb A ≤ a
and Glb_greatest: b ∈ L ⟹ ∀ a∈A. b ≤ a ⟹ b ≤ Glb A
and Glb_in_L: A ⊆ L ⟹ Glb A ∈ L
begin

definition lfp :: ('a ⇒ 'a) ⇒ 'a where
lfp f = Glb {a : L. f a ≤ a}

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

lemma lfp_lowerbound:
  ⟦ a ∈ L; f a ≤ a ⟧ ⟹ lfp f ≤ a
by (auto simp add: lfp_def intro: Glb_lower)

lemma lfp_greatest:
  ⟦ a ∈ L; ⋀u. ⟦ u ∈ L; f u ≤ u⟧ ⟹ a ≤ u ⟧ ⟹ a ≤ lfp f
by (auto simp add: lfp_def intro: Glb_greatest)
```

lemma *lfp_unfold*: **assumes** $\bigwedge x.\ f\ x \in L \longleftrightarrow x \in L$
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

14.1 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)
```

notation *com.SKIP* (*SKIP*)
fun *strip* :: *'a acom* ⇒ *com* **where**
strip (SKIP {P}) = 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 *asize* :: *com* ⇒ *nat* **where**

$asize\ SKIP = 1\ |$
$asize\ (x ::= e) = 1\ |$
$asize\ (C_1;;C_2) = asize\ C_1 + asize\ C_2\ |$
$asize\ (IF\ b\ THEN\ C_1\ ELSE\ C_2) = asize\ C_1 + asize\ C_2 + 3\ |$
$asize\ (WHILE\ b\ DO\ C) = asize\ C + 3$

definition $shift :: (nat \Rightarrow 'a) \Rightarrow nat \Rightarrow nat \Rightarrow 'a$ **where**
$shift\ f\ n = (\lambda p.\ f(p+n))$

fun $annotate :: (nat \Rightarrow 'a) \Rightarrow com \Rightarrow 'a\ acom$ **where**
$annotate\ f\ SKIP = SKIP\ \{f\ 0\}\ |$
$annotate\ f\ (x ::= e) = x ::= e\ \{f\ 0\}\ |$
$annotate\ f\ (c_1;;c_2) = annotate\ f\ c_1;;\ annotate\ (shift\ f\ (asize\ c_1))\ c_2\ |$
$annotate\ f\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) =$
$\quad IF\ b\ THEN\ \{f\ 0\}\ annotate\ (shift\ f\ 1)\ c_1$
$\quad ELSE\ \{f(asize\ c_1 + 1)\}\ annotate\ (shift\ f\ (asize\ c_1 + 2))\ c_2$
$\quad \{f(asize\ c_1 + asize\ c_2 + 2)\}\ |$
$annotate\ f\ (WHILE\ b\ DO\ c) =$
$\quad \{f\ 0\}\ WHILE\ b\ DO\ \{f\ 1\}\ annotate\ (shift\ f\ 2)\ c\ \{f(asize\ c + 2)\}$

fun $annos :: 'a\ acom \Rightarrow 'a\ list$ **where**
$annos\ (SKIP\ \{P\}) = [P]\ |$
$annos\ (x ::= e\ \{P\}) = [P]\ |$
$annos\ (C_1;;C_2) = annos\ C_1\ @\ annos\ C_2\ |$
$annos\ (IF\ b\ THEN\ \{P_1\}\ C_1\ ELSE\ \{P_2\}\ C_2\ \{Q\}) =$
$\quad P_1\ \#\ annos\ C_1\ @\ \ P_2\ \#\ annos\ C_2\ @\ [Q]\ |$
$annos\ (\{I\}\ WHILE\ b\ DO\ \{P\}\ C\ \{Q\}) = I\ \#\ P\ \#\ annos\ C\ @\ [Q]$
definition $anno :: 'a\ acom \Rightarrow nat \Rightarrow 'a$ **where**
$anno\ C\ p = annos\ C\ !\ p$

definition $post :: 'a\ acom \Rightarrow 'a$ **where**
$post\ C = last(annos\ C)$
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 $annos\_ne$: $annos\ C \neq []$
by$(induction\ C)\ auto$

lemma *strip_annotate*[*simp*]: *strip*(*annotate f c*) = *c*
by(*induction c arbitrary*: *f*) *auto*

lemma *length_annos_annotate*[*simp*]: *length* (*annos* (*annotate f c*)) = *asize c*
by(*induction c arbitrary*: *f*) *auto*

lemma *size_annos*: *size*(*annos C*) = *asize*(*strip C*)
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(*case_tac C1*, *simp_all*)+
done

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

lemma *anno_annotate*[*simp*]: *p* < *asize c* $\Longrightarrow$ *anno* (*annotate f c*) *p* = *f p*
apply(*induction c arbitrary*: *f p*)
apply (*auto simp*: *anno_def nth_append nth_Cons numeral_eq_Suc shift_def*
split: *nat.split*)
apply (*metis add_Suc_right add_diff_inverse add.commute*)
apply(*rule_tac f=f* **in** *arg_cong*)
apply *arith*
apply (*metis less_Suc_eq*)
done

lemma *eq_acom_iff_strip_annos*:
C1 = *C2* $\longleftrightarrow$ *strip C1* = *strip C2* $\wedge$ *annos C1* = *annos C2*
apply(*induction C1 arbitrary*: *C2*)
apply(*case_tac C2*, *auto simp*: *size_annos_same2*)+
done

lemma *eq_acom_iff_strip_anno*:
C1=*C2* $\longleftrightarrow$ *strip C1* = *strip C2* $\wedge$ ($\forall$ *p*<*size*(*annos C1*). *anno C1 p* = *anno C2 p*)
by(*auto simp add*: *eq_acom_iff_strip_annos anno_def*
list_eq_iff_nth_eq size_annos_same2)

lemma *post_map_acom*[*simp*]: *post*(*map_acom f C*) = *f*(*post C*)
by (*induction C*) (*auto simp*: *post_def last_append annos_ne*)

lemma *strip_map_acom*[*simp*]: *strip* (*map_acom f C*) = *strip C*
by (*induction C*) *auto*

lemma *anno_map_acom*: p < *size*(*annos C*) $\Longrightarrow$ *anno* (*map_acom f C*) *p* = *f*(*anno C p*)
apply(*induction C arbitrary*: *p*)
apply(*auto simp*: *anno_def nth_append nth_Cons′ size_annos*)
done

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

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

lemma *strip_eq_Seq*:
strip C = *c1*;;*c2* $\longleftrightarrow$ ($\exists$ *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* $\longleftrightarrow$
($\exists$ *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* $\longleftrightarrow$
($\exists$ *I P C1 Q*. *C* = {*I*} *WHILE b DO* {*P*} *C1* {*Q*} & *strip C1* = *c1*)
by (*cases C*) *simp_all*

lemma [*simp*]: *shift* (λ*p*. *a*) *n* = (λ*p*. *a*)
by(*simp add*:*shift_def*)

lemma *set_annos_anno*[*simp*]: *set* (*annos* (*annotate* (λ*p*. *a*) *c*)) = {*a*}
by(*induction c*) *simp_all*

lemma *post_in_annos*: *post C* $\in$ *set*(*annos C*)
by(*auto simp*: *post_def annos_ne*)

lemma *post_anno_asize*: *post C* = *anno C* (*size*(*annos C*) − *1*)
by(*simp add*: *post_def last_conv_nth*[*OF annos_ne*] *anno_def*)

end
theory *Collecting*
imports *Complete_Lattice Big_Step ACom*
begin

14.2 The generic Step function

notation
  sup (**infixl** ⊔ *65*) **and**
  inf (**infixl** ⊓ *70*) **and**
  bot (⊥) **and**
  top (⊤)

context
  fixes f :: *vname* ⇒ *aexp* ⇒ *'a* ⇒ *'a::sup*
  fixes g :: *bexp* ⇒ *'a* ⇒ *'a*
begin
fun *Step* :: *'a* ⇒ *'a acom* ⇒ *'a acom* **where**
Step S (*SKIP* {*Q*}) = (*SKIP* {*S*}) |
Step S (*x* ::= *e* {*Q*}) =
  x ::= *e* {*f x 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 {*g b S*} *Step P1 C1 ELSE* {*g* (*Not b*) *S*} *Step P2 C2*
  {*post C1* ⊔ *post C2*} |
Step S ({*I*} *WHILE b DO* {*P*} *C* {*Q*}) =
  {*S* ⊔ *post C*} *WHILE b DO* {*g b I*} *Step P C* {*g* (*Not b*) *I*}
end

lemma *strip_Step*[*simp*]: *strip*(*Step f g S C*) = *strip C*
by(*induct C arbitrary*: *S*) *auto*

14.3 Collecting Semantics of Commands

14.3.1 Annotated commands as a complete lattice

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

definition *less_eq_acom* :: (*'a::order*)*acom* ⇒ *'a acom* ⇒ *bool* **where**
C1 ≤ *C2* ⟷ *strip C1* = *strip C2* ∧ (∀ *p*<*size*(*annos C1*). *anno C1 p* ≤ *anno C2 p*)

definition *less_acom* :: *'a acom* ⇒ *'a acom* ⇒ *bool* **where**

less_acom x y = (*x* ≤ *y* ∧ ¬ *y* ≤ *x*)

instance
proof (*standard*, *goal_cases*)
  case *1* **show** *?case* **by**(*simp add*: *less_acom_def*)
next
  case *2* **thus** *?case* **by**(*auto simp*: *less_eq_acom_def*)
next
  case *3* **thus** *?case* **by**(*fastforce simp*: *less_eq_acom_def size_annos*)
next
  case *4* **thus** *?case*
    by(*fastforce simp*: *le_antisym less_eq_acom_def size_annos*
        eq_acom_iff_strip_anno)
qed

end

lemma *less_eq_acom_annos*:
  C1 ≤ *C2* ⟷ *strip C1* = *strip C2* ∧ *list_all2* (≤) (*annos C1*) (*annos*
C2)
by(*auto simp add*: *less_eq_acom_def anno_def list_all2_conv_all_nth size_annos_same2*)

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

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

lemma *Seq_le*[*simp*]: *C1*;;*C2* ≤ *C* ⟷ (∃ *C1' C2'*. *C* = *C1'*;;*C2'* ∧ *C1* ≤
C1' ∧ *C2* ≤ *C2'*)
apply (*cases C*)
apply(*auto simp*: *less_eq_acom_annos list_all2_append size_annos_same2*)
done

lemma *If_le*[*simp*]: *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'*)
apply (*cases C*)
apply(*auto simp*: *less_eq_acom_annos list_all2_append size_annos_same2*)
done

lemma *While_le*[*simp*]: $\{I\}$ *WHILE b DO* $\{p\}$ *C* $\{P\} \le W \longleftrightarrow$
$(\exists I'\ p'\ C'\ P'.\ W = \{I'\}$ *WHILE b DO* $\{p'\}\ C'\ \{P'\} \wedge C \le C' \wedge p \le p' \wedge I \le I' \wedge P \le P')$
apply (*cases W*)
apply(*auto simp*: *less_eq_acom_annos list_all2_append size_annos_same2*)
done

lemma *mono_post*: $C \le C' \Longrightarrow$ *post* $C \le$ *post* C'
using *annos_ne*[*of C'*]
by(*auto simp*: *post_def less_eq_acom_def last_conv_nth*[*OF annos_ne*] *anno_def*
dest: *size_annos_same*)

definition *Inf_acom* :: *com* ⇒ *'a::complete_lattice acom set* ⇒ *'a acom*
where
Inf_acom c M = *annotate* (λ*p*. *INF C*:*M*. *anno C p*) *c*

global_interpretation
Complete_Lattice {*C*. *strip C* = *c*} *Inf_acom c* **for** *c*
proof (*standard*, *goal_cases*)
case *1* **thus** *?case*
by(*auto simp*: *Inf_acom_def less_eq_acom_def size_annos intro*:*INF_lower*)
next
case *2* **thus** *?case*
by(*auto simp*: *Inf_acom_def less_eq_acom_def size_annos intro*:*INF_greatest*)
next
case *3* **thus** *?case* **by**(*auto simp*: *Inf_acom_def*)
qed

14.3.2 Collecting semantics

definition *step* = *Step* (λ*x e S*. {*s*(*x* := *aval e s*) |*s*. *s* ∈ *S*}) (λ*b S*. {*s*:*S*. *bval b s*})

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

lemma *mono2_Step*: **fixes** *C1 C2* :: *'a::semilattice_sup acom*
assumes !!*x e S1 S2*. $S1 \le S2 \Longrightarrow f\ x\ e\ S1 \le f\ x\ e\ S2$
!!*b S1 S2*. $S1 \le S2 \Longrightarrow g\ b\ S1 \le g\ b\ S2$
shows $C1 \le C2 \Longrightarrow S1 \le S2 \Longrightarrow$ *Step f g S1 C1* $\le$ *Step f g S2 C2*
proof(*induction S1 C1 arbitrary*: *C2 S2 rule*: *Step.induct*)
case *1* **thus** *?case* **by**(*auto*)
next
case *2* **thus** *?case* **by** (*auto simp*: *assms*(*1*))

next
  case *3* **thus** *?case* **by**(*auto simp*: *mono_post*)
next
  case *4* **thus** *?case*
    by(*auto simp*: *subset_iff assms*(*2*))
      (*metis mono_post le_supI1 le_supI2*)+
next
  case *5* **thus** *?case*
    by(*auto simp*: *subset_iff assms*(*2*))
      (*metis mono_post le_supI1 le_supI2*)+
qed

lemma *mono2_step*: *C1* $\leq$ *C2* $\Longrightarrow$ *S1* $\subseteq$ *S2* $\Longrightarrow$ *step S1 C1* $\leq$ *step S2 C2*
unfolding *step_def* **by**(*rule mono2_Step*) *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 simp*: *step_def*)

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*])

14.3.3 Relation to big-step semantics

lemma *asize_nz*: *asize*(*c*::*com*) $\neq$ *0*
by (*metis length_0_conv length_annos_annotate annos_ne*)

lemma *post_Inf_acom*:
  $\forall$ *C*$\in$*M*. *strip C* = *c* $\Longrightarrow$ *post* (*Inf_acom c M*) = $\bigcap$(*post* ‘ *M*)
apply(*subgoal_tac* $\forall$ *C*$\in$*M*. *size*(*annos C*) = *asize c*)
apply(*simp add*: *post_anno_asize Inf_acom_def asize_nz neq0_conv*[*symmetric*])
apply(*simp add*: *size_annos*)
done

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_Inf_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 step_def post_def*)
next
 case *Assign* **thus** *?case*
  by(*fastforce simp*: *strip_eq_Assign step_def post_def*)
next
 case *Seq* **thus** *?case*
  by(*fastforce simp*: *strip_eq_Seq step_def post_def last_append annos_ne*)
next
 case *IfTrue* **thus** *?case* **apply**(*auto simp*: *strip_eq_If step_def post_def*)
  by (*metis* (*lifting,full_types*) *mem_Collect_eq set_mp*)
next
 case *IfFalse* **thus** *?case* **apply**(*auto simp*: *strip_eq_If step_def post_def*)
  by (*metis* (*lifting,full_types*) *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'* $\leq$ *C'* \{*s* $\in$ *I*. *bval b s*\} $\leq$ *P* *S* $\leq$ *I* *step* (*post C'*) *C* $\leq$ *C*
  by (*auto simp*: *step_def post_def*)
 have *step* \{*s* $\in$ *I*. *bval b s*\} *C'* $\leq$ *C'*
  by (*rule order_trans*[*OF mono2_step*[*OF order_refl* ‹\{*s* $\in$ *I*. *bval b s*\} $\leq$ *P*›] ‹*step P C'* $\leq$ *C'*›])
 have *s1* $\in$ \{*s*$\in$*I*. *bval b s*\} **using** ‹*s1* $\in$ *S*› ‹*S* $\subseteq$ *I*› ‹*bval b s1*› **by** *auto*
 note *s2_in_post_C'* = *WhileTrue.IH*(*1*)[*OF* ‹*strip C'* = *c'*› *this* ‹*step* \{*s* $\in$ *I*. *bval b s*\} *C'* $\leq$ *C'*›]
 from *WhileTrue.IH*(*2*)[*OF WhileTrue.prems*(*1*) *s2_in_post_C'* ‹*step* (*post C'*) *C* $\leq$ *C*›]
 show *?case* **.**
next
 case (*WhileFalse b s1 c'*) **thus** *?case*
  by (*force simp*: *strip_eq_While step_def post_def*)
qed

lemma *big_step_lfp*: ⟦ (*c*,*s*) $\Rightarrow$ *t*; *s* $\in$ *S* ⟧ $\Longrightarrow$ *t* $\in$ *post*(*lfp c* (*step S*))
by(*auto simp add*: *post_lfp intro*: *big_step_post_step*)

lemma *big_step_CS*: $(c,s) \Rightarrow t \Longrightarrow t \in post(CS\ c)$
by(*simp add*: *CS_def big_step_lfp*)

end
theory *Collecting1*
imports *Collecting*
begin

14.4 A small step semantics on annotated commands

The idea: the state is propagated through the annotated command as an annotation $\{s\}$, all other annotations are $\{\}$. It is easy to show that this semantics approximates the collecting semantics.

lemma *step_preserves_le*:
$\llbracket$ *step S cs = cs*; $S' \subseteq S$; $cs' \leq cs$ $\rrbracket \Longrightarrow$
step S' $cs' \leq cs$
by (*metis mono2_step*)

lemma *steps_empty_preserves_le*: **assumes** *step S cs = cs*
shows $cs' \leq cs \Longrightarrow (step\ \{\}\ \hat{}\hat{}\ n)\ cs' \leq cs$
proof(*induction n arbitrary*: cs')
case *0* **thus** *?case* **by** *simp*
next
case (*Suc n*) **thus** *?case*
using *Suc.IH*[*OF step_preserves_le*[*OF assms empty_subsetI Suc.prems*]]
by(*simp add*:*funpow_swap1*)
qed

definition *steps* :: *state* $\Rightarrow$ *com* $\Rightarrow$ *nat* $\Rightarrow$ *state set acom* **where**
steps s c n = ((*step* $\{\}$)$\hat{}\hat{}$*n*) (*step* $\{s\}$ (*annotate* (λp. $\{\}$) *c*))

lemma *steps_approx_fix_step*: **assumes** *step S cs = cs* **and** $s \in S$
shows *steps s* (*strip cs*) $n \leq cs$
proof−
let *?bot* = *annotate* (λp. $\{\}$) (*strip cs*)
have *?bot* $\leq$ *cs* **by**(*induction cs*) *auto*
from *step_preserves_le*[*OF assms*(*1*)_ *this*, *of* $\{s\}$] ‹$s \in S$›
have *1*: *step* $\{s\}$ *?bot* $\leq$ *cs* **by** *simp*
from *steps_empty_preserves_le*[*OF assms*(*1*) *1*]
show *?thesis* **by**(*simp add*: *steps_def*)
qed

theorem *steps_approx_CS*: *steps s c n* $\leq$ *CS c*
by (*metis CS_unfold UNIV_I steps_approx_fix_step strip_CS*)

end
theory *Collecting_Examples*
imports *Collecting Vars*
begin

14.5 Pretty printing state sets

Tweak code generation to work with sets of non-equality types:

declare *insert_code*[*code del*] *union_coset_filter*[*code del*]
lemma *insert_code* [*code*]: *insert x* (*set xs*) = *set* (*x*#*xs*)
by *simp*

Compensate for the fact that sets may now have duplicates:

definition *compact* :: *'a set* ⇒ *'a set* **where**
compact X = *X*

lemma [*code*]: *compact*(*set xs*) = *set*(*remdups xs*)
by(*simp add*: *compact_def*)

definition *vars_acom* = *compact o vars o strip*

In order to display commands annotated with state sets, states must be translated into a printable format as sets of variable-state pairs, for the variables in the command:

definition *show_acom* :: *state set acom* ⇒ (*vname*∗*val*)*set set acom* **where**
show_acom C =
  annotate (λ*p*. (λ*s*. (λ*x*. (*x*, *s x*)) ‘ (*vars_acom C*)) ‘ *anno C p*) (*strip C*)

14.6 Examples

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

definition *C0* :: *state set acom* **where** *C0* = *annotate* (λ*p*. {}) *c0*

Collecting semantics:

value *show_acom* (((*step* {<>}) ^^ *0*) *C0*)
value *show_acom* (((*step* {<>}) ^^ *1*) *C0*)
value *show_acom* (((*step* {<>}) ^^ *2*) *C0*)
value *show_acom* (((*step* {<>}) ^^ *3*) *C0*)
value *show_acom* (((*step* {<>}) ^^ *4*) *C0*)

value *show_acom* (((*step* {<>}) ^^ *5*) *C0*)
value *show_acom* (((*step* {<>}) ^^ *6*) *C0*)
value *show_acom* (((*step* {<>}) ^^ *7*) *C0*)
value *show_acom* (((*step* {<>}) ^^ *8*) *C0*)

Small-step semantics:

value *show_acom* (((*step* {}) ^^ *0*) (*step* {<>} *C0*))
value *show_acom* (((*step* {}) ^^ *1*) (*step* {<>} *C0*))
value *show_acom* (((*step* {}) ^^ *2*) (*step* {<>} *C0*))
value *show_acom* (((*step* {}) ^^ *3*) (*step* {<>} *C0*))
value *show_acom* (((*step* {}) ^^ *4*) (*step* {<>} *C0*))
value *show_acom* (((*step* {}) ^^ *5*) (*step* {<>} *C0*))
value *show_acom* (((*step* {}) ^^ *6*) (*step* {<>} *C0*))
value *show_acom* (((*step* {}) ^^ *7*) (*step* {<>} *C0*))
value *show_acom* (((*step* {}) ^^ *8*) (*step* {<>} *C0*))

end
theory *Abs_Int_Tests*
imports *Com*
begin

14.7 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 0;;
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 100)
DO (''y'' ::= V ''x'';; ''x'' ::= Plus (V ''x'') (N 1))

definition *test6_ivl* =
''x'' ::= N 0;;
WHILE Less (N (− 1)) (V ''x'') DO ''x'' ::= Plus (V ''x'') (N 1)

end
theory *Abs_Int_init*
imports *HOL−Library.While_Combinator*
HOL−Library.Extended
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

14.8 Orderings

The basic type classes *order*, *semilattice_sup* and *order_top* are defined in *Main*, more precisely in theories *HOL.Orderings* and *HOL.Lattices*. If you view this theory with jedit, just click on the names to get there.

class *semilattice_sup_top* = *semilattice_sup* + *order_top*

instance *fun* :: (*type*, *semilattice_sup_top*) *semilattice_sup_top* **..**

instantiation *option* :: (*order*)*order*
begin

fun *less_eq_option* **where**
$Some\ x \le Some\ y = (x \le y)$ |
$None \le y = True$ |
$Some\ \_ \le None = False$

definition *less_option* **where** $x < (y::'a\ option) = (x \le y \wedge \neg\ y \le x)$

lemma *le_None*[*simp*]: $(x \le None) = (x = None)$
by (*cases x*) *simp_all*

lemma *Some_le*[*simp*]: $(Some\ x \le u) = (\exists\, y.\ u = Some\ y \wedge x \le y)$
by (*cases u*) *auto*

instance
proof (*standard*, *goal_cases*)
  case *1* **show** *?case* **by**(*rule less_option_def*)
next
  case (*2 x*) **show** *?case* **by**(*cases x*, *simp_all*)
next
  case (*3 x y z*) **thus** *?case* **by**(*cases z*, *simp*, *cases y*, *simp*, *cases x*, *auto*)
next

case (*4 x y*) **thus** *?case* **by**(*cases y*, *simp*, *cases x*, *auto*)
qed

end

instantiation *option* :: (*sup*)*sup*
begin

fun *sup_option* **where**
Some x $\sqcup$ *Some y* = *Some*(*x* $\sqcup$ *y*) |
None $\sqcup$ *y* = *y* |
x $\sqcup$ *None* = *x*

lemma *sup_None2*[*simp*]: *x* $\sqcup$ *None* = *x*
by (*cases x*) *simp_all*

instance ..

end

instantiation *option* :: (*semilattice_sup_top*)*semilattice_sup_top*
begin

definition *top_option* **where** $\top$ = *Some* $\top$

instance
proof (*standard*, *goal_cases*)
case (*4 a*) **show** *?case* **by**(*cases a*, *simp_all add*: *top_option_def*)
next
case (*1 x y*) **thus** *?case* **by**(*cases x*, *simp*, *cases y*, *simp_all*)
next
case (*2 x y*) **thus** *?case* **by**(*cases y*, *simp*, *cases x*, *simp_all*)
next
case (*3 x y z*) **thus** *?case* **by**(*cases z*, *simp*, *cases y*, *simp*, *cases x*, *simp_all*)
qed

end

lemma [*simp*]: (*Some x* < *Some y*) = (*x* < *y*)
by(*auto simp*: *less_le*)

instantiation *option* :: (*order*)*order_bot*
begin

definition *bot_option* :: *'a option* **where**
$\perp$ = *None*

instance
proof (*standard*, *goal_cases*)
  case *1* **thus** *?case* **by**(*auto simp*: *bot_option_def*)
qed

end

definition *bot* :: *com* $\Rightarrow$ *'a option acom* **where**
bot c = *annotate* (λ*p*. *None*) *c*

lemma *bot_least*: *strip C* = *c* $\Longrightarrow$ *bot c* $\leq$ *C*
by(*auto simp*: *bot_def less_eq_acom_def*)

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

14.8.1 Pre-fixpoint iteration

definition *pfp* :: ((*'a*::*order*) $\Rightarrow$ *'a*) $\Rightarrow$ *'a* $\Rightarrow$ *'a option* **where**
pfp f = *while_option* (λ*x*. $\neg$ *f x* $\leq$ *x*) *f*

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

lemma *while_least*:
fixes *q* :: *'a*::*order*
assumes $\forall$ *x*$\in$*L*.$\forall$ *y*$\in$*L*. *x* $\leq$ *y* $\longrightarrow$ *f x* $\leq$ *f y* **and** $\forall$ *x*. *x* $\in$ *L* $\longrightarrow$ *f x* $\in$ *L*
and $\forall$ *x* $\in$ *L*. *b* $\leq$ *x* **and** *b* $\in$ *L* **and** *f q* $\leq$ *q* **and** *q* $\in$ *L*
and *while_option P f b* = *Some p*
shows *p* $\leq$ *q*
using *while_option_rule*[*OF _ assms*(*7*)[*unfolded pfp_def*],
                    where *P* = %*x*. *x* $\in$ *L* $\wedge$ *x* $\leq$ *q*]
by (*metis assms*(*1*−*6*) *order_trans*)

lemma *pfp_bot_least*:
assumes $\forall$ *x*$\in${*C*. *strip C* = *c*}.$\forall$ *y*$\in${*C*. *strip C* = *c*}. *x* $\leq$ *y* $\longrightarrow$ *f x* $\leq$ *f y*
and $\forall$ *C*. *C* $\in$ {*C*. *strip C* = *c*} $\longrightarrow$ *f C* $\in$ {*C*. *strip C* = *c*}
and *f C'* $\leq$ *C' strip C'* = *c pfp f* (*bot c*) = *Some C*
shows *C* $\leq$ *C'*

by(*rule while_least*[*OF assms*(*1*,*2*) _ _ *assms*(*3*) _ *assms*(*5*)[*unfolded pfp_def*]])
  (*simp_all add*: *assms*(*4*) *bot_least*)

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** (*blast intro*: *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*

14.9 Abstract Interpretation

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

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

The interface for abstract values:

locale *Val_semilattice* =
fixes $\gamma :: 'av::semilattice\_sup\_top \Rightarrow val\ set$
  assumes *mono_gamma*: $a \le b \Longrightarrow \gamma\ a \le \gamma\ b$
  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 $'av\ st = (vname \Rightarrow 'av)$

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_fun* = *Val_semilattice* **where** $\gamma{=}\gamma$
  for $\gamma :: 'av::semilattice\_sup\_top \Rightarrow val\ set$
begin

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

definition $asem\ x\ e\ S = (case\ S\ of\ None \Rightarrow None \mid Some\ S \Rightarrow Some(S(x := aval'\ e\ S)))$

definition $step' = Step\ asem\ (\lambda b\ S.\ S)$

lemma $strip\_step'[simp]$: $strip(step'\ S\ C) = strip\ C$
by$(simp\ add{:}\ step'\_def)$

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

abbreviation $\gamma_s :: 'av\ st \Rightarrow state\ set$
where $\gamma_s == \gamma\_fun\ \gamma$

abbreviation $\gamma_o :: 'av\ st\ option \Rightarrow state\ set$
where $\gamma_o == \gamma\_option\ \gamma_s$

abbreviation $\gamma_c :: 'av\ st\ option\ acom \Rightarrow state\ set\ acom$
where $\gamma_c == map\_acom\ \gamma_o$

lemma $gamma\_s\_Top[simp]$: $\gamma_s\ \top = UNIV$
by$(simp\ add{:}\ top\_fun\_def\ \gamma\_fun\_def)$

lemma $gamma\_o\_Top[simp]$: $\gamma_o\ \top = UNIV$
by $(simp\ add{:}\ top\_option\_def)$

lemma $mono\_gamma\_s$: $f1 \le f2 \Longrightarrow \gamma_s\ f1 \subseteq \gamma_s\ f2$
by$(auto\ simp{:}\ le\_fun\_def\ \gamma\_fun\_def\ dest{:}\ mono\_gamma)$

lemma $mono\_gamma\_o$:
$S1 \le S2 \Longrightarrow \gamma_o\ S1 \subseteq \gamma_o\ S2$
by$(induction\ S1\ S2\ rule{:}\ less\_eq\_option.induct)(simp\_all\ add{:}\ mono\_gamma\_s)$

lemma $mono\_gamma\_c$: $C1 \le C2 \Longrightarrow \gamma_c\ C1 \le \gamma_c\ C2$
by $(simp\ add{:}\ less\_eq\_acom\_def\ mono\_gamma\_o\ size\_annos\ anno\_map\_acom\ size\_annos\_same[of\ C1\ C2])$

Correctness:

lemma $aval'\_correct$: $s \in \gamma_s\ S \Longrightarrow aval\ a\ s \in \gamma(aval'\ a\ S)$
by $(induct\ a)\ (auto\ simp{:}\ gamma\_num'\ gamma\_plus'\ \gamma\_fun\_def)$

lemma in_gamma_update: $[\![\ s \in \gamma_s\ S;\ i \in \gamma\ a\ ]\!] \Longrightarrow s(x := i) \in \gamma_s(S(x := a))$

by(*simp add*: γ*_fun_def*)

lemma *gamma_Step_subcomm*:
  assumes $\bigwedge x\ e\ S.\ f1\ x\ e\ (\gamma_o\ S) \subseteq \gamma_o\ (f2\ x\ e\ S)$ $\bigwedge b\ S.\ g1\ b\ (\gamma_o\ S) \subseteq \gamma_o\ (g2\ b\ S)$
  shows $Step\ f1\ g1\ (\gamma_o\ S)\ (\gamma_c\ C) \le \gamma_c\ (Step\ f2\ g2\ S\ C)$
by (*induction C arbitrary*: *S*) (*auto simp*: *mono_gamma_o assms*)

lemma *step_step'*: $step\ (\gamma_o\ S)\ (\gamma_c\ C) \le \gamma_c\ (step'\ S\ C)$
unfolding *step_def step'_def*
by(*rule gamma_Step_subcomm*)
  (*auto simp*: *aval'_correct in_gamma_update asem_def split*: *option.splits*)

lemma *AI_correct*: $AI\ c = Some\ C \Longrightarrow CS\ c \le \gamma_c\ C$
proof(*simp add*: *CS_def AI_def*)
  assume *1*: $pfp\ (step'\ \top)\ (bot\ c) = Some\ C$
  have *pfp'*: $step'\ \top\ C \le C$ **by**(*rule pfp_pfp*[*OF 1*])
  have *2*: $step\ (\gamma_o\ \top)\ (\gamma_c\ C) \le \gamma_c\ C$ — transfer the pfp'
  proof(*rule order_trans*)
    show $step\ (\gamma_o\ \top)\ (\gamma_c\ C) \le \gamma_c\ (step'\ \top\ C)$ **by**(*rule step_step'*)
    show ... $\le \gamma_c\ C$ **by** (*metis mono_gamma_c*[*OF pfp'*])
  qed
  have *3*: $strip\ (\gamma_c\ C) = c$ **by**(*simp add*: *strip_pfp*[*OF _ 1*] *step'_def*)
  have $lfp\ c\ (step\ (\gamma_o\ \top)) \le \gamma_c\ C$
    by(*rule lfp_lowerbound*[*simplified*,**where** $f{=}step\ (\gamma_o\ \top)$, *OF 3 2*])
  thus $lfp\ c\ (step\ UNIV) \le \gamma_c\ C$ **by** *simp*
qed

end

14.9.1 Monotonicity

locale *Abs_Int_fun_mono* = *Abs_Int_fun* +
assumes *mono_plus'*: $a1 \le b1 \Longrightarrow a2 \le b2 \Longrightarrow plus'\ a1\ a2 \le plus'\ b1\ b2$
begin

lemma *mono_aval'*: $S \le S' \Longrightarrow aval'\ e\ S \le aval'\ e\ S'$
by(*induction e*)(*auto simp*: *le_fun_def mono_plus'*)

lemma *mono_update*: $a \le a' \Longrightarrow S \le S' \Longrightarrow S(x := a) \le S'(x := a')$
by(*simp add*: *le_fun_def*)

lemma *mono_step'*: $S1 \le S2 \Longrightarrow C1 \le C2 \Longrightarrow step'\ S1\ C1 \le step'\ S2\ C2$

unfolding *step'_def*
by(*rule mono2_Step*)
(*auto simp*: *mono_update mono_aval' asem_def split*: *option.split*)

lemma *mono_step'_top*: $C \le C' \Longrightarrow step' \top\ C \le step' \top\ C'$
by (*metis mono_step' order_refl*)

lemma *AI_least_pfp*: **assumes** $AI\ c = Some\ C\ step' \top\ C' \le C'\ strip\ C' = c$
shows $C \le C'$
by(*rule pfp_bot_least*[*OF* _ _ *assms*(*2*,*3*) *assms*(*1*)[*unfolded AI_def*]])
(*simp_all add*: *mono_step'_top*)

end

instantiation *acom* :: (*type*) *vars*
begin

definition *vars_acom* = *vars o strip*

instance ..

end

lemma *finite_Cvars*: *finite*(*vars*(*C*::'*a acom*))
by(*simp add*: *vars_acom_def*)

14.9.2 Termination

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

by (*blast intro*: *I mono*)
qed

lemma *le_iff_le_annos*: $C1 \leq C2 \longleftrightarrow$
$strip\ C1 = strip\ C2 \wedge (\forall\ i{<}size(annos\ C1).\ annos\ C1\ !\ i \leq annos\ C2\ !\ i)$
by(*simp add*: *less_eq_acom_def anno_def*)

locale *Measure1_fun* =
fixes $m :: 'av{::}top \Rightarrow nat$
fixes $h :: nat$
assumes h: $m\ x \leq h$
begin

definition $m\_s :: 'av\ st \Rightarrow vname\ set \Rightarrow nat$ (m_s) **where**
$m\_s\ S\ X = (\sum\ x \in X.\ m(S\ x))$

lemma *m_s_h*: $finite\ X \Longrightarrow m\_s\ S\ X \leq h * card\ X$
by(*simp add*: *m_s_def*) (*metis mult.commute of_nat_id sum_bounded_above*[*OF h*])

fun $m\_o :: 'av\ st\ option \Rightarrow vname\ set \Rightarrow nat$ (m_o) **where**
$m\_o\ (Some\ S)\ X = m\_s\ S\ X\ |$
$m\_o\ None\ X = h * card\ X + 1$

lemma *m_o_h*: $finite\ X \Longrightarrow m\_o\ opt\ X \leq (h{*}card\ X + 1)$
by(*cases opt*)(*auto simp add*: *m_s_h le_SucI dest*: *m_s_h*)

definition $m\_c :: 'av\ st\ option\ acom \Rightarrow nat$ (m_c) **where**
$m\_c\ C = sum\_list\ (map\ (\lambda a.\ m\_o\ a\ (vars\ C))\ (annos\ C))$

Upper complexity bound:

lemma *m_c_h*: $m\_c\ C \leq size(annos\ C) * (h * card(vars\ C) + 1)$
proof−
  let $?X = vars\ C$ **let** $?n = card\ ?X$ **let** $?a = size(annos\ C)$
  have $m\_c\ C = (\sum i{<}?a.\ m\_o\ (annos\ C\ !\ i)\ ?X)$
    by(*simp add*: *m_c_def sum_list_sum_nth atLeast0LessThan*)
  also have $\ldots \leq (\sum i{<}?a.\ h * ?n + 1)$
    apply(*rule sum_mono*) **using** *m_o_h*[*OF finite_Cvars*] **by** *simp*
  also have $\ldots = ?a * (h * ?n + 1)$ **by** *simp*
  finally show *?thesis* **.**
qed

end

locale *Measure_fun* = *Measure1_fun* **where** $m{=}m$
  for m :: $'av{::}semilattice\_sup\_top \Rightarrow nat$ +
assumes *m2*: $x < y \Longrightarrow m\ x > m\ y$
begin

The predicates *top_on_ty a X* that follow describe that any abstract state in a maps all variables in X to $\top$. This is an important invariant for the termination proof where we argue that only the finitely many variables in the program change. That the others do not change follows because they remain $\top$.

fun *top_on_st* :: $'av\ st \Rightarrow vname\ set \Rightarrow bool$ (top'_on_s) **where**
$top\_on\_st\ S\ X = (\forall x{\in}X.\ S\ x = \top)$

fun *top_on_opt* :: $'av\ st\ option \Rightarrow vname\ set \Rightarrow bool$ (top'_on_o) **where**
$top\_on\_opt\ (Some\ S)\ X = top\_on\_st\ S\ X$ |
$top\_on\_opt\ None\ X = True$

definition *top_on_acom* :: $'av\ st\ option\ acom \Rightarrow vname\ set \Rightarrow bool$ (top'_on_c)
where
$top\_on\_acom\ C\ X = (\forall a \in set(annos\ C).\ top\_on\_opt\ a\ X)$

lemma *top_on_top*: $top\_on\_opt\ \top\ X$
by(*auto simp*: *top_option_def*)

lemma *top_on_bot*: $top\_on\_acom\ (bot\ c)\ X$
by(*auto simp add*: *top_on_acom_def bot_def*)

lemma *top_on_post*: $top\_on\_acom\ C\ X \Longrightarrow top\_on\_opt\ (post\ C)\ X$
by(*simp add*: *top_on_acom_def post_in_annos*)

lemma *top_on_acom_simps*:
  $top\_on\_acom\ (SKIP\ \{Q\})\ X = top\_on\_opt\ Q\ X$
  $top\_on\_acom\ (x ::= e\ \{Q\})\ X = top\_on\_opt\ Q\ X$
  $top\_on\_acom\ (C1;;C2)\ X = (top\_on\_acom\ C1\ X \wedge top\_on\_acom\ C2\ X)$
  $top\_on\_acom\ (IF\ b\ THEN\ \{P1\}\ C1\ ELSE\ \{P2\}\ C2\ \{Q\})\ X =$
  $(top\_on\_opt\ P1\ X \wedge top\_on\_acom\ C1\ X \wedge top\_on\_opt\ P2\ X \wedge top\_on\_acom\ C2\ X \wedge top\_on\_opt\ Q\ X)$
  $top\_on\_acom\ (\{I\}\ WHILE\ b\ DO\ \{P\}\ C\ \{Q\})\ X =$
  $(top\_on\_opt\ I\ X \wedge top\_on\_acom\ C\ X \wedge top\_on\_opt\ P\ X \wedge top\_on\_opt\ Q\ X)$
by(*auto simp add*: *top_on_acom_def*)

lemma *top_on_sup*:
  top_on_opt o1 X $\Longrightarrow$ *top_on_opt o2 X* $\Longrightarrow$ *top_on_opt* (*o1* $\sqcup$ *o2*) *X*
apply(*induction o1 o2 rule*: *sup_option.induct*)
apply(*auto*)
done

lemma *top_on_Step*: **fixes** *C* :: *'av st option acom*
assumes !!*x e S*. ⟦*top_on_opt S X*; *x* $\notin$ *X*; *vars e* $\subseteq$ $-X$⟧ $\Longrightarrow$ *top_on_opt* (*f x e S*) *X*
        !!*b S*. *top_on_opt S X* $\Longrightarrow$ *vars b* $\subseteq$ $-X$ $\Longrightarrow$ *top_on_opt* (*g b S*) *X*
shows ⟦ *vars C* $\subseteq$ $-X$; *top_on_opt S X*; *top_on_acom C X* ⟧ $\Longrightarrow$ *top_on_acom* (*Step f g S C*) *X*
proof(*induction C arbitrary*: *S*)
qed (*auto simp*: *top_on_acom_simps vars_acom_def top_on_post top_on_sup assms*)

lemma *m1*: $x \le y \Longrightarrow m\ x \ge m\ y$
by(*auto simp*: *le_less m2*)

lemma *m_s2_rep*: **assumes** *finite*(*X*) **and** *S1* = *S2 on* $-X$ **and** $\forall x$. *S1 x* $\le$ *S2 x* **and** *S1* $\neq$ *S2*
shows ($\sum x{\in}X$. *m* (*S2 x*)) < ($\sum x{\in}X$. *m* (*S1 x*))
proof−
  from *assms*(*3*) **have** *1*: $\forall x{\in}X$. *m*(*S1 x*) $\ge$ *m*(*S2 x*) **by** (*simp add*: *m1*)
  from *assms*(*2*,*3*,*4*) **have** $\exists x{\in}X$. *S1 x* < *S2 x*
    by(*simp add*: *fun_eq_iff*) (*metis Compl_iff le_neq_trans*)
  hence *2*: $\exists x{\in}X$. *m*(*S1 x*) > *m*(*S2 x*) **by** (*metis m2*)
  from *sum_strict_mono_ex1*[*OF* ⟨*finite X*⟩ *1 2*]
  show ($\sum x{\in}X$. *m* (*S2 x*)) < ($\sum x{\in}X$. *m* (*S1 x*)) **.**
qed

lemma *m_s2*: *finite*(*X*) $\Longrightarrow$ *S1* = *S2 on* $-X$ $\Longrightarrow$ *S1* < *S2* $\Longrightarrow$ *m_s S1 X* > *m_s S2 X*
apply(*auto simp add*: *less_fun_def m_s_def*)
apply(*simp add*: *m_s2_rep le_fun_def*)
done

lemma *m_o2*: *finite X* $\Longrightarrow$ *top_on_opt o1* ($-X$) $\Longrightarrow$ *top_on_opt o2* ($-X$) $\Longrightarrow$
  o1 < *o2* $\Longrightarrow$ *m_o o1 X* > *m_o o2 X*
proof(*induction o1 o2 rule*: *less_eq_option.induct*)
  case *1* **thus** *?case* **by** (*auto simp*: *m_s2 less_option_def*)
next
  case *2* **thus** *?case* **by**(*auto simp*: *less_option_def le_imp_less_Suc m_s_h*)

next
  case *3* **thus** *?case* **by** (*auto simp*: *less_option_def*)
qed

lemma *m_o1*: *finite X* $\Longrightarrow$ *top_on_opt o1* $(-X)$ $\Longrightarrow$ *top_on_opt o2* $(-X)$ $\Longrightarrow$
  o1 $\leq$ *o2* $\Longrightarrow$ *m_o o1 X* $\geq$ *m_o o2 X*
by(*auto simp*: *le_less m_o2*)

lemma *m_c2*: *top_on_acom C1* (−*vars C1*) $\Longrightarrow$ *top_on_acom C2* (−*vars C2*) $\Longrightarrow$
  C1 < *C2* $\Longrightarrow$ *m_c C1* > *m_c C2*
proof(*auto simp add*: *le_iff_le_annos size_annos_same*[*of C1 C2*] *vars_acom_def less_acom_def*)
  let *?X* = *vars*(*strip C2*)
  assume *top*: *top_on_acom C1* (− *vars*(*strip C2*)) *top_on_acom C2* (− *vars*(*strip C2*))
  and *strip_eq*: *strip C1* = *strip C2*
  and *0*: $\forall$ *i*<*size*(*annos C2*). *annos C1* ! *i* $\leq$ *annos C2* ! *i*
  hence *1*: $\forall$ *i*<*size*(*annos C2*). *m_o* (*annos C1* ! *i*) *?X* $\geq$ *m_o* (*annos C2* ! *i*) *?X*
    apply (*auto simp*: *all_set_conv_all_nth vars_acom_def top_on_acom_def*)
    by (*metis* (*lifting*, *no_types*) *finite_cvars m_o1 size_annos_same2*)
  fix *i* **assume** *i*: *i* < *size*(*annos C2*) $\neg$ *annos C2* ! *i* $\leq$ *annos C1* ! *i*
  have *topo1*: *top_on_opt* (*annos C1* ! *i*) (− *?X*)
    using *i*(*1*) *top*(*1*) **by**(*simp add*: *top_on_acom_def size_annos_same*[*OF strip_eq*])
  have *topo2*: *top_on_opt* (*annos C2* ! *i*) (− *?X*)
    using *i*(*1*) *top*(*2*) **by**(*simp add*: *top_on_acom_def size_annos_same*[*OF strip_eq*])
  from *i* **have** *m_o* (*annos C1* ! *i*) *?X* > *m_o* (*annos C2* ! *i*) *?X* (**is** *?P i*)
    by (*metis 0 less_option_def m_o2*[*OF finite_cvars topo1*] *topo2*)
  hence *2*: $\exists$ *i* < *size*(*annos C2*). *?P i* **using** ‹*i* < *size*(*annos C2*)› **by** *blast*
  have ($\sum$ *i*<*size*(*annos C2*). *m_o* (*annos C2* ! *i*) *?X*)
      < ($\sum$ *i*<*size*(*annos C2*). *m_o* (*annos C1* ! *i*) *?X*)
    apply(*rule sum_strict_mono_ex1*) **using** *1 2* **by** (*auto*)
  thus *?thesis*
    by(*simp add*: *m_c_def vars_acom_def strip_eq sum_list_sum_nth atLeast0LessThan size_annos_same*[*OF strip_eq*])
qed

end

locale *Abs_Int_fun_measure* =
  Abs_Int_fun_mono **where** γ=γ + *Measure_fun* **where** *m*=*m*
  for γ :: *'av*::*semilattice_sup_top* ⇒ *val set* **and** *m* :: *'av* ⇒ *nat*
begin

lemma *top_on_step'*: *top_on_acom C* (−*vars C*) ⟹ *top_on_acom* (*step'* ⊤ *C*) (−*vars C*)
unfolding *step'_def*
by(*rule top_on_Step*)
  (*auto simp add*: *top_option_def asem_def split*: *option.splits*)

lemma *AI_Some_measure*: ∃ *C*. *AI c* = *Some C*
unfolding *AI_def*
apply(*rule pfp_termination*[**where** *I* = λ*C*. *top_on_acom C* (− *vars C*) **and** *m*=*m_c*])
apply(*simp_all add*: *m_c2 mono_step'_top bot_least top_on_bot*)
using *top_on_step'* **apply**(*auto simp add*: *vars_acom_def*)
done

end

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

end

theory *Abs_State*
imports *Abs_Int0*
begin

type_synonym *'a st_rep* = (*vname* ∗ *'a*) *list*

fun *fun_rep* :: (*'a*::*top*) *st_rep* ⇒ *vname* ⇒ *'a* **where**
fun_rep [] = (λ*x*. ⊤) |
fun_rep ((*x*,*a*)#*ps*) = (*fun_rep ps*) (*x* := *a*)

lemma *fun_rep_map_of*[*code*]: — original def is too slow
  fun_rep ps = (%*x*. *case map_of ps x of None* ⇒ ⊤ | *Some a* ⇒ *a*)
by(*induction ps rule*: *fun_rep.induct*) *auto*

definition *eq_st* :: (*'a*::*top*) *st_rep* ⇒ *'a st_rep* ⇒ *bool* **where**
eq_st S1 S2 = (*fun_rep S1* = *fun_rep S2*)

hide_type *st* — hide previous def to avoid long names
declare [[*typedef_overloaded*]] — allow quotient types to depend on classes

quotient_type $'a\ st = ('a{::}top)\ st\_rep\ /\ eq\_st$
morphisms *rep_st St*
by (*metis eq_st_def equivpI reflpI sympI transpI*)

lift_definition $update :: ('a{::}top)\ st \Rightarrow vname \Rightarrow 'a \Rightarrow 'a\ st$
is $\lambda ps\ x\ a.\ (x,a)\#ps$
by(*auto simp*: *eq_st_def*)

lift_definition $fun :: ('a{::}top)\ st \Rightarrow vname \Rightarrow 'a$ **is** *fun_rep*
by(*simp add*: *eq_st_def*)

definition $show\_st :: vname\ set \Rightarrow ('a{::}top)\ st \Rightarrow (vname * 'a)set$ **where**
$show\_st\ X\ S = (\lambda x.\ (x,\ fun\ S\ x))\ `\ X$

definition $show\_acom\ C = map\_acom\ (map\_option\ (show\_st\ (vars(strip\ C))))\ C$
definition $show\_acom\_opt = map\_option\ show\_acom$

lemma *fun_update*[*simp*]: $fun\ (update\ S\ x\ y) = (fun\ S)(x{:=}y)$
by *transfer auto*

definition $\gamma\_st :: (('a{::}top) \Rightarrow 'b\ set) \Rightarrow 'a\ st \Rightarrow (vname \Rightarrow 'b)\ set$ **where**
$\gamma\_st\ \gamma\ F = \{f.\ \forall x.\ f\ x \in \gamma(fun\ F\ x)\}$

instantiation $st :: (order\_top)\ order$
begin

definition $less\_eq\_st\_rep :: 'a\ st\_rep \Rightarrow 'a\ st\_rep \Rightarrow bool$ **where**
$less\_eq\_st\_rep\ ps1\ ps2 =$
$((\forall x \in set(map\ fst\ ps1) \cup set(map\ fst\ ps2).\ fun\_rep\ ps1\ x \le fun\_rep\ ps2\ x))$

lemma *less_eq_st_rep_iff*:
$less\_eq\_st\_rep\ r1\ r2 = (\forall x.\ fun\_rep\ r1\ x \le fun\_rep\ r2\ x)$
apply(*auto simp*: *less_eq_st_rep_def fun_rep_map_of split*: *option.split*)
apply (*metis Un_iff map_of_eq_None_iff option.distinct*(*1*))
apply (*metis Un_iff map_of_eq_None_iff option.distinct*(*1*))
done

corollary *less_eq_st_rep_iff_fun*:
$less\_eq\_st\_rep\ r1\ r2 = (fun\_rep\ r1 \le fun\_rep\ r2)$

by (*metis less_eq_st_rep_iff le_fun_def*)

lift_definition *less_eq_st* :: $'a$ *st* $\Rightarrow$ $'a$ *st* $\Rightarrow$ *bool* **is** *less_eq_st_rep*
by(*auto simp add*: *eq_st_def less_eq_st_rep_iff*)

definition *less_st* **where** $F < (G{::}'a\ st) = (F \leq G \wedge \neg\ G \leq F)$

instance
proof (*standard*, *goal_cases*)
  case *1* **show** *?case* **by**(*rule less_st_def*)
next
  case *2* **show** *?case* **by** *transfer* (*auto simp*: *less_eq_st_rep_def*)
next
  case *3* **thus** *?case* **by** *transfer* (*metis less_eq_st_rep_iff order_trans*)
next
  case *4* **thus** *?case*
    by *transfer* (*metis less_eq_st_rep_iff eq_st_def fun_eq_iff antisym*)
qed

end

lemma *le_st_iff*: $(F \leq G) = (\forall x.\ fun\ F\ x \leq fun\ G\ x)$
by *transfer* (*rule less_eq_st_rep_iff*)

fun *map2_st_rep* :: $('a{::}top \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a$ *st_rep* $\Rightarrow$ $'a$ *st_rep* $\Rightarrow$ $'a$ *st_rep*
where
map2_st_rep f [] *ps2* = *map* (%(*x*,*y*). (*x*, *f* $\top$ *y*)) *ps2* |
map2_st_rep f ((*x*,*y*)#*ps1*) *ps2* =
  (*let y2* = *fun_rep ps2 x*
   in (*x*,*f y y2*) # *map2_st_rep f ps1 ps2*)

lemma *fun_rep_map2_rep*[*simp*]: $f\ \top\ \top = \top \Longrightarrow$
  fun_rep (*map2_st_rep f ps1 ps2*) = (λx. *f* (*fun_rep ps1 x*) (*fun_rep ps2 x*))
apply(*induction f ps1 ps2 rule*: *map2_st_rep.induct*)
apply(*simp add*: *fun_rep_map_of map_of_map fun_eq_iff split*: *option.split*)
apply(*fastforce simp*: *fun_rep_map_of fun_eq_iff split*:*option.splits*)
done

instantiation *st* :: (*semilattice_sup_top*) *semilattice_sup_top*
begin

lift_definition *sup_st* :: $'a$ *st* $\Rightarrow$ $'a$ *st* $\Rightarrow$ $'a$ *st* **is** *map2_st_rep* ($\sqcup$)
by (*simp add*: *eq_st_def*)

lift_definition *top_st* :: *'a st* **is** [] **.**

instance
proof (*standard*, *goal_cases*)
  case *1* **show** *?case* **by** *transfer* (*simp add*:*less_eq_st_rep_iff*)
next
  case *2* **show** *?case* **by** *transfer* (*simp add*:*less_eq_st_rep_iff*)
next
  case *3* **thus** *?case* **by** *transfer* (*simp add*:*less_eq_st_rep_iff*)
next
  case *4* **show** *?case* **by** *transfer* (*simp add*:*less_eq_st_rep_iff fun_rep_map_of*)
qed

end

lemma *fun_top*: *fun* $\top$ = (λx. $\top$)
by *transfer simp*

lemma *mono_update*[*simp*]:
  a1 $\leq$ *a2* $\Longrightarrow$ *S1* $\leq$ *S2* $\Longrightarrow$ *update S1 x a1* $\leq$ *update S2 x a2*
by *transfer* (*auto simp add*: *less_eq_st_rep_def*)

lemma *mono_fun*: *S1* $\leq$ *S2* $\Longrightarrow$ *fun S1 x* $\leq$ *fun S2 x*
by *transfer* (*simp add*: *less_eq_st_rep_iff*)

locale *Gamma_semilattice* = *Val_semilattice* **where** γ=γ
  for γ :: *'av*::*semilattice_sup_top* $\Rightarrow$ *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 $\top$ = *UNIV*
by(*auto simp*: γ*_st_def fun_top*)

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

lemma *mono_gamma_s*: $f \le g \Longrightarrow \gamma_s\ f \subseteq \gamma_s\ g$
by(*simp add*:γ*_st_def le_st_iff subset_iff*) (*metis mono_gamma subsetD*)

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

lemma *mono_gamma_c*: $C1 \le C2 \Longrightarrow \gamma_c\ C1 \le \gamma_c\ C2$
by (*simp add*: *less_eq_acom_def mono_gamma_o size_annos anno_map_acom size_annos_same*[*of C1 C2*])

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

end

end

theory *Abs_Int1*
imports *Abs_State*
begin

14.10 Computable Abstract Interpretation

Abstract interpretation over type *st* instead of functions.

context *Gamma_semilattice*
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′_correct*: $s \in \gamma_s\ S \Longrightarrow aval\ a\ s \in \gamma(aval'\ a\ S)$
by (*induction a*) (*auto simp*: *gamma_num′ gamma_plus′* γ*_st_def*)

lemma *gamma_Step_subcomm*: **fixes** *C1 C2* :: *′a*::*semilattice_sup acom*
  assumes !!*x e S*. $f1\ x\ e\ (\gamma_o\ S) \subseteq \gamma_o\ (f2\ x\ e\ S)$
    !!*b S*. $g1\ b\ (\gamma_o\ S) \subseteq \gamma_o\ (g2\ b\ S)$
  shows $Step\ f1\ g1\ (\gamma_o\ S)\ (\gamma_c\ C) \le \gamma_c\ (Step\ f2\ g2\ S\ C)$
proof(*induction C arbitrary*: *S*)
qed (*auto simp*: *assms intro*!: *mono_gamma_o sup_ge1 sup_ge2*)

lemma *in_gamma_update*: $[\![\ s \in \gamma_s\ S;\ i \in \gamma\ a\ ]\!] \Longrightarrow s(x := i) \in \gamma_s(update$ $S\ x\ a)$
by(*simp add*: *γ_st_def*)

end

locale *Abs_Int* = *Gamma_semilattice* **where** $\gamma{=}\gamma$
  for γ :: *'av*::*semilattice_sup_top* $\Rightarrow$ *val set*
begin

definition *step'* = *Step*
  ($\lambda x\ e\ S.\ case\ S\ of\ None \Rightarrow None\ |\ Some\ S \Rightarrow Some(update\ S\ x\ (aval'\ e$ $S)))$
  $(\lambda b\ S.\ S)$

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(*simp add*: *step'_def*)

Correctness:

lemma *step_step'*: *step* $(\gamma_o\ S)\ (\gamma_c\ C) \leq \gamma_c$ (*step' S C*)
unfolding *step_def step'_def*
by(*rule gamma_Step_subcomm*)
  (*auto simp*: *intro*!: *aval'_correct in_gamma_update split*: *option.splits*)

lemma *AI_correct*: *AI c* = *Some C* $\Longrightarrow$ *CS c* $\leq \gamma_c$ *C*
proof(*simp add*: *CS_def AI_def*)
  assume *1*: *pfp* (*step'* $\top$) (*bot c*) = *Some C*
  have *pfp'*: *step'* $\top$ *C* $\leq$ *C* **by**(*rule pfp_pfp*[*OF 1*])
  have *2*: *step* $(\gamma_o\ \top)\ (\gamma_c\ C) \leq \gamma_c\ C$ — transfer the pfp'
  proof(*rule order_trans*)
    show *step* $(\gamma_o\ \top)\ (\gamma_c\ C) \leq \gamma_c$ (*step'* $\top$ *C*) **by**(*rule step_step'*)
    show ... $\leq \gamma_c$ *C* **by** (*metis mono_gamma_c*[*OF pfp'*])
  qed
  have *3*: *strip* $(\gamma_c\ C)$ = *c* **by**(*simp add*: *strip_pfp*[*OF _ 1*] *step'_def*)
  have *lfp c* (*step* $(\gamma_o\ \top)) \leq \gamma_c$ *C*
    by(*rule lfp_lowerbound*[*simplified*,**where** *f*=*step* $(\gamma_o\ \top)$, *OF 3 2*])
  thus *lfp c* (*step UNIV*) $\leq \gamma_c$ *C* **by** *simp*
qed

end

14.10.1 Monotonicity

locale *Abs_Int_mono* = *Abs_Int* +
assumes *mono_plus'*: *a1* ≤ *b1* ⟹ *a2* ≤ *b2* ⟹ *plus' a1 a2* ≤ *plus' b1 b2*
begin

lemma *mono_aval'*: *S1* ≤ *S2* ⟹ *aval' e S1* ≤ *aval' e S2*
by(*induction e*) (*auto simp*: *mono_plus' mono_fun*)

theorem *mono_step'*: *S1* ≤ *S2* ⟹ *C1* ≤ *C2* ⟹ *step' S1 C1* ≤ *step' S2 C2*
unfolding *step'_def*
by(*rule mono2_Step*) (*auto simp*: *mono_aval' split*: *option.split*)

lemma *mono_step'_top*: *C* ≤ *C'* ⟹ *step'* ⊤ *C* ≤ *step'* ⊤ *C'*
by (*metis mono_step' order_refl*)

lemma *AI_least_pfp*: **assumes** *AI c* = *Some C step'* ⊤ *C'* ≤ *C' strip C'* = *c*
shows *C* ≤ *C'*
by(*rule pfp_bot_least*[*OF* _ _ *assms*(*2*,*3*) *assms*(*1*)[*unfolded AI_def*]])
(*simp_all add*: *mono_step'_top*)

end

14.10.2 Termination

locale *Measure1* =
fixes *m* :: *'av*::*order_top* ⇒ *nat*
fixes *h* :: *nat*
assumes *h*: *m x* ≤ *h*
begin

definition *m_s* :: *'av st* ⇒ *vname set* ⇒ *nat* (m_s) **where**
m_s S X = (∑ *x* ∈ *X*. *m*(*fun S x*))

lemma *m_s_h*: *finite X* ⟹ *m_s S X* ≤ *h* ∗ *card X*
by(*simp add*: *m_s_def*) (*metis mult.commute of_nat_id sum_bounded_above*[*OF h*])

definition *m_o* :: *'av st option* ⇒ *vname set* ⇒ *nat* (m_o) **where**

$m\_o\ opt\ X = (case\ opt\ of\ None \Rightarrow h * card\ X + 1 \mid Some\ S \Rightarrow m\_s\ S\ X)$

lemma m_o_h: $finite\ X \Longrightarrow m\_o\ opt\ X \le (h{*}card\ X + 1)$
by$(auto\ simp\ add{:}\ m\_o\_def\ m\_s\_h\ le\_SucI\ split{:}\ option.split\ dest{:}m\_s\_h)$

definition $m\_c :: {'av}\ st\ option\ acom \Rightarrow nat\ (m_c)$ **where**
$m\_c\ C = sum\_list\ (map\ (\lambda a.\ m\_o\ a\ (vars\ C))\ (annos\ C))$

Upper complexity bound:

lemma m_c_h: $m\_c\ C \le size(annos\ C) * (h * card(vars\ C) + 1)$
proof−
  let $?X = vars\ C$ **let** $?n = card\ ?X$ **let** $?a = size(annos\ C)$
  have $m\_c\ C = (\sum i{<}?a.\ m\_o\ (annos\ C\ !\ i)\ ?X)$
    by$(simp\ add{:}\ m\_c\_def\ sum\_list\_sum\_nth\ atLeast0LessThan)$
  also have $\ldots \le (\sum i{<}?a.\ h * ?n + 1)$
    apply$(rule\ sum\_mono)$ **using** $m\_o\_h[OF\ finite\_Cvars]$ **by** $simp$
  also have $\ldots = ?a * (h * ?n + 1)$ **by** $simp$
  finally show $?thesis$ **.**
qed

end

fun $top\_on\_st :: {'a}{::}order\_top\ st \Rightarrow vname\ set \Rightarrow bool\ (top'\_on_s)$ **where**
$top\_on\_st\ S\ X = (\forall x{\in}X.\ fun\ S\ x = \top)$

fun $top\_on\_opt :: {'a}{::}order\_top\ st\ option \Rightarrow vname\ set \Rightarrow bool\ (top'\_on_o)$
where
$top\_on\_opt\ (Some\ S)\ \ X = top\_on\_st\ S\ X \mid$
$top\_on\_opt\ None\ X = True$

definition $top\_on\_acom :: {'a}{::}order\_top\ st\ option\ acom \Rightarrow vname\ set \Rightarrow bool$
(top'_on_c) **where**
$top\_on\_acom\ C\ X = (\forall\, a \in set(annos\ C).\ top\_on\_opt\ a\ X)$

lemma top_on_top: $top\_on\_opt\ (\top{::}\_\ st\ option)\ X$
by$(auto\ simp{:}\ top\_option\_def\ fun\_top)$

lemma top_on_bot: $top\_on\_acom\ (bot\ c)\ X$
by$(auto\ simp\ add{:}\ top\_on\_acom\_def\ bot\_def)$

lemma top_on_post: $top\_on\_acom\ C\ X \Longrightarrow top\_on\_opt\ (post\ C)\ X$
by$(simp\ add{:}\ top\_on\_acom\_def\ post\_in\_annos)$

lemma $top\_on\_acom\_simps$:

top_on_acom ($SKIP$ {Q}) X = top_on_opt Q X
top_on_acom (x ::= e {Q}) X = top_on_opt Q X
top_on_acom ($C1$;;$C2$) X = (top_on_acom $C1$ X $\wedge$ top_on_acom $C2$ X)
top_on_acom (IF b $THEN$ {$P1$} $C1$ $ELSE$ {$P2$} $C2$ {Q}) X =
(top_on_opt $P1$ X $\wedge$ top_on_acom $C1$ X $\wedge$ top_on_opt $P2$ X $\wedge$ top_on_acom $C2$ X $\wedge$ top_on_opt Q X)
top_on_acom ({I} $WHILE$ b DO {P} C {Q}) X =
(top_on_opt I X $\wedge$ top_on_acom C X $\wedge$ top_on_opt P X $\wedge$ top_on_opt Q X)
by($auto$ $simp$ add: $top\_on\_acom\_def$)

lemma top_on_sup:
top_on_opt $o1$ X $\Longrightarrow$ top_on_opt $o2$ X $\Longrightarrow$ top_on_opt ($o1$ $\sqcup$ $o2$ :: _ st $option$) X
apply($induction$ $o1$ $o2$ $rule$: $sup\_option.induct$)
apply($auto$)
by $transfer$ $simp$

lemma top_on_Step: **fixes** C :: ($'a$::$semilattice\_sup\_top$)st $option$ $acom$
assumes !!x e S. $[\![top\_on\_opt\ S\ X;\ x \notin X;\ vars\ e \subseteq -X]\!] \Longrightarrow top\_on\_opt$ (f x e S) X
!!b S. top_on_opt S X $\Longrightarrow$ $vars$ b $\subseteq -X \Longrightarrow$ top_on_opt (g b S) X
shows $[\![$ $vars$ $C \subseteq -X$; top_on_opt S X; top_on_acom C X $]\!] \Longrightarrow top\_on\_acom$ ($Step$ f g S C) X
proof($induction$ C $arbitrary$: S)
qed ($auto$ $simp$: $top\_on\_acom\_simps$ $vars\_acom\_def$ top_on_post top_on_sup $assms$)

locale $Measure$ = $Measure1$ +
assumes $m2$: $x < y \Longrightarrow m\ x > m\ y$
begin

lemma $m1$: $x \leq y \Longrightarrow m\ x \geq m\ y$
by($auto$ $simp$: le_less $m2$)

lemma m_s2_rep: **assumes** $finite(X)$ **and** $S1 = S2\ on\ -X$ **and** $\forall x.\ S1\ x \leq S2\ x$ **and** $S1 \neq S2$
shows $(\sum x{\in}X.\ m\ (S2\ x)) < (\sum x{\in}X.\ m\ (S1\ x))$
proof−
from $assms(3)$ **have** 1: $\forall x{\in}X.\ m(S1\ x) \geq m(S2\ x)$ **by** ($simp$ add: $m1$)
from $assms(2,3,4)$ **have** $\exists x{\in}X.\ S1\ x < S2\ x$
by($simp$ add: fun_eq_iff) ($metis$ $Compl\_iff$ le_neq_trans)
hence 2: $\exists x{\in}X.\ m(S1\ x) > m(S2\ x)$ **by** ($metis$ $m2$)

from *sum_strict_mono_ex1*[*OF* ‹*finite X*› *1 2*]
show $(\sum x{\in}X.\ m\ (S2\ x)) < (\sum x{\in}X.\ m\ (S1\ x))$ **.**
qed

lemma *m_s2*: *finite*(*X*) $\Longrightarrow$ *fun S1* = *fun S2 on* $-X$
$\Longrightarrow$ *S1* < *S2* $\Longrightarrow$ *m_s S1 X* > *m_s S2 X*
apply(*auto simp add*: *less_st_def m_s_def*)
apply (*transfer fixing*: *m*)
apply(*simp add*: *less_eq_st_rep_iff eq_st_def m_s2_rep*)
done

lemma *m_o2*: *finite X* $\Longrightarrow$ *top_on_opt o1* ($-X$) $\Longrightarrow$ *top_on_opt o2* ($-X$)
$\Longrightarrow$
o1 < *o2* $\Longrightarrow$ *m_o o1 X* > *m_o o2 X*
proof(*induction o1 o2 rule*: *less_eq_option.induct*)
case *1* **thus** *?case* **by** (*auto simp*: *m_o_def m_s2 less_option_def*)
next
case *2* **thus** *?case* **by**(*auto simp*: *m_o_def less_option_def le_imp_less_Suc m_s_h*)
next
case *3* **thus** *?case* **by** (*auto simp*: *less_option_def*)
qed

lemma *m_o1*: *finite X* $\Longrightarrow$ *top_on_opt o1* ($-X$) $\Longrightarrow$ *top_on_opt o2* ($-X$)
$\Longrightarrow$
o1 $\le$ *o2* $\Longrightarrow$ *m_o o1 X* $\ge$ *m_o o2 X*
by(*auto simp*: *le_less m_o2*)

lemma *m_c2*: *top_on_acom C1* ($-$*vars C1*) $\Longrightarrow$ *top_on_acom C2* ($-$*vars C2*) $\Longrightarrow$
C1 < *C2* $\Longrightarrow$ *m_c C1* > *m_c C2*
proof(*auto simp add*: *le_iff_le_annos size_annos_same*[*of C1 C2*] *vars_acom_def less_acom_def*)
let *?X* = *vars*(*strip C2*)
assume *top*: *top_on_acom C1* ($-$ *vars*(*strip C2*)) *top_on_acom C2* ($-$ *vars*(*strip C2*))
and *strip_eq*: *strip C1* = *strip C2*
and *0*: $\forall$ *i*<*size*(*annos C2*). *annos C1* ! *i* $\le$ *annos C2* ! *i*
hence *1*: $\forall$ *i*<*size*(*annos C2*). *m_o* (*annos C1* ! *i*) *?X* $\ge$ *m_o* (*annos C2* ! *i*) *?X*
apply (*auto simp*: *all_set_conv_all_nth vars_acom_def top_on_acom_def*)
by (*metis finite_cvars m_o1 size_annos_same2*)
fix *i* **assume** *i*: *i* < *size*(*annos C2*) $\neg$ *annos C2* ! *i* $\le$ *annos C1* ! *i*

have *topo1*: *top_on_opt* (*annos C1* ! *i*) (− *?X*)
  using *i*(*1*) *top*(*1*) **by**(*simp add*: *top_on_acom_def size_annos_same*[*OF strip_eq*])
have *topo2*: *top_on_opt* (*annos C2* ! *i*) (− *?X*)
  using *i*(*1*) *top*(*2*) **by**(*simp add*: *top_on_acom_def size_annos_same*[*OF strip_eq*])
from *i* **have** *m_o* (*annos C1* ! *i*) *?X* > *m_o* (*annos C2* ! *i*) *?X* (**is** *?P i*)
  by (*metis 0 less_option_def m_o2*[*OF finite_cvars topo1*] *topo2*)
hence *2*: ∃ *i* < *size*(*annos C2*). *?P i* **using** ⟨*i* < *size*(*annos C2*)⟩ **by** *blast*
have ($\sum$ *i*<*size*(*annos C2*). *m_o* (*annos C2* ! *i*) *?X*)
    < ($\sum$ *i*<*size*(*annos C2*). *m_o* (*annos C1* ! *i*) *?X*)
  apply(*rule sum_strict_mono_ex1*) **using** *1 2* **by** (*auto*)
thus *?thesis*
  by(*simp add*: *m_c_def vars_acom_def strip_eq sum_list_sum_nth atLeast0LessThan size_annos_same*[*OF strip_eq*])
qed

end

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

lemma *top_on_step*′: ⟦ *top_on_acom C* (−*vars C*) ⟧ ⟹ *top_on_acom* (*step*′ ⊤ *C*) (−*vars C*)
unfolding *step*′*_def*
by(*rule top_on_Step*)
  (*auto simp add*: *top_option_def fun_top split*: *option.splits*)

lemma *AI_Some_measure*: ∃ *C*. *AI c* = *Some C*
unfolding *AI_def*
apply(*rule pfp_termination*[**where** *I* = λ*C*. *top_on_acom C* (− *vars C*) **and** *m*=*m_c*])
apply(*simp_all add*: *m_c2 mono_step*′*_top bot_least top_on_bot*)
using *top_on_step*′ **apply**(*auto simp add*: *vars_acom_def*)
done

end

end

theory *Abs_Int1_parity*
imports *Abs_Int1*
begin

14.11 Parity Analysis

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

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

instantiation *parity* :: *order*
begin

First the definition of the interface function $\leq$. Note that the header of the definition must refer to the ascii name ($\leq$) of the constants as *less_eq_parity* and the definition is named *less_eq_parity_def*. Inside the definition the symbolic names can be used.

definition *less_eq_parity* **where**
$x \leq y = (y = Either \lor x{=}y)$

We also need $<$, which is defined canonically:

definition *less_parity* **where**
$x < y = (x \leq y \land \neg\, y \leq (x{::}parity))$

(The type annotation is necessary to fix the type of the polymorphic predicates.)

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** $x \leq x$ **by**(*auto simp*: *less_eq_parity_def*)
next
  fix $x\ y\ z :: parity$ **assume** $x \leq y\ y \leq z$ **thus** $x \leq z$
    by(*auto simp*: *less_eq_parity_def*)
next
  fix $x\ y :: parity$ **assume** $x \leq y\ y \leq x$ **thus** $x = y$
    by(*auto simp*: *less_eq_parity_def*)
next
  fix $x\ y :: parity$ **show** $(x < y) = (x \leq y \land \neg\, y \leq x)$ **by**(*rule less_parity_def*)
qed

end

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

instantiation *parity* :: *semilattice_sup_top*

begin

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

definition *top_parity* **where**
$\top = \textit{Either}$

Now the instance proof. This time we take a shortcut with the help of proof method *goal_cases*: it creates cases 1 ... n for the subgoals 1 ... n; in case i, i is also the name of the assumptions of subgoal i and *case?* refers to the conclusion of subgoal i. The class axioms are presented in the same order as in the class definition.

instance
proof (*standard*, *goal_cases*)
  case *1* **show** *?case* **by**(*auto simp*: *less_eq_parity_def sup_parity_def*)
next
  case *2* **show** *?case* **by**(*auto simp*: *less_eq_parity_def sup_parity_def*)
next
  case *3* **thus** *?case* **by**(*auto simp*: *less_eq_parity_def sup_parity_def*)
next
  case *4* **show** *?case* **by**(*auto simp*: *less_eq_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* ⇒ *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* ⇒ *parity* **where**
num_parity i = (*if i mod 2 = 0 then Even else Odd*)

fun *plus_parity* :: *parity* ⇒ *parity* ⇒ *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:

global_interpretation *Val_semilattice*
where γ = γ*_parity* **and** *num′* = *num_parity* **and** *plus′* = *plus_parity*
proof (*standard*, *goal_cases*)

subgoals are the locale axioms

case *1* **thus** *?case* **by**(*auto simp*: *less_eq_parity_def*)
next
case *2* **show** *?case* **by**(*auto simp*: *top_parity_def*)
next
case *3* **show** *?case* **by** *auto*
next
case (*4* _ *a1* _ *a2*) **thus** *?case*
by (*induction a1 a2 rule*: *plus_parity.induct*)
(*auto simp add*: *mod_add_eq* [*symmetric*])
qed

In case 4 we needed to refer to particular variables. Writing (i x y z) fixes the names of the variables in case i to be x, y and z in the left-to-right order in which the variables occur in the subgoal. Underscores are anonymous placeholders for variable names we don't care to fix.

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*:

global_interpretation *Abs_Int*
where γ = γ*_parity* **and** *num′* = *num_parity* **and** *plus′* = *plus_parity*
defines *aval_parity* = *aval′* **and** *step_parity* = *step′* **and** *AI_parity* = *AI*
..

14.11.1 Tests

definition *test1_parity* =
″x″ ::= *N 1*;;
WHILE Less (*V ″x″*) (*N 100*) *DO ″x″* ::= *Plus* (*V ″x″*) (*N 2*)
value *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* $\top$) ^^ *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*))

14.11.2 Termination

global_interpretation *Abs_Int_mono*
where $\gamma = \gamma$*_parity* **and** *num'* = *num_parity* **and** *plus'* = *plus_parity*
proof (*standard*, *goal_cases*)
  case (*1 _ a1 _ a2*) **thus** *?case*
    by(*induction a1 a2 rule*: *plus_parity.induct*)
      (*auto simp add*:*less_eq_parity_def*)
qed

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

global_interpretation *Abs_Int_measure*
where $\gamma = \gamma$*_parity* **and** *num'* = *num_parity* **and** *plus'* = *plus_parity*
and *m* = *m_parity* **and** *h* = *1*
proof (*standard*, *goal_cases*)
  case *1* **thus** *?case* **by**(*auto simp add*: *m_parity_def less_eq_parity_def*)
next
  case *2* **thus** *?case* **by**(*auto simp add*: *m_parity_def less_eq_parity_def less_parity_def*)
qed

thm *AI_Some_measure*

end

theory *Abs_Int1_const*
imports *Abs_Int1*
begin

14.12 Constant Propagation

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

```
fun γ_const where
γ_const (Const i) = {i} |
γ_const (Any) = UNIV

fun plus_const where
plus_const (Const i) (Const j) = Const(i+j) |
plus_const _ _ = Any

lemma plus_const_cases: plus_const a1 a2 =
  (case (a1,a2) of (Const i, Const j) ⇒ Const(i+j) | _ ⇒ Any)
by(auto split: prod.split const.split)

instantiation const :: semilattice_sup_top
begin

fun less_eq_const where x ≤ y = (y = Any | x=y)

definition x < (y::const) = (x ≤ y & ¬ y ≤ x)

fun sup_const where x ⊔ y = (if x=y then x else Any)

definition ⊤ = Any

instance
proof (standard, goal_cases)
  case 1 thus ?case by (rule less_const_def)
next
  case (2 x) show ?case by (cases x) simp_all
next
  case (3 x y z) thus ?case by(cases z, cases y, cases x, simp_all)
next
  case (4 x y) thus ?case by(cases x, cases y, simp_all, cases y, simp_all)
next
  case (6 x y) thus ?case by(cases x, cases y, simp_all)
next
  case (5 x y) thus ?case by(cases y, cases x, simp_all)
next
  case (7 x y z) thus ?case by(cases z, cases y, cases x, simp_all)
next
  case 8 thus ?case by(simp add: top_const_def)
qed

end
```

global_interpretation *Val_semilattice*
where $\gamma = \gamma\_const$ **and** $num' = Const$ **and** $plus' = plus\_const$
proof (*standard*, *goal_cases*)
  case (*1 a b*) **thus** *?case*
    by(*cases a*, *cases b*, *simp*, *simp*, *cases b*, *simp*, *simp*)
next
  case *2* **show** *?case* **by**(*simp add*: *top_const_def*)
next
  case *3* **show** *?case* **by** *simp*
next
  case *4* **thus** *?case* **by**(*auto simp*: *plus_const_cases split*: *const.split*)
qed

global_interpretation *Abs_Int*
where $\gamma = \gamma\_const$ **and** $num' = Const$ **and** $plus' = plus\_const$
defines $AI\_const = AI$ **and** $step\_const = step'$ **and** $aval'\_const = aval'$
..

14.12.1 Tests

definition $steps\ c\ i = (step\_const\ \top\ \hat{}\hat{}\ i)\ (bot\ c)$

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:

global_interpretation *Abs_Int_mono*
where $\gamma = \gamma$*_const* **and** *num'* = *Const* **and** *plus'* = *plus_const*
proof (*standard*, *goal_cases*)
 case *1* **thus** *?case* **by**(*auto simp*: *plus_const_cases split*: *const.split*)
qed

Termination:

definition *m_const* :: *const* ⇒ *nat* **where**
m_const x = (*if x* = *Any then 0 else 1*)

global_interpretation *Abs_Int_measure*
where $\gamma = \gamma$*_const* **and** *num'* = *Const* **and** *plus'* = *plus_const*
and *m* = *m_const* **and** *h* = *1*
proof (*standard*, *goal_cases*)
 case *1* **thus** *?case* **by**(*auto simp*: *m_const_def split*: *const.splits*)
next
 case *2* **thus** *?case* **by**(*auto simp*: *m_const_def less_const_def split*: *const.splits*)
qed

thm *AI_Some_measure*

end

theory *Abs_Int2*
imports *Abs_Int1*
begin

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

definition *less_eq_prod p1 p2* = (*fst p1* ≤ *fst p2* ∧ *snd p1* ≤ *snd p2*)
definition *less_prod p1 p2* = (*p1* ≤ *p2* ∧ ¬ *p2* ≤ (*p1*::'*a*∗'*b*))

instance
proof (*standard*, *goal_cases*)
  case *1* **show** *?case* **by**(*rule less_prod_def*)
next
  case *2* **show** *?case* **by**(*simp add*: *less_eq_prod_def*)
next
  case *3* **thus** *?case* **unfolding** *less_eq_prod_def* **by**(*metis order_trans*)
next
  case *4* **thus** *?case* **by**(*simp add*: *less_eq_prod_def*)(*metis eq_iff surjective_pairing*)
qed

end

14.13 Backward Analysis of Expressions

subclass (**in** *bounded_lattice*) *semilattice_sup_top* **..**

locale *Val_lattice_gamma* = *Gamma_semilattice* **where** γ = γ
  for γ :: '*av*::*bounded_lattice* ⇒ *val set* +
assumes *inter_gamma_subset_gamma_inf*:
  γ *a1* ∩ γ *a2* ⊆ γ(*a1* ⊓ *a2*)
and *gamma_bot*[*simp*]: γ ⊥ = {}
begin

lemma *in_gamma_inf*: *x* ∈ γ *a1* ⟹ *x* ∈ γ *a2* ⟹ *x* ∈ γ(*a1* ⊓ *a2*)
by (*metis IntI inter_gamma_subset_gamma_inf set_mp*)

lemma *gamma_inf*: γ(*a1* ⊓ *a2*) = γ *a1* ∩ γ *a2*
by(*rule equalityI*[*OF* _ *inter_gamma_subset_gamma_inf*])
  (*metis inf_le1 inf_le2 le_inf_iff mono_gamma*)

end

locale *Val_inv* = *Val_lattice_gamma* **where** $\gamma = \gamma$
  for γ :: *'av::bounded_lattice* $\Rightarrow$ *val set* +
fixes *test_num'* :: *val* $\Rightarrow$ *'av* $\Rightarrow$ *bool*
and *inv_plus'* :: *'av* $\Rightarrow$ *'av* $\Rightarrow$ *'av* $\Rightarrow$ *'av* $*$ *'av*
and *inv_less'* :: *bool* $\Rightarrow$ *'av* $\Rightarrow$ *'av* $\Rightarrow$ *'av* $*$ *'av*
assumes *test_num'*: *test_num' i a* = ($i \in \gamma\ a$)
and *inv_plus'*: *inv_plus' a a1 a2* = (a_1',a_2') $\Longrightarrow$
  $i1 \in \gamma\ a1 \Longrightarrow i2 \in \gamma\ a2 \Longrightarrow i1{+}i2 \in \gamma\ a \Longrightarrow i1 \in \gamma\ a_1' \wedge i2 \in \gamma\ a_2'$
and *inv_less'*: *inv_less'* ($i1{<}i2$) *a1 a2* = (a_1',a_2') $\Longrightarrow$
  $i1 \in \gamma\ a1 \Longrightarrow i2 \in \gamma\ a2 \Longrightarrow i1 \in \gamma\ a_1' \wedge i2 \in \gamma\ a_2'$

locale *Abs_Int_inv* = *Val_inv* **where** $\gamma = \gamma$
  for γ :: *'av::bounded_lattice* $\Rightarrow$ *val set*
begin

lemma *in_gamma_sup_UpI*:
  $s \in \gamma_o\ S1 \vee s \in \gamma_o\ S2 \Longrightarrow s \in \gamma_o(S1 \sqcup S2)$
by (*metis* (*hide_lams*, *no_types*) *sup_ge1 sup_ge2 mono_gamma_o subsetD*)

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

lemma *aval''_correct*: $s \in \gamma_o\ S \Longrightarrow$ *aval a s* $\in \gamma$(*aval'' a S*)
by(*cases S*)(*auto simp add*: *aval'_correct split*: *option.splits*)

14.13.1 Backward analysis

fun *inv_aval'* :: *aexp* $\Rightarrow$ *'av* $\Rightarrow$ *'av st option* $\Rightarrow$ *'av st option* **where**
inv_aval' (*N n*) *a S* = (*if test_num' n a then S else None*) |
inv_aval' (*V x*) *a S* = (*case S of None* $\Rightarrow$ *None* | *Some S* $\Rightarrow$
  let a' = *fun S x* $\sqcap$ *a in*
  if a' = $\bot$ *then None else Some*(*update S x a'*)) |
inv_aval' (*Plus e1 e2*) *a S* =
 (*let* (*a1*,*a2*) = *inv_plus' a* (*aval'' e1 S*) (*aval'' e2 S*)
  in inv_aval' e1 a1 (*inv_aval' e2 a2 S*))

The test for *bot* in the V-case is important: *bot* indicates that a variable has no possible values, i.e. that the current program point is unreachable. But then the abstract state should collapse to *None*. Put differently, we maintain the invariant that in an abstract state of the form *Some s*, all variables are mapped to non-*bot* values. Otherwise the (pointwise) sup of two abstract states, one of which contains *bot* values, may produce too large a result, thus making the analysis less precise.

fun *inv_bval′* :: *bexp* ⇒ *bool* ⇒ *′av st option* ⇒ *′av st option* **where**
inv_bval′ (*Bc v*) *res S* = (*if v=res then S else None*) |
inv_bval′ (*Not b*) *res S* = *inv_bval′ b* (¬ *res*) *S* |
inv_bval′ (*And b1 b2*) *res S* =
  (*if res then inv_bval′ b1 True* (*inv_bval′ b2 True S*)
   else inv_bval′ b1 False S ⊔ *inv_bval′ b2 False S*) |
inv_bval′ (*Less e1 e2*) *res S* =
  (*let* (*a1*,*a2*) = *inv_less′ res* (*aval″ e1 S*) (*aval″ e2 S*)
   in inv_aval′ e1 a1 (*inv_aval′ e2 a2 S*))

lemma *inv_aval′_correct*: $s \in \gamma_o\ S \Longrightarrow aval\ e\ s \in \gamma\ a \Longrightarrow s \in \gamma_o$ (*inv_aval′ 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** $s \in \gamma_s\ S'$ **using** ⟨$s \in \gamma_o\ S$⟩
    by(*auto simp*: *in_gamma_option_iff*)
  moreover hence $s\ x \in \gamma$ (*fun S′ x*)
    by(*simp add*: *γ_st_def*)
  moreover have $s\ x \in \gamma\ a$ **using** *V*(*2*) **by** *simp*
  ultimately show *?case*
    by(*simp add*: *Let_def γ_st_def*)
      (*metis mono_gamma emptyE in_gamma_inf gamma_bot subset_empty*)
next
  case (*Plus e1 e2*) **thus** *?case*
    using *inv_plus′*[*OF _ aval″_correct aval″_correct*]
    by (*auto split*: *prod.split*)
qed

lemma *inv_bval′_correct*: $s \in \gamma_o\ S \Longrightarrow bv = bval\ b\ s \Longrightarrow s \in \gamma_o$(*inv_bval′ 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*) *in_gamma_sup_UpI*)
next

case (*Less e1 e2*) **thus** *?case*
apply *hypsubst_thin*
apply (*auto split*: *prod.split*)
apply (*metis* (*lifting*) *inv_aval'_correct aval''_correct inv_less'*)
done
qed

definition *step'* = *Step*
(λx *e S*. *case S of None* $\Rightarrow$ *None* | *Some S* $\Rightarrow$ *Some*(*update S x* (*aval' e S*)))
(λb *S*. *inv_bval' b True S*)

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(*simp add*: *step'_def*)

lemma *top_on_inv_aval'*: $[\![$ *top_on_opt S X*; *vars e* $\subseteq -X$ $]\!] \Longrightarrow$ *top_on_opt* (*inv_aval' e a S*) *X*
by(*induction e arbitrary*: *a S*) (*auto simp*: *Let_def split*: *option.splits prod.split*)

lemma *top_on_inv_bval'*: $[\![$*top_on_opt S X*; *vars b* $\subseteq -X$$]\!] \Longrightarrow$ *top_on_opt* (*inv_bval' b r S*) *X*
by(*induction b arbitrary*: *r S*) (*auto simp*: *top_on_inv_aval' top_on_sup split*: *prod.split*)

lemma *top_on_step'*: *top_on_acom C* (− *vars C*) $\Longrightarrow$ *top_on_acom* (*step'* $\top$ *C*) (− *vars C*)
unfolding *step'_def*
by(*rule top_on_Step*)
(*auto simp add*: *top_on_top top_on_inv_bval' split*: *option.split*)

14.13.2 Correctness

lemma *step_step'*: *step* (γ_o *S*) (γ_c *C*) $\le \gamma_c$ (*step' S C*)
unfolding *step_def step'_def*
by(*rule gamma_Step_subcomm*)
(*auto simp*: *intro*!: *aval'_correct inv_bval'_correct in_gamma_update split*: *option.splits*)

lemma *AI_correct*: *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* $\leq$ *C* **by**(*rule pfp_pfp*[*OF 1*])
have *2*: *step* (γ_o $\top$) (γ_c *C*) $\leq$ γ_c *C* — transfer the pfp'
proof(*rule order_trans*)
show *step* (γ_o $\top$) (γ_c *C*) $\leq$ γ_c (*step'* $\top$ *C*) **by**(*rule step_step'*)
show ... $\leq$ γ_c *C* **by** (*metis mono_gamma_c*[*OF pfp'*])
qed
have *3*: *strip* (γ_c *C*) = *c* **by**(*simp add*: *strip_pfp*[*OF _ 1*] *step'_def*)
have *lfp c* (*step* (γ_o $\top$)) $\leq$ γ_c *C*
by(*rule lfp_lowerbound*[*simplified*,**where** *f*=*step* (γ_o $\top$), *OF 3 2*])
thus *lfp c* (*step UNIV*) $\leq$ γ_c *C* **by** *simp*
qed

end

14.13.3 Monotonicity

locale *Abs_Int_inv_mono* = *Abs_Int_inv* +
assumes *mono_plus'*: *a1* $\leq$ *b1* $\Longrightarrow$ *a2* $\leq$ *b2* $\Longrightarrow$ *plus' a1 a2* $\leq$ *plus' b1 b2*
and *mono_inv_plus'*: *a1* $\leq$ *b1* $\Longrightarrow$ *a2* $\leq$ *b2* $\Longrightarrow$ *r* $\leq$ *r'* $\Longrightarrow$
inv_plus' r a1 a2 $\leq$ *inv_plus' r' b1 b2*
and *mono_inv_less'*: *a1* $\leq$ *b1* $\Longrightarrow$ *a2* $\leq$ *b2* $\Longrightarrow$
inv_less' bv a1 a2 $\leq$ *inv_less' bv b1 b2*
begin

lemma *mono_aval'*:
S1 $\leq$ *S2* $\Longrightarrow$ *aval' e S1* $\leq$ *aval' e S2*
by(*induction e*) (*auto simp*: *mono_plus' mono_fun*)

lemma *mono_aval''*:
S1 $\leq$ *S2* $\Longrightarrow$ *aval'' e S1* $\leq$ *aval'' e S2*
apply(*cases S1*)
apply *simp*
apply(*cases S2*)
apply *simp*
by (*simp add*: *mono_aval'*)

lemma *mono_inv_aval'*: *r1* $\leq$ *r2* $\Longrightarrow$ *S1* $\leq$ *S2* $\Longrightarrow$ *inv_aval' e r1 S1* $\leq$ *inv_aval' e r2 S2*
apply(*induction e arbitrary*: *r1 r2 S1 S2*)
apply(*auto simp*: *test_num' Let_def inf_mono split*: *option.splits prod.splits*)
apply (*metis mono_gamma subsetD*)
apply (*metis le_bot inf_mono le_st_iff*)
apply (*metis inf_mono mono_update le_st_iff*)
apply(*metis mono_aval'' mono_inv_plus'*[*simplified less_eq_prod_def*] *fst_conv*

snd_conv)
done

lemma *mono_inv_bval'*: $S1 \leq S2 \Longrightarrow$ *inv_bval'* *b bv* $S1 \leq$ *inv_bval'* *b bv* $S2$
apply(*induction b arbitrary*: *bv S1 S2*)
apply(*simp*)
apply(*simp*)
apply *simp*
apply(*metis order_trans*[*OF* _ *sup_ge1*] *order_trans*[*OF* _ *sup_ge2*])
apply (*simp split*: *prod.splits*)
apply(*metis mono_aval'' mono_inv_aval' mono_inv_less'*[*simplified less_eq_prod_def*] *fst_conv snd_conv*)
done

theorem *mono_step'*: $S1 \leq S2 \Longrightarrow C1 \leq C2 \Longrightarrow$ *step'* $S1\ C1 \leq$ *step'* $S2\ C2$
unfolding *step'_def*
by(*rule mono2_Step*) (*auto simp*: *mono_aval' mono_inv_bval' split*: *option.split*)

lemma *mono_step'_top*: $C1 \leq C2 \Longrightarrow$ *step'* $\top\ C1 \leq$ *step'* $\top\ C2$
by (*metis mono_step' order_refl*)

end

end

theory *Abs_Int2_ivl*
imports *Abs_Int2*
begin

14.14 Interval Analysis

type_synonym *eint* = *int extended*
type_synonym *eint2* = *eint* ∗ *eint*

definition γ*_rep* :: *eint2* ⇒ *int set* **where**
γ*_rep p* = (*let* (*l*,*h*) = *p in* $\{i.\ l \leq Fin\ i \wedge Fin\ i \leq h\}$)

definition *eq_ivl* :: *eint2* ⇒ *eint2* ⇒ *bool* **where**
eq_ivl p1 p2 = (γ*_rep p1* = γ*_rep p2*)

lemma *refl_eq_ivl*[*simp*]: *eq_ivl p p*
by(*auto simp*: *eq_ivl_def*)

quotient_type *ivl* = *eint2* / *eq_ivl*
by(*rule equivpI*)(*auto simp*: *reflp_def symp_def transp_def eq_ivl_def*)

abbreviation *ivl_abbr* :: *eint* ⇒ *eint* ⇒ *ivl* ([_, _]) **where**
[*l*,*h*] == *abs_ivl*(*l*,*h*)

lift_definition *γ_ivl* :: *ivl* ⇒ *int set* **is** *γ_rep*
by(*simp add*: *eq_ivl_def*)

lemma *γ_ivl_nice*: *γ_ivl*[*l*,*h*] = {*i*. *l* ≤ *Fin i* ∧ *Fin i* ≤ *h*}
by *transfer* (*simp add*: *γ_rep_def*)

lift_definition *num_ivl* :: *int* ⇒ *ivl* **is** λ*i*. (*Fin i*, *Fin i*) **.**

lift_definition *in_ivl* :: *int* ⇒ *ivl* ⇒ *bool*
 is λ*i* (*l*,*h*). *l* ≤ *Fin i* ∧ *Fin i* ≤ *h*
by(*auto simp*: *eq_ivl_def γ_rep_def*)

lemma *in_ivl_nice*: *in_ivl i* [*l*,*h*] = (*l* ≤ *Fin i* ∧ *Fin i* ≤ *h*)
by *transfer simp*

definition *is_empty_rep* :: *eint2* ⇒ *bool* **where**
is_empty_rep p = (*let* (*l*,*h*) = *p in l*>*h* | *l*=*Pinf* & *h*=*Pinf* | *l*=*Minf* & *h*=*Minf*)

lemma *γ_rep_cases*: *γ_rep p* = (*case p of* (*Fin i*,*Fin j*) => {*i*..*j*} | (*Fin i*,*Pinf*) => {*i*..} |
 (*Minf*,*Fin i*) ⇒ {..*i*} | (*Minf*,*Pinf*) ⇒ *UNIV* | _ ⇒ {})
by(*auto simp add*: *γ_rep_def split*: *prod.splits extended.splits*)

lift_definition *is_empty_ivl* :: *ivl* ⇒ *bool* **is** *is_empty_rep*
apply(*auto simp*: *eq_ivl_def γ_rep_cases is_empty_rep_def*)
apply(*auto simp*: *not_less less_eq_extended_case split*: *extended.splits*)
done

lemma *eq_ivl_iff*: *eq_ivl p1 p2* = (*is_empty_rep p1* & *is_empty_rep p2* | *p1* = *p2*)
by(*auto simp*: *eq_ivl_def is_empty_rep_def γ_rep_cases Icc_eq_Icc split*: *prod.splits extended.splits*)

definition *empty_rep* :: *eint2* **where** *empty_rep* = (*Pinf*,*Minf*)

lift_definition *empty_ivl* :: *ivl* **is** *empty_rep* **.**

lemma *is_empty_empty_rep*[*simp*]: *is_empty_rep empty_rep*
by(*auto simp add*: *is_empty_rep_def empty_rep_def*)

lemma *is_empty_rep_iff*: *is_empty_rep p* = (γ*_rep p* = {})
by(*auto simp add*: γ*_rep_cases is_empty_rep_def split*: *prod.splits extended.splits*)

declare *is_empty_rep_iff* [*THEN iffD1*, *simp*]

instantiation *ivl* :: *semilattice_sup_top*
begin

definition *le_rep* :: *eint2* ⇒ *eint2* ⇒ *bool* **where**
le_rep p1 p2 = (*let* (*l1*,*h1*) = *p1*; (*l2*,*h2*) = *p2 in*
  if is_empty_rep(*l1*,*h1*) *then True else*
  if is_empty_rep(*l2*,*h2*) *then False else l1* ≥ *l2* & *h1* ≤ *h2*)

lemma *le_iff_subset*: *le_rep p1 p2* ⟷ γ*_rep p1* ⊆ γ*_rep p2*
apply *rule*
apply(*auto simp*: *is_empty_rep_def le_rep_def* γ*_rep_def split*: *if_splits prod.splits*)[*1*]
apply(*auto simp*: *is_empty_rep_def* γ*_rep_cases le_rep_def*)
apply(*auto simp*: *not_less split*: *extended.splits*)
done

lift_definition *less_eq_ivl* :: *ivl* ⇒ *ivl* ⇒ *bool* **is** *le_rep*
by(*auto simp*: *eq_ivl_def le_iff_subset*)

definition *less_ivl* **where** *i1* < *i2* = (*i1* ≤ *i2* ∧ ¬ *i2* ≤ (*i1*::*ivl*))

lemma *le_ivl_iff_subset*: *iv1* ≤ *iv2* ⟷ γ*_ivl iv1* ⊆ γ*_ivl iv2*
by *transfer* (*rule le_iff_subset*)

definition *sup_rep* :: *eint2* ⇒ *eint2* ⇒ *eint2* **where**
sup_rep p1 p2 = (*if is_empty_rep p1 then p2 else if is_empty_rep p2 then p1*
  else let (*l1*,*h1*) = *p1*; (*l2*,*h2*) = *p2 in* (*min l1 l2*, *max h1 h2*))

lift_definition *sup_ivl* :: *ivl* ⇒ *ivl* ⇒ *ivl* **is** *sup_rep*
by(*auto simp*: *eq_ivl_iff sup_rep_def*)

lift_definition *top_ivl* :: *ivl* **is** (*Minf*,*Pinf*) **.**

lemma *is_empty_min_max*:
  ¬ *is_empty_rep* (*l1*,*h1*) ⟹ ¬ *is_empty_rep* (*l2*, *h2*) ⟹ ¬ *is_empty_rep*

(*min l1 l2*, *max h1 h2*)
by(*auto simp add*: *is_empty_rep_def max_def min_def split*: *if_splits*)

instance
proof (*standard*, *goal_cases*)
  case *1* **show** *?case* **by** (*rule less_ivl_def*)
next
  case *2* **show** *?case* **by** *transfer* (*simp add*: *le_rep_def split*: *prod.splits*)
next
  case *3* **thus** *?case* **by** *transfer* (*auto simp*: *le_rep_def split*: *if_splits*)
next
  case *4* **thus** *?case* **by** *transfer* (*auto simp*: *le_rep_def eq_ivl_iff split*: *if_splits*)
next
  case *5* **thus** *?case* **by** *transfer* (*auto simp add*: *le_rep_def sup_rep_def is_empty_min_max*)
next
  case *6* **thus** *?case* **by** *transfer* (*auto simp add*: *le_rep_def sup_rep_def is_empty_min_max*)
next
  case *7* **thus** *?case* **by** *transfer* (*auto simp add*: *le_rep_def sup_rep_def*)
next
  case *8* **show** *?case* **by** *transfer* (*simp add*: *le_rep_def is_empty_rep_def*)
qed

end

Implement (naive) executable equality:

instantiation *ivl* :: *equal*
begin

definition *equal_ivl* **where**
equal_ivl i1 (*i2*::*ivl*) = (*i1*≤*i2* ∧ *i2* ≤ *i1*)

instance
proof (*standard*, *goal_cases*)
  case *1* **show** *?case* **by**(*simp add*: *equal_ivl_def eq_iff*)
qed

end

lemma [*simp*]: **fixes** *x* :: *'a*::*linorder extended* **shows** (¬ *x* < *Pinf*) = (*x* = *Pinf*)
by(*simp add*: *not_less*)

lemma [*simp*]: **fixes** *x* :: *'a::linorder extended* **shows** (¬ *Minf* < *x*) = (*x* = *Minf*)
by(*simp add*: *not_less*)

instantiation *ivl* :: *bounded_lattice*
begin

definition *inf_rep* :: *eint2* ⇒ *eint2* ⇒ *eint2* **where**
inf_rep p1 p2 = (*let* (*l1*,*h1*) = *p1*; (*l2*,*h2*) = *p2 in* (*max l1 l2*, *min h1 h2*))

lemma *γ_inf_rep*: *γ_rep*(*inf_rep p1 p2*) = *γ_rep p1* ∩ *γ_rep p2*
by(*auto simp*:*inf_rep_def γ_rep_cases split*: *prod.splits extended.splits*)

lift_definition *inf_ivl* :: *ivl* ⇒ *ivl* ⇒ *ivl* **is** *inf_rep*
by(*auto simp*: *γ_inf_rep eq_ivl_def*)

lemma *γ_inf*: *γ_ivl* (*iv1* ⊓ *iv2*) = *γ_ivl iv1* ∩ *γ_ivl iv2*
by *transfer* (*rule γ_inf_rep*)

definition ⊥ = *empty_ivl*

instance
proof (*standard*, *goal_cases*)
  case *1* **thus** *?case* **by** (*simp add*: *γ_inf le_ivl_iff_subset*)
next
  case *2* **thus** *?case* **by** (*simp add*: *γ_inf le_ivl_iff_subset*)
next
  case *3* **thus** *?case* **by** (*simp add*: *γ_inf le_ivl_iff_subset*)
next
  case *4* **show** *?case*
    unfolding *bot_ivl_def* **by** *transfer* (*auto simp*: *le_iff_subset*)
qed

end

lemma *eq_ivl_empty*: *eq_ivl p empty_rep* = *is_empty_rep p*
by (*metis eq_ivl_iff is_empty_empty_rep*)

lemma *le_ivl_nice*: [*l1*,*h1*] ≤ [*l2*,*h2*] ⟷
  (*if* [*l1*,*h1*] = ⊥ *then True else*
   if [*l2*,*h2*] = ⊥ *then False else l1* ≥ *l2* & *h1* ≤ *h2*)
unfolding *bot_ivl_def* **by** *transfer* (*simp add*: *le_rep_def eq_ivl_empty*)

lemma *sup_ivl_nice*: $[l1,h1] \sqcup [l2,h2] =$
$(if\ [l1,h1] = \bot\ then\ [l2,h2]\ else$
$if\ [l2,h2] = \bot\ then\ [l1,h1]\ else\ [min\ l1\ l2,max\ h1\ h2])$
unfolding *bot_ivl_def* **by** *transfer* (*simp add*: *sup_rep_def eq_ivl_empty*)

lemma *inf_ivl_nice*: $[l1,h1] \sqcap [l2,h2] = [max\ l1\ l2,min\ h1\ h2]$
by *transfer* (*simp add*: *inf_rep_def*)

lemma *top_ivl_nice*: $\top = [-\infty,\infty]$
by (*simp add*: *top_ivl_def*)

instantiation *ivl* :: *plus*
begin

definition *plus_rep* :: *eint2* ⇒ *eint2* ⇒ *eint2* **where**
plus_rep p1 p2 =
(*if is_empty_rep p1* ∨ *is_empty_rep p2 then empty_rep else*
let (*l1*,*h1*) = *p1*; (*l2*,*h2*) = *p2 in* (*l1*+*l2*, *h1*+*h2*))

lift_definition *plus_ivl* :: *ivl* ⇒ *ivl* ⇒ *ivl* **is** *plus_rep*
by(*auto simp*: *plus_rep_def eq_ivl_iff*)

instance ..
end

lemma *plus_ivl_nice*: $[l1,h1] + [l2,h2] =$
$(if\ [l1,h1] = \bot \lor [l2,h2] = \bot\ then\ \bot\ else\ [l1+l2\ ,\ h1+h2])$
unfolding *bot_ivl_def* **by** *transfer* (*auto simp*: *plus_rep_def eq_ivl_empty*)

lemma *uminus_eq_Minf*[*simp*]: $-x = Minf \longleftrightarrow x = Pinf$
by(*cases x*) *auto*
lemma *uminus_eq_Pinf*[*simp*]: $-x = Pinf \longleftrightarrow x = Minf$
by(*cases x*) *auto*

lemma *uminus_le_Fin_iff*: $-\ x \le Fin(-y) \longleftrightarrow Fin\ y \le (x::'a::ordered\_ab\_group\_add\ extended)$
by(*cases x*) *auto*
lemma *Fin_uminus_le_iff*: $Fin(-y) \le -x \longleftrightarrow x \le ((Fin\ y)::'a::ordered\_ab\_group\_add\ extended)$
by(*cases x*) *auto*

instantiation *ivl* :: *uminus*
begin

definition *uminus_rep* :: *eint2* ⇒ *eint2* **where**
uminus_rep p = (*let* (*l*,*h*) = *p in* (−*h*, −*l*))

lemma *γ_uminus_rep*: *i* ∈ *γ_rep p* ⟹ −*i* ∈ *γ_rep*(*uminus_rep p*)
by(*auto simp*: *uminus_rep_def γ_rep_def image_def uminus_le_Fin_iff Fin_uminus_le_iff*
split: *prod.split*)

lift_definition *uminus_ivl* :: *ivl* ⇒ *ivl* **is** *uminus_rep*
by (*auto simp*: *uminus_rep_def eq_ivl_def γ_rep_cases*)
(*auto simp*: *Icc_eq_Icc split*: *extended.splits*)

instance ..
end

lemma *γ_uminus*: *i* ∈ *γ_ivl iv* ⟹ −*i* ∈ *γ_ivl*(− *iv*)
by *transfer* (*rule γ_uminus_rep*)

lemma *uminus_nice*: −[*l*,*h*] = [−*h*,−*l*]
by *transfer* (*simp add*: *uminus_rep_def*)

instantiation *ivl* :: *minus*
begin

definition *minus_ivl* :: *ivl* ⇒ *ivl* ⇒ *ivl* **where**
(*iv1*::*ivl*) − *iv2* = *iv1* + −*iv2*

instance ..
end

definition *inv_plus_ivl* :: *ivl* ⇒ *ivl* ⇒ *ivl* ⇒ *ivl*∗*ivl* **where**
inv_plus_ivl iv iv1 iv2 = (*iv1* ⊓ (*iv* − *iv2*), *iv2* ⊓ (*iv* − *iv1*))

definition *above_rep* :: *eint2* ⇒ *eint2* **where**
above_rep p = (*if is_empty_rep p then empty_rep else let* (*l*,*h*) = *p in* (*l*,∞))

definition *below_rep* :: *eint2* ⇒ *eint2* **where**
below_rep p = (*if is_empty_rep p then empty_rep else let* (*l*,*h*) = *p in* (−∞,*h*))

lift_definition *above* :: *ivl* ⇒ *ivl* **is** *above_rep*
by(*auto simp*: *above_rep_def eq_ivl_iff*)

lift_definition *below* :: *ivl* ⇒ *ivl* **is** *below_rep*

by(*auto simp*: *below_rep_def eq_ivl_iff*)

lemma γ*_aboveI*: $i \in \gamma\_ivl\ iv \Longrightarrow i \leq j \Longrightarrow j \in \gamma\_ivl(above\ iv)$
by *transfer*
  (*auto simp add*: *above_rep_def* γ*_rep_cases is_empty_rep_def*
    split: *extended.splits*)

lemma γ*_belowI*: $i \in \gamma\_ivl\ iv \Longrightarrow j \leq i \Longrightarrow j \in \gamma\_ivl(below\ iv)$
by *transfer*
  (*auto simp add*: *below_rep_def* γ*_rep_cases is_empty_rep_def*
    split: *extended.splits*)

definition *inv_less_ivl* :: $bool \Rightarrow ivl \Rightarrow ivl \Rightarrow ivl * ivl$ **where**
inv_less_ivl res iv1 iv2 =
 (*if res*
  then (*iv1* $\sqcap$ (*below iv2* $-$ [*1*,*1*]),
    iv2 $\sqcap$ (*above iv1* + [*1*,*1*]))
  else (*iv1* $\sqcap$ *above iv2*, *iv2* $\sqcap$ *below iv1*))

lemma *above_nice*: $above[l,h] = (if\ [l,h] = \bot\ then\ \bot\ else\ [l,\infty])$
unfolding *bot_ivl_def* **by** *transfer* (*simp add*: *above_rep_def eq_ivl_empty*)

lemma *below_nice*: $below[l,h] = (if\ [l,h] = \bot\ then\ \bot\ else\ [-\infty,h])$
unfolding *bot_ivl_def* **by** *transfer* (*simp add*: *below_rep_def eq_ivl_empty*)

lemma *add_mono_le_Fin*:
 $[\![x1 \leq Fin\ y1; x2 \leq Fin\ y2]\!] \Longrightarrow x1 + x2 \leq Fin\ (y1 + (y2::'a::ordered\_ab\_group\_add))$
by(*drule* (*1*) *add_mono*) *simp*

lemma *add_mono_Fin_le*:
 $[\![Fin\ y1 \leq x1; Fin\ y2 \leq x2]\!] \Longrightarrow Fin(y1 + y2::'a::ordered\_ab\_group\_add)$
$\leq x1 + x2$
by(*drule* (*1*) *add_mono*) *simp*

global_interpretation *Val_semilattice*
where $\gamma = \gamma\_ivl$ **and** $num' = num\_ivl$ **and** $plus' = (+)$
proof (*standard*, *goal_cases*)
 case *1* **thus** *?case* **by** *transfer* (*simp add*: *le_iff_subset*)
next
 case *2* **show** *?case* **by** *transfer* (*simp add*: γ*_rep_def*)
next
 case *3* **show** *?case* **by** *transfer* (*simp add*: γ*_rep_def*)
next
 case *4* **thus** *?case*

```
    apply transfer
   apply(auto simp: γ_rep_def plus_rep_def add_mono_le_Fin add_mono_Fin_le)
    by(auto simp: empty_rep_def is_empty_rep_def)
qed


global_interpretation Val_lattice_gamma
where γ = γ_ivl and num' = num_ivl and plus' = (+)
defines aval_ivl = aval'
proof (standard, goal_cases)
  case 1 show ?case by(simp add: γ_inf)
next
  case 2 show ?case unfolding bot_ivl_def by transfer simp
qed

global_interpretation Val_inv
where γ = γ_ivl and num' = num_ivl and plus' = (+)
and test_num' = in_ivl
and inv_plus' = inv_plus_ivl and inv_less' = inv_less_ivl
proof (standard, goal_cases)
  case 1 thus ?case by transfer (auto simp: γ_rep_def)
next
  case (2 _ _ _ _ _ i1 i2) thus ?case
    unfolding inv_plus_ivl_def minus_ivl_def
    apply(clarsimp simp add: γ_inf)
    using gamma_plus'[of i1+i2 _ -i1] gamma_plus'[of i1+i2 _ -i2]
    by(simp add:  γ_uminus)
next
  case (3 i1 i2) thus ?case
    unfolding inv_less_ivl_def minus_ivl_def one_extended_def
    apply(clarsimp simp add: γ_inf split: if_splits)
    using gamma_plus'[of i1+1 _ -1] gamma_plus'[of i2 - 1 _ 1]
    apply(simp add: γ_belowI[of i2] γ_aboveI[of i1]
      uminus_ivl.abs_eq uminus_rep_def  γ_ivl_nice)
    apply(simp add: γ_aboveI[of i2] γ_belowI[of i1])
    done
qed

global_interpretation Abs_Int_inv
where γ = γ_ivl and num' = num_ivl and plus' = (+)
and test_num' = in_ivl
and inv_plus' = inv_plus_ivl and inv_less' = inv_less_ivl
defines inv_aval_ivl = inv_aval'
and inv_bval_ivl = inv_bval'
```

```
and step_ivl = step'
and AI_ivl = AI
and aval_ivl' = aval''
..
```

Monotonicity:

```
lemma mono_plus_ivl: iv1 ≤ iv2 ⟹ iv3 ≤ iv4 ⟹ iv1+iv3 ≤ iv2+(iv4::ivl)
apply transfer
apply(auto simp: plus_rep_def le_iff_subset split: if_splits)
by(auto simp: is_empty_rep_iff γ_rep_cases split: extended.splits)

lemma mono_minus_ivl: iv1 ≤ iv2 ⟹ -iv1 ≤ -(iv2::ivl)
apply transfer
apply(auto simp: uminus_rep_def le_iff_subset split: if_splits prod.split)
by(auto simp: γ_rep_cases split: extended.splits)

lemma mono_above: iv1 ≤ iv2 ⟹ above iv1 ≤ above iv2
apply transfer
apply(auto simp: above_rep_def le_iff_subset split: if_splits prod.split)
by(auto simp: is_empty_rep_iff γ_rep_cases split: extended.splits)

lemma mono_below: iv1 ≤ iv2 ⟹ below iv1 ≤ below iv2
apply transfer
apply(auto simp: below_rep_def le_iff_subset split: if_splits prod.split)
by(auto simp: is_empty_rep_iff γ_rep_cases split: extended.splits)

global_interpretation Abs_Int_inv_mono
where γ = γ_ivl and num' = num_ivl and plus' = (+)
and test_num' = in_ivl
and inv_plus' = inv_plus_ivl and inv_less' = inv_less_ivl
proof (standard, goal_cases)
  case 1 thus ?case by (rule mono_plus_ivl)
next
  case 2 thus ?case
    unfolding inv_plus_ivl_def minus_ivl_def less_eq_prod_def
    by (auto simp: le_infI1 le_infI2 mono_plus_ivl mono_minus_ivl)
next
  case 3 thus ?case
    unfolding less_eq_prod_def inv_less_ivl_def minus_ivl_def
    by (auto simp: le_infI1 le_infI2 mono_plus_ivl mono_above mono_below)
qed
```

14.14.1 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* $\top$ ^^ *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

14.15 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* + *order* +
assumes *widen1*: $x \le x \nabla y$
assumes *widen2*: $y \le x \nabla y$
assumes *narrow1*: $y \le x \Longrightarrow y \le x \triangle y$
assumes *narrow2*: $y \le x \Longrightarrow x \triangle y \le x$
begin

lemma *narrowid*[*simp*]: $x \triangle x = x$
by (*metis eq_iff narrow1 narrow2*)

end

lemma *top_widen_top*[*simp*]: $\top \nabla \top = (\top::\_::\{wn,order\_top\})$
by (*metis eq_iff top_greatest widen2*)

instantiation *ivl* :: *wn*
begin

definition *widen_rep p1 p2* =
  (*if is_empty_rep p1 then p2 else if is_empty_rep p2 then p1 else*
   let (*l1*,*h1*) = *p1*; (*l2*,*h2*) = *p2*
   in (*if l2* < *l1 then Minf else l1*, *if h1* < *h2 then Pinf else h1*))

lift_definition *widen_ivl* :: *ivl* ⇒ *ivl* ⇒ *ivl* **is** *widen_rep*
by(*auto simp*: *widen_rep_def eq_ivl_iff*)

definition *narrow_rep p1 p2* =
  (*if is_empty_rep p1* ∨ *is_empty_rep p2 then empty_rep else*
   let (*l1*,*h1*) = *p1*; (*l2*,*h2*) = *p2*
   in (*if l1* = *Minf then l2 else l1*, *if h1* = *Pinf then h2 else h1*))

lift_definition *narrow_ivl* :: *ivl* ⇒ *ivl* ⇒ *ivl* **is** *narrow_rep*
by(*auto simp*: *narrow_rep_def eq_ivl_iff*)

instance
proof

qed (*transfer*, *auto simp*: *widen_rep_def narrow_rep_def le_iff_subset* γ*_rep_def subset_eq is_empty_rep_def empty_rep_def eq_ivl_def split*: *if_splits extended.splits*)+

end

instantiation *st* :: ({*order_top*,*wn*})*wn*
begin

lift_definition *widen_st* :: $'a$ *st* $\Rightarrow$ $'a$ *st* $\Rightarrow$ $'a$ *st* **is** *map2_st_rep* (∇)
by(*auto simp*: *eq_st_def*)

lift_definition *narrow_st* :: $'a$ *st* $\Rightarrow$ $'a$ *st* $\Rightarrow$ $'a$ *st* **is** *map2_st_rep* ($\triangle$)
by(*auto simp*: *eq_st_def*)

instance
proof (*standard*, *goal_cases*)
  case *1* **thus** *?case* **by** *transfer* (*simp add*: *less_eq_st_rep_iff widen1*)
next
  case *2* **thus** *?case* **by** *transfer* (*simp add*: *less_eq_st_rep_iff widen2*)
next
  case *3* **thus** *?case* **by** *transfer* (*simp add*: *less_eq_st_rep_iff narrow1*)
next
  case *4* **thus** *?case* **by** *transfer* (*simp add*: *less_eq_st_rep_iff narrow2*)
qed

end

instantiation *option* :: (*wn*)*wn*
begin

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

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

instance
proof (*standard*, *goal_cases*)
  case (*1* x y) **thus** *?case*

by(*induct x y rule*: *widen_option.induct*)(*simp_all add*: *widen1*)
next
case (*2 x y*) **thus** *?case*
by(*induct x y rule*: *widen_option.induct*)(*simp_all add*: *widen2*)
next
case (*3 x y*) **thus** *?case*
by(*induct x y rule*: *narrow_option.induct*) (*simp_all add*: *narrow1*)
next
case (*4 y x*) **thus** *?case*
by(*induct x y rule*: *narrow_option.induct*) (*simp_all add*: *narrow2*)
qed

end

definition *map2_acom* :: $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a\ acom \Rightarrow 'a\ acom \Rightarrow 'a\ acom$
where
map2_acom f C1 C2 = *annotate* ($\lambda p.$ *f* (*anno C1 p*) (*anno C2 p*)) (*strip C1*)

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

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

lemma *strip_map2_acom*[*simp*]:
strip C1 = *strip C2* $\Longrightarrow$ *strip*(*map2_acom f C1 C2*) = *strip C1*
by(*simp add*: *map2_acom_def*)

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 *narrow1_acom*: $C2 \leq C1 \Longrightarrow C2 \leq C1 \bigtriangleup (C2{::}'a{::}wn\ acom)$
by(*simp add*: *narrow_acom_def narrow1 map2_acom_def less_eq_acom_def size_annos*)

lemma *narrow2_acom*: $C2 \leq C1 \Longrightarrow C1 \bigtriangleup (C2{::}'a{::}wn\ acom) \leq C1$
by(*simp add*: *narrow_acom_def narrow2 map2_acom_def less_eq_acom_def size_annos*)

14.15.1 Pre-fixpoint computation

definition *iter_widen* :: $('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow ('a{::}\{order,widen\})option$
where *iter_widen f* = *while_option* $(\lambda x.\ \neg\ f\ x \leq x)\ (\lambda x.\ x \nabla f\ x)$

definition *iter_narrow* :: $('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow ('a{::}\{order,narrow\})option$
where *iter_narrow f* = *while_option* $(\lambda x.\ x \bigtriangleup f\ x < x)\ (\lambda x.\ x \bigtriangleup f\ x)$

definition *pfp_wn* :: $('a{::}\{order,widen,narrow\} \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ option$
where *pfp_wn f x* =
 (*case iter_widen f x of None* ⇒ *None* | *Some p* ⇒ *iter_narrow f p*)

lemma *iter_widen_pfp*: *iter_widen f x* = *Some p* $\Longrightarrow f\ p \leq p$
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 :: $'a\ acom \Rightarrow 'a\ acom$
assumes $\forall C.\ strip\ (f\ C) = strip\ C$ **and** *while_option P f C* = *Some* C'
shows *strip* C' = *strip C*
using *while_option_rule*[**where** $P = \lambda C'.\ strip\ C' = strip\ C$, *OF* _ *assms*(2)]
by (*metis assms*(1))

lemma *strip_iter_widen*: **fixes** f :: $'a{::}\{order,widen\}\ acom \Rightarrow 'a\ acom$
assumes $\forall C.\ strip\ (f\ C) = strip\ C$ **and** *iter_widen f C* = *Some* C'
shows *strip* C' = *strip C*
proof−
 have $\forall C.\ strip(C \nabla f\ C) = strip\ C$
  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 acom*. $P\ x1 \Longrightarrow P\ x2 \Longrightarrow x1 \le x2 \Longrightarrow f\ x1 \le f\ x2$
and *Pinv*: !!*x*. $P\ x \Longrightarrow P(f\ x)$!!*x1 x2*. $P\ x1 \Longrightarrow P\ x2 \Longrightarrow P(x1 \triangle x2)$
and $P\ p0$ **and** $f\ p0 \le p0$ **and** *iter_narrow f p0* = *Some p*
shows $P\ p \wedge f\ p \le p$
proof−
  let *?Q* = %*p*. $P\ p \wedge f\ p \le p \wedge p \le p0$
  have *?Q* $(p \triangle f\ p)$ **if** *Q*: *?Q p* **for** *p*
  proof *auto*
    note *P* = *conjunct1*[*OF Q*] **and** *12* = *conjunct2*[*OF Q*]
    note *1* = *conjunct1*[*OF 12*] **and** *2* = *conjunct2*[*OF 12*]
    let $?p' = p \triangle f\ p$
    show $P\ ?p'$ **by** (*blast intro*: *P Pinv*)
    have $f\ ?p' \le f\ p$ **by**(*rule mono*[*OF* ‹$P\ (p \triangle f\ p)$› *P narrow2_acom*[*OF 1*]])
    also have $\ldots \le\ ?p'$ **by**(*rule narrow1_acom*[*OF 1*])
    finally show $f\ ?p' \le\ ?p'$ **.**
    have $?p' \le p$ **by** (*rule narrow2_acom*[*OF 1*])
    also have $p \le p0$ **by**(*rule 2*)
    finally show $?p' \le p0$ **.**
  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 acom*. $P\ x1 \Longrightarrow P\ x2 \Longrightarrow x1 \le x2 \Longrightarrow f\ x1 \le f\ x2$
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** $P\ p \wedge f\ p \le p$
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*:
$\llbracket \forall C.\ strip(f\ C) = strip\ C;\ pfp\_wn\ f\ C = Some\ C' \rrbracket \Longrightarrow strip\ C' = strip\ C$
by(*auto simp add*: *pfp_wn_def iter_narrow_def split*: *option.splits*)
(*metis* (*mono_tags*) *strip_iter_widen strip_narrow_acom strip_while*)

locale *Abs_Int_wn* = *Abs_Int_inv_mono* **where** γ=γ
for γ :: *'av*::{*wn*,*bounded_lattice*} $\Rightarrow$ *val set*
begin

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

lemma *AI_wn_correct*: *AI_wn c = Some C* $\Longrightarrow$ *CS c* $\le \gamma_c$ *C*
proof(*simp add*: *CS_def AI_wn_def*)
assume *1*: *pfp_wn* (*step'* $\top$) (*bot c*) = *Some C*
have *2*: *strip C = c* $\wedge$ *step'* $\top$ *C* $\le$ *C*
by(*rule pfp_wn_pfp*[**where** *x=bot c*]) (*simp_all add*: *1 mono_step'_top*)
have *pfp*: *step* (γ_o $\top$) (γ_c *C*) $\le \gamma_c$ *C*
proof(*rule order_trans*)
show *step* (γ_o $\top$) (γ_c *C*) $\le$ γ_c (*step'* $\top$ *C*)
by(*rule step_step'*)
show ... $\le \gamma_c$ *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 $\top$)) $\le \gamma_c$ *C*
by(*rule lfp_lowerbound*[*simplified*,**where** *f=step* (γ_o $\top$), *OF 3 pfp*])
thus *lfp c* (*step UNIV*) $\le \gamma_c$ *C* **by** *simp*
qed

end

global_interpretation *Abs_Int_wn*
where γ = γ*_ivl* **and** *num'* = *num_ivl* **and** *plus'* = (+)
and *test_num'* = *in_ivl*
and *inv_plus'* = *inv_plus_ivl* **and** *inv_less'* = *inv_less_ivl*
defines *AI_wn_ivl* = *AI_wn*
..

14.15.2 Tests

definition *step_up_ivl n* = $((\lambda C.\ C \nabla \textit{step\_ivl} \top C)\hat{}\hat{}n)$
definition *step_down_ivl n* = $((\lambda C.\ C \triangle \textit{step\_ivl} \top C)\hat{}\hat{}n)$

For *test3_ivl*, *AI_ivl* needed as many iterations as the loop took to execute. In contrast, *AI_wn_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_wn_ivl test3_ivl*)

Now all the analyses terminate:

value *show_acom_opt* (*AI_wn_ivl test4_ivl*)
value *show_acom_opt* (*AI_wn_ivl test5_ivl*)
value *show_acom_opt* (*AI_wn_ivl test6_ivl*)

14.15.3 Generic Termination Proof

lemma *top_on_opt_widen*:
top_on_opt o1 X $\Longrightarrow$ *top_on_opt o2 X* $\Longrightarrow$ *top_on_opt* (*o1* ∇ *o2* :: _ *st option*) *X*
apply(*induct o1 o2 rule*: *widen_option.induct*)
apply (*auto*)
by *transfer simp*

lemma *top_on_opt_narrow*:
top_on_opt o1 X $\Longrightarrow$ *top_on_opt o2 X* $\Longrightarrow$ *top_on_opt* (*o1* $\triangle$ *o2* :: _ *st option*) *X*
apply(*induct o1 o2 rule*: *narrow_option.induct*)
apply (*auto*)
by *transfer simp*

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(*simp add*: *map2_acom_def list_eq_iff_nth_eq size_annos anno_def*[*symmetric*] *size_annos_same*[*of C1 C2*])

lemma *top_on_acom_widen*:
$[\![top\_on\_acom\ C1\ X;\ strip\ C1 = strip\ C2;\ top\_on\_acom\ C2\ X]\!]$
$\Longrightarrow top\_on\_acom\ (C1\ \nabla\ C2 :: \_\ st\ option\ acom)\ X$
by(*auto simp add*: *widen_acom_def top_on_acom_def*)(*metis top_on_opt_widen in_set_zipE*)

lemma *top_on_acom_narrow*:
$[\![top\_on\_acom\ C1\ X;\ strip\ C1 = strip\ C2;\ top\_on\_acom\ C2\ X]\!]$
$\Longrightarrow top\_on\_acom\ (C1\ \triangle\ C2 :: \_\ st\ option\ acom)\ X$
by(*auto simp add*: *narrow_acom_def top_on_acom_def*)(*metis top_on_opt_narrow in_set_zipE*)

The assumptions for widening and narrowing differ because during narrowing we have the invariant $y \leq x$ (where y is the next iterate), but during widening there is no such invariant, there we only have that not yet $y \leq x$. This complicates the termination proof for widening.

locale $Measure\_wn = Measure1$ **where** $m{=}m$
for $m :: {'av}{::}\{order\_top,wn\} \Rightarrow nat\ +$
fixes $n :: {'av} \Rightarrow nat$
assumes m_anti_mono: $x \leq y \Longrightarrow m\ x \geq m\ y$
assumes m_widen: $\sim\ y \leq x \Longrightarrow m(x\ \nabla\ y) < m\ x$
assumes n_narrow: $y \leq x \Longrightarrow x\ \triangle\ y < x \Longrightarrow n(x\ \triangle\ y) < n\ x$

begin

lemma *m_s_anti_mono_rep*: **assumes** $\forall x.\ S1\ x \leq S2\ x$
shows $(\sum x{\in}X.\ m\ (S2\ x)) \leq (\sum x{\in}X.\ m\ (S1\ x))$
proof−
from *assms* **have** $\forall x.\ m(S1\ x) \geq m(S2\ x)$ **by** (*metis m_anti_mono*)
thus $(\sum x{\in}X.\ m\ (S2\ x)) \leq (\sum x{\in}X.\ m\ (S1\ x))$ **by** (*metis sum_mono*)
qed

lemma *m_s_anti_mono*: $S1 \leq S2 \Longrightarrow m\_s\ S1\ X \geq m\_s\ S2\ X$
unfolding *m_s_def*
apply (*transfer fixing*: *m*)
apply(*simp add*: *less_eq_st_rep_iff eq_st_def m_s_anti_mono_rep*)
done

lemma *m_s_widen_rep*: **assumes** $finite\ X\ S1 = S2\ on\ {-X}\ \neg\ S2\ x \leq S1\ x$

```
  shows (∑x∈X. m (S1 x ∇ S2 x)) < (∑x∈X. m (S1 x))
proof−
  have 1: ∀x∈X. m(S1 x) ≥ m(S1 x ∇ S2 x)
    by (metis m_anti_mono wn_class.widen1)
  have x ∈ X using assms(2,3)
    by(auto simp add: Ball_def)
  hence 2: ∃x∈X. m(S1 x) > m(S1 x ∇ S2 x)
    using assms(3) m_widen by blast
  from sum_strict_mono_ex1[OF ‹finite X› 1 2]
  show ?thesis .
qed

lemma m_s_widen: finite X ⟹ fun S1 = fun S2 on −X ==>
  ~ S2 ≤ S1 ⟹ m_s (S1 ∇ S2) X < m_s S1 X
apply(auto simp add: less_st_def m_s_def)
apply (transfer fixing: m)
apply(auto simp add: less_eq_st_rep_iff m_s_widen_rep)
done

lemma m_o_anti_mono: finite X ⟹ top_on_opt o1 (−X) ⟹ top_on_opt
o2 (−X) ⟹
  o1 ≤ o2 ⟹ m_o o1 X ≥ m_o o2 X
proof(induction o1 o2 rule: less_eq_option.induct)
  case 1 thus ?case by (simp add: m_o_def)(metis m_s_anti_mono)
next
  case 2 thus ?case
    by(simp add: m_o_def le_SucI m_s_h split: option.splits)
next
  case 3 thus ?case by simp
qed

lemma m_o_widen: ⟦ finite X; top_on_opt S1 (−X); top_on_opt S2 (−X);
¬ S2 ≤ S1 ⟧ ⟹
  m_o (S1 ∇ S2) X < m_o S1 X
by(auto simp: m_o_def m_s_h less_Suc_eq_le m_s_widen split: option.split)

lemma m_c_widen:
  strip C1 = strip C2 ⟹ top_on_acom C1 (−vars C1) ⟹ top_on_acom
C2 (−vars C2)
   ⟹ ¬ C2 ≤ C1 ⟹ m_c (C1 ∇ C2) < m_c C1
apply(auto simp: m_c_def widen_acom_def map2_acom_def size_annos[symmetric]
anno_def[symmetric]sum_list_sum_nth)
apply(subgoal_tac length(annos C2) = length(annos C1))
 prefer 2 apply (simp add: size_annos_same2)
```

apply (*auto*)
apply(*rule sum_strict_mono_ex1*)
apply(*auto simp add*: *m_o_anti_mono vars_acom_def anno_def top_on_acom_def top_on_opt_widen widen1 less_eq_acom_def listrel_iff_nth*)
apply(*rule_tac x=p* **in** *bexI*)
apply (*auto simp*: *vars_acom_def m_o_widen top_on_acom_def*)
done

definition *n_s* :: *'av st* $\Rightarrow$ *vname set* $\Rightarrow$ *nat* (n_s) **where**
n_s *S X* = ($\sum x{\in}X.$ *n*(*fun S x*))

lemma *n_s_narrow_rep*:
assumes *finite X* *S1* = *S2 on* $-X$ $\forall x.$ *S2 x* $\leq$ *S1 x* $\forall x.$ *S1 x* $\triangle$ *S2 x* $\leq$ *S1 x*
S1 x $\neq$ *S1 x* $\triangle$ *S2 x*
shows ($\sum x{\in}X.$ *n* (*S1 x* $\triangle$ *S2 x*)) < ($\sum x{\in}X.$ *n* (*S1 x*))
proof−
have *1*: $\forall x.$ *n*(*S1 x* $\triangle$ *S2 x*) $\leq$ *n*(*S1 x*)
by (*metis assms*(*3*) *assms*(*4*) *eq_iff less_le_not_le n_narrow*)
have $x \in X$ **by** (*metis Compl_iff assms*(*2*) *assms*(*5*) *narrowid*)
hence *2*: $\exists x{\in}X.$ *n*(*S1 x* $\triangle$ *S2 x*) < *n*(*S1 x*)
by (*metis assms*(*3*−*5*) *eq_iff less_le_not_le n_narrow*)
show *?thesis*
apply(*rule sum_strict_mono_ex1*[*OF* ⟨*finite X*⟩]) **using** *1 2* **by** *blast+*
qed

lemma *n_s_narrow*: *finite X* $\Longrightarrow$ *fun S1* = *fun S2 on* $-X$ $\Longrightarrow$ *S2* $\leq$ *S1* $\Longrightarrow$ *S1* $\triangle$ *S2* < *S1*
$\Longrightarrow$ n_s (*S1* $\triangle$ *S2*) *X* < n_s *S1 X*
apply(*auto simp add*: *less_st_def n_s_def*)
apply (*transfer fixing*: *n*)
apply(*auto simp add*: *less_eq_st_rep_iff eq_st_def fun_eq_iff n_s_narrow_rep*)
done

definition *n_o* :: *'av st option* $\Rightarrow$ *vname set* $\Rightarrow$ *nat* (n_o) **where**
n_o *opt X* = (*case opt of None* $\Rightarrow$ *0* | *Some S* $\Rightarrow$ n_s *S X* + *1*)

lemma *n_o_narrow*:
top_on_opt S1 ($-X$) $\Longrightarrow$ *top_on_opt S2* ($-X$) $\Longrightarrow$ *finite X*
$\Longrightarrow$ *S2* $\leq$ *S1* $\Longrightarrow$ *S1* $\triangle$ *S2* < *S1* $\Longrightarrow$ n_o (*S1* $\triangle$ *S2*) *X* < n_o *S1 X*
apply(*induction S1 S2 rule*: *narrow_option.induct*)
apply(*auto simp*: *n_o_def n_s_narrow*)
done

definition *n_c* :: *'av st option acom* ⇒ *nat* (n_c) **where**
n_c *C* = *sum_list* (*map* (λ*a*. n_o *a* (*vars C*)) (*annos C*))

lemma *less_annos_iff*: (*C1* < *C2*) = (*C1* ≤ *C2* ∧
(∃ *i*<*length* (*annos C1*). *annos C1* ! *i* < *annos C2* ! *i*))
by(*metis* (*hide_lams*, *no_types*) *less_le_not_le le_iff_le_annos size_annos_same2*)

lemma *n_c_narrow*: *strip C1* = *strip C2*
⟹ *top_on_acom C1* (− *vars C1*) ⟹ *top_on_acom C2* (− *vars C2*)
⟹ *C2* ≤ *C1* ⟹ *C1* △ *C2* < *C1* ⟹ n_c (*C1* △ *C2*) < n_c *C1*
apply(*auto simp*: *n_c_def narrow_acom_def sum_list_sum_nth*)
apply(*subgoal_tac length*(*annos C2*) = *length*(*annos C1*))
prefer *2* **apply** (*simp add*: *size_annos_same2*)
apply (*auto*)
apply(*simp add*: *less_annos_iff le_iff_le_annos*)
apply(*rule sum_strict_mono_ex1*)
apply (*auto simp*: *vars_acom_def top_on_acom_def*)
apply (*metis n_o_narrow nth_mem finite_cvars less_imp_le le_less order_refl*)
apply(*rule_tac x*=*i* **in** *bexI*)
prefer *2* **apply** *simp*
apply(*rule n_o_narrow*[**where** *X* = *vars*(*strip C2*)])
apply (*simp_all*)
done

end

lemma *iter_widen_termination*:
fixes *m* :: *'a*::*wn acom* ⇒ *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** ∃ *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* ¬ *f C* ≤ *C* **thus** *P* (*C* ∇ *f C*) ∧ *m* (*C* ∇ *f C*) < *m C*
by(*simp add*: *P_f P_widen m_widen*)
qed

lemma *iter_narrow_termination*:
fixes n :: $'a$::*wn acom* $\Rightarrow$ *nat*
assumes *P_f*: $\bigwedge C.\ P\ C \Longrightarrow P(f\ C)$
and *P_narrow*: $\bigwedge C1\ C2.\ P\ C1 \Longrightarrow P\ C2 \Longrightarrow P(C1 \bigtriangleup C2)$
and *mono*: $\bigwedge C1\ C2.\ P\ C1 \Longrightarrow P\ C2 \Longrightarrow C1 \le C2 \Longrightarrow f\ C1 \le f\ C2$
and *n_narrow*: $\bigwedge C1\ C2.\ P\ C1 \Longrightarrow P\ C2 \Longrightarrow C2 \le C1 \Longrightarrow C1 \bigtriangleup C2 < C1 \Longrightarrow n(C1 \bigtriangleup C2) < n\ C1$
and *init*: $P\ C\ f\ C \le C$ **shows** $\exists\, C'.$ *iter_narrow* $f\ C$ = *Some* C'
proof(*simp add*: *iter_narrow_def*,
    rule measure_while_option_Some[**where** f=n **and** P = %$C.\ P\ C \wedge f\ C \le C$])
  show $P\ C \wedge f\ C \le C$ **using** *init* **by** *blast*
next
  fix C **assume** *1*: $P\ C \wedge f\ C \le C$ **and** *2*: $C \bigtriangleup f\ C < C$
  hence $P\ (C \bigtriangleup f\ C)$ **by**(*simp add*: *P_f P_narrow*)
  moreover then have $f\ (C \bigtriangleup f\ C) \le C \bigtriangleup f\ C$
    by (*metis narrow1_acom narrow2_acom 1 mono order_trans*)
  moreover have $n\ (C \bigtriangleup f\ C) < n\ C$ **using** *1 2* **by**(*simp add*: *n_narrow P_f*)
  ultimately show $(P\ (C \bigtriangleup f\ C) \wedge f\ (C \bigtriangleup f\ C) \le C \bigtriangleup f\ C) \wedge n(C \bigtriangleup f\ C) < n\ C$
    by *blast*
qed

locale *Abs_Int_wn_measure* = *Abs_Int_wn* **where** γ=γ + *Measure_wn* **where** m=m
  for γ :: $'av$::{*wn*,*bounded_lattice*} $\Rightarrow$ *val set* **and** m :: $'av \Rightarrow$ *nat*

14.15.4 Termination: Intervals

definition *m_rep* :: *eint2* $\Rightarrow$ *nat* **where**
m_rep p = (*if is_empty_rep p then 3 else*
  let (*l*,*h*) = *p in* (*case l of Minf* $\Rightarrow$ *0* | _ $\Rightarrow$ *1*) + (*case h of Pinf* $\Rightarrow$ *0* | _ $\Rightarrow$ *1*))

lift_definition *m_ivl* :: *ivl* $\Rightarrow$ *nat* **is** *m_rep*
by(*auto simp*: *m_rep_def eq_ivl_iff*)

lemma *m_ivl_nice*: *m_ivl*[*l*,*h*] = (*if* [*l*,*h*] = $\bot$ *then 3 else*
   (*if l* = *Minf then 0 else 1*) + (*if h* = *Pinf then 0 else 1*))
unfolding *bot_ivl_def*
by *transfer* (*auto simp*: *m_rep_def eq_ivl_empty split*: *extended.split*)

lemma *m_ivl_height*: *m_ivl iv* $\leq$ *3*
by *transfer* (*simp add*: *m_rep_def split*: *prod.split extended.split*)

lemma *m_ivl_anti_mono*: $y \leq x \Longrightarrow$ *m_ivl x* $\leq$ *m_ivl y*
by *transfer*
  (*auto simp*: *m_rep_def is_empty_rep_def* γ*_rep_cases le_iff_subset*
      split: *prod.split extended.splits if_splits*)

lemma *m_ivl_widen*:
  $\sim y \leq x \Longrightarrow$ *m_ivl*$(x \nabla y) <$ *m_ivl x*
by *transfer*
  (*auto simp*: *m_rep_def widen_rep_def is_empty_rep_def* γ*_rep_cases le_iff_subset*
      split: *prod.split extended.splits if_splits*)

definition *n_ivl* :: *ivl* $\Rightarrow$ *nat* **where**
n_ivl iv = *3* $-$ *m_ivl iv*

lemma *n_ivl_narrow*:
  $x \bigtriangleup y < x \Longrightarrow$ *n_ivl*$(x \bigtriangleup y) <$ *n_ivl x*
unfolding *n_ivl_def*
apply(*subst* (*asm*) *less_le_not_le*)
apply *transfer*
by(*auto simp add*: *m_rep_def narrow_rep_def is_empty_rep_def empty_rep_def*
γ*_rep_cases le_iff_subset*
      split: *prod.splits if_splits extended.split*)

global_interpretation *Abs_Int_wn_measure*
where $\gamma = \gamma$*_ivl* **and** *num′* = *num_ivl* **and** *plus′* = (+)
and *test_num′* = *in_ivl*
and *inv_plus′* = *inv_plus_ivl* **and** *inv_less′* = *inv_less_ivl*
and *m* = *m_ivl* **and** *n* = *n_ivl* **and** *h* = *3*
proof (*standard*, *goal_cases*)
  case *2* **thus** *?case* **by**(*rule m_ivl_anti_mono*)
next
  case *1* **thus** *?case* **by**(*rule m_ivl_height*)
next
  case *3* **thus** *?case* **by**(*rule m_ivl_widen*)
next
  case *4* **from** *4*(*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* $\top$) (*bot c*) = *Some C*
apply(*rule iter_widen_termination*[**where** m = *m_c* **and** $P = \%C.$ *strip C* = *c* $\wedge$ *top_on_acom C* ($-$ *vars C*)])
apply (*auto simp add*: *m_c_widen top_on_bot top_on_step'*[*simplified comp_def vars_acom_def*]
vars_acom_def top_on_acom_widen)
done

lemma *iter_narrow_step_ivl_termination*:
top_on_acom C ($-$ *vars C*) $\Longrightarrow$ *step_ivl* $\top$ $C \leq C \Longrightarrow$
$\exists C'.$ *iter_narrow* (*step_ivl* $\top$) C = *Some* C'
apply(*rule iter_narrow_termination*[**where** n = *n_c* **and** $P = \%C'.$ *strip* C = *strip* C' $\wedge$ *top_on_acom* C' ($-$*vars* C')])
apply(*auto simp*: *top_on_step'*[*simplified comp_def vars_acom_def*]
mono_step'_top n_c_narrow vars_acom_def top_on_acom_narrow)
done

theorem *AI_wn_ivl_termination*:
$\exists C.$ *AI_wn_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(*rule conjunct2*)
apply(*rule iter_widen_inv*[**where** f = *step'* $\top$ **and** $P = \%C.$ c = *strip C* & *top_on_acom C* ($-$ *vars C*)])
apply(*auto simp*: *top_on_acom_widen top_on_step'*[*simplified comp_def vars_acom_def*]
iter_widen_pfp top_on_bot vars_acom_def)
done

14.15.5 Counterexamples

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

lemma assumes !!x y::$'a$::*wn*. $x \leq y \Longrightarrow f\,x \leq f\,y$
and $x \leq f\,x$ **and** $\neg\, f\,x \leq x$ **shows** $x \nabla f\,x \leq f(x \nabla f\,x)$
nitpick[*card* = *3*, *expect* = *genuine*, *show_consts*, *timeout* = *120*]

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*. $x \leq y \Longrightarrow f\,x \leq f\,y$
and $x \leq f\,x$ **and** $\neg\, f\,x \leq x$ **and** $f(f\,x) \leq f\,x$

shows $f(x \nabla f\ x) \leq x \nabla f\ x$
nitpick[*card* = *4*, *expect* = *genuine*, *show_consts*, *timeout* = *120*]

oops

end

15 Extensions and Variations of IMP

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

15.1 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

15.1.1 Dynamic Scoping of Procedures and Variables

type_synonym *penv* = *pname* $\Rightarrow$ *com*

inductive
  big_step :: *penv* $\Rightarrow$ *com* $\times$ *state* $\Rightarrow$ *state* $\Rightarrow$ *bool* (_ $\vdash$ _ $\Rightarrow$ _ [*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

15.1.2 Static Scoping of Procedures, Dynamic of Variables

type_synonym $penv = (pname \times com)\ list$

inductive
$big\_step :: penv \Rightarrow com \times state \Rightarrow state \Rightarrow bool\ (\_ \vdash \_ \Rightarrow \_\ [60,0,60]\ 55)$
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: $\llbracket\ \neg bval\ b\ s;\ \ pe \vdash (c_2,s) \Rightarrow t\ \rrbracket \Longrightarrow$
$pe \vdash (IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \Rightarrow t$ |

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

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: $\llbracket\ p' \neq p;\ \ pe \vdash (CALL\ p,\ s) \Rightarrow t\ \rrbracket \Longrightarrow$
$(p',c)\#pe \vdash (CALL\ p,\ s) \Rightarrow t$ |

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

15.1.3 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: $\llbracket\ e \vdash (c_1,s_1) \Rightarrow s_2;\ \ e \vdash (c_2,s_2) \Rightarrow s_3\ \rrbracket \Longrightarrow$
$e \vdash (c_1;;c_2,\ s_1) \Rightarrow s_3$ |

IfTrue: $\llbracket\ bval\ b\ (s \circ venv\ e);\ \ e \vdash (c_1,s) \Rightarrow t\ \rrbracket \Longrightarrow$

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

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

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

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

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

code_pred *big_step* **.**

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

end
theory *C_like* **imports** *Main* **begin**

15.2 A C-like Language

type_synonym *state* $= nat \Rightarrow nat$

datatype *aexp* $=$ *N nat* | *Deref aexp* (!) | *Plus aexp aexp*

fun *aval* :: $aexp \Rightarrow state \Rightarrow nat$ **where**
$aval\ (N\ n)\ s = n\ |$
$aval\ (!a)\ s = s(aval\ a\ s)\ |$
$aval\ (Plus\ a_1\ a_2)\ s = aval\ a_1\ s + aval\ a_2\ s$

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

primrec *bval* :: *bexp* ⇒ *state* ⇒ *bool* **where**
bval (*Bc v*) _ = *v* |
bval (*Not b*) *s* = (¬ *bval b s*) |
bval (*And* b_1 b_2) *s* = (*if bval* b_1 *s then bval* b_2 *s else False*) |
bval (*Less* a_1 a_2) *s* = (*aval* a_1 *s* < *aval* a_2 *s*)

datatype
com = *SKIP*
| *Assign aexp aexp* (_ ::= _ [*61*, *61*] *61*)
| *New aexp aexp*
| *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*)

inductive
big_step :: *com* × *state* × *nat* ⇒ *state* × *nat* ⇒ *bool* (**infix** ⇒ *55*)
where
Skip: (*SKIP*,*sn*) ⇒ *sn* |
Assign: (*lhs* ::= *a*,*s*,*n*) ⇒ (*s*(*aval lhs s* := *aval a s*),*n*) |
New: (*New lhs a*,*s*,*n*) ⇒ (*s*(*aval lhs s* := *n*), *n*+*aval a s*) |
Seq: ⟦ (c_1,sn_1) ⇒ sn_2; (c_2,sn_2) ⇒ sn_3 ⟧ ⟹
(c_1;c_2, sn_1) ⇒ sn_3 |

IfTrue: ⟦ *bval b s*; (c_1,*s*,*n*) ⇒ *tn* ⟧ ⟹
(*IF b THEN* c_1 *ELSE* c_2, *s*,*n*) ⇒ *tn* |
IfFalse: ⟦ ¬*bval b s*; (c_2,*s*,*n*) ⇒ *tn* ⟧ ⟹
(*IF b THEN* c_1 *ELSE* c_2, *s*,*n*) ⇒ *tn* |

WhileFalse: ¬*bval b s* ⟹ (*WHILE b DO c*,*s*,*n*) ⇒ (*s*,*n*) |
WhileTrue:
⟦ *bval b* s_1; (*c*,s_1,*n*) ⇒ sn_2; (*WHILE b DO c*, sn_2) ⇒ sn_3 ⟧ ⟹
(*WHILE b DO c*, s_1,*n*) ⇒ sn_3

code_pred *big_step* **.**

declare [[*values_timeout* = *3600*]]

Examples:

definition
array_sum =
WHILE Less (!(*N 0*)) (*Plus* (!(*N 1*)) (*N 1*))
DO (*N 2* ::= *Plus* (!(*N 2*)) (!(!(*N 0*)));
N 0 ::= *Plus* (!(*N 0*)) (*N 1*))

To show the first n variables in a *nat* ⇒ *nat* state:

```
definition
 list t n = map t [0 ..< n]

values {list t n |t n. (array_sum, nth[3,4,0,3,7],5) ⇒ (t,n)}

definition
linked_list_sum =
 WHILE Less (N 0) (!(N 0))
 DO ( N 1 ::= Plus(!(N 1)) (!(!(N 0)));
      N 0 ::= !(Plus(!(N 0))(N 1)) )

values {list t n |t n. (linked_list_sum, nth[4,0,3,0,7,2],6) ⇒ (t,n)}

definition
array_init =
 New (N 0) (!(N 1)); N 2 ::= !(N 0);
 WHILE Less (!(N 2)) (Plus (!(N 0)) (!(N 1)))
 DO ( !(N 2) ::= !(N 2);
      N 2 ::= Plus (!(N 2)) (N 1) )

values {list t n |t n. (array_init, nth[5,2,7],3) ⇒ (t,n)}

definition
linked_list_init =
 WHILE Less (!(N 1)) (!(N 0))
 DO ( New (N 3) (N 2);
      N 1 ::=  Plus (!(N 1)) (N 1);
      !(N 3) ::= !(N 1);
      Plus (!(N 3)) (N 1) ::= !(N 2);
      N 2 ::= !(N 3) )

values {list t n |t n. (linked_list_init, nth[2,0,0,0],4) ⇒ (t,n)}

end
theory OO imports Main begin
```

15.3 Towards an OO Language: A Language of Records

```
abbreviation fun_upd2 :: ('a ⇒ 'b ⇒ 'c) ⇒ 'a ⇒ 'b ⇒ 'c ⇒ 'a ⇒ 'b ⇒ 'c
 (_/'((2_,_ :=/ _)') [1000,0,0,0] 900)
where f(x,y := z) == f(x := (f x)(y := z))

type_synonym addr = nat
```

```
datatype ref = null | Ref addr

type_synonym obj = string ⇒ ref
type_synonym venv = string ⇒ ref
type_synonym store = addr ⇒ obj

datatype exp =
  Null |
  New |
  V string |
  Faccess exp string        (_•/_ [63,1000] 63) |
  Vassign string exp        ((_ ::=/ _) [1000,61] 62) |
  Fassign exp string exp    ((_•_ ::=/ _) [63,0,62] 62) |
  Mcall exp string exp      ((_•/_<_>) [63,0,0] 63) |
  Seq exp exp               (_;/ _ [61,60] 60) |
  If bexp exp exp           (IF _/ THEN (2_)/ ELSE (2_) [0,0,61] 61)
and bexp = B bool | Not bexp | And bexp bexp | Eq exp exp

type_synonym menv = string ⇒ exp
type_synonym config = venv × store × addr

inductive
  big_step :: menv ⇒ exp × config ⇒ ref × config ⇒ bool
    ((_ ⊢/ (_/ ⇒ _)) [60,0,60] 55) and
  bval :: menv ⇒ bexp × config ⇒ bool × config ⇒ bool
    (_ ⊢ _ → _ [60,0,60] 55)
where
Null:
me ⊢ (Null,c) ⇒ (null,c) |
New:
me ⊢ (New,ve,s,n) ⇒ (Ref n,ve,s(n := (λf. null)),n+1) |
Vaccess:
me ⊢ (V x,ve,sn) ⇒ (ve x,ve,sn) |
Faccess:
me ⊢ (e,c) ⇒ (Ref a,ve',s',n') ⟹
 me ⊢ (e•f,c) ⇒ (s' a f,ve',s',n') |
Vassign:
me ⊢ (e,c) ⇒ (r,ve',sn') ⟹
 me ⊢ (x ::= e,c) ⇒ (r,ve'(x:=r),sn') |
Fassign:
⟦ me ⊢ (oe,c1) ⇒ (Ref a,c2);  me ⊢ (e,c2) ⇒ (r,ve3,s3,n3) ⟧ ⟹
 me ⊢ (oe•f ::= e,c1) ⇒ (r,ve3,s3(a,f := r),n3) |
Mcall:
⟦ me ⊢ (oe,c1) ⇒ (or,c2);  me ⊢ (pe,c2) ⇒ (pr,ve3,sn3);
```

$ve = (\lambda x.\ null)(''this'' := or,\ ''param'' := pr);$
$me \vdash (me\ m, ve, sn_3) \Rightarrow (r, ve', sn_4)\ \rrbracket$
$\Longrightarrow$
$me \vdash (oe{\cdot}m{<}pe{>}, c_1) \Rightarrow (r, ve_3, sn_4)\ |$
Seq:
$\llbracket\ me \vdash (e_1, c_1) \Rightarrow (r, c_2);\ \ me \vdash (e_2, c_2) \Rightarrow c_3\ \rrbracket \Longrightarrow$
$me \vdash (e_1;\ e_2, c_1) \Rightarrow c_3\ |$
IfTrue:
$\llbracket\ me \vdash (b, c_1) \rightarrow (True, c_2);\ \ me \vdash (e_1, c_2) \Rightarrow c_3\ \rrbracket \Longrightarrow$
$me \vdash (IF\ b\ THEN\ e_1\ ELSE\ e_2, c_1) \Rightarrow c_3\ |$
IfFalse:
$\llbracket\ me \vdash (b, c_1) \rightarrow (False, c_2);\ \ me \vdash (e_2, c_2) \Rightarrow c_3\ \rrbracket \Longrightarrow$
$me \vdash (IF\ b\ THEN\ e_1\ ELSE\ e_2, c_1) \Rightarrow c_3\ |$

$me \vdash (B\ bv, c) \rightarrow (bv, c)\ |$

$me \vdash (b, c_1) \rightarrow (bv, c_2) \Longrightarrow me \vdash (Not\ b, c_1) \rightarrow (\neg bv, c_2)\ |$

$\llbracket\ me \vdash (b_1, c_1) \rightarrow (bv_1, c_2);\ \ me \vdash (b_2, c_2) \rightarrow (bv_2, c_3)\ \rrbracket \Longrightarrow$
$me \vdash (And\ b_1\ b_2, c_1) \rightarrow (bv_1 \wedge bv_2, c_3)\ |$

$\llbracket\ me \vdash (e_1, c_1) \Rightarrow (r_1, c_2);\ \ me \vdash (e_2, c_2) \Rightarrow (r_2, c_3)\ \rrbracket \Longrightarrow$
$me \vdash (Eq\ e_1\ e_2, c_1) \rightarrow (r_1 = r_2, c_3)$

code_pred (*modes*: $i => i => o => bool$) *big_step* **.**

Example: natural numbers encoded as objects with a predecessor field. Null is zero. Method succ adds an object in front, method add adds as many objects in front as the parameter specifies.

First, the method bodies:

definition
$m\_succ\ =\ (''s'' ::= New){\cdot}''pred'' ::= V\ ''this'';\ V\ ''s''$

definition $m\_add =$
$IF\ Eq\ (V\ ''param'')\ Null$
$THEN\ V\ ''this''$
$ELSE\ V\ ''this''{\cdot}''succ''{<}Null{>}{\cdot}''add''{<}V\ ''param''{\cdot}''pred''{>}$

The method environment:

definition
$menv = (\lambda m.\ Null)(''succ'' := m\_succ,\ ''add'' := m\_add)$

The main code, adding 1 and 2:

definition $main =$

$''1'' ::= Null\bullet''succ''\text{<}Null\text{>};$
$''2'' ::= V\ ''1''\bullet''succ''\text{<}Null\text{>};$
$V\ ''2''\bullet\ ''add''\ \text{<}V\ ''1''\text{>}$

Execution of semantics. The final variable environment and store are converted into lists of references based on given lists of variable and field names to extract.

values
$\{(r,\ map\ ve'\ [''1'',''2''],\ map\ (\lambda n.\ map\ (s'\ n)[''pred''])\ [0..{<}n])|$
$\quad r\ ve'\ s'\ n.\ menv \vdash (main,\ \lambda x.\ null,\ nth[],\ 0) \Rightarrow (r,ve',s',n)\}$

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.

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