

Abstract Hoare Logics

Tobias Nipkow

December 12, 2009

Abstract

These theories describe Hoare logics for a number of imperative language constructs, from while-loops to mutually recursive procedures. Both partial and total correctness are treated. In particular a proof system for total correctness of recursive procedures in the presence of unbounded nondeterminism is presented.

Contents

1	Introduction	1
2	Hoare Logics for While	2
2.1	The language	2
2.2	Hoare logic for partial correctness	4
2.3	Termination	7
2.4	Hoare logic for total correctness	9
3	Hoare Logics for 1 Procedure	12
3.1	The language	12
3.2	Hoare logic for partial correctness	16
3.3	Termination	20
3.4	Hoare logic for total correctness	22
4	Hoare Logics for Mutually Recursive Procedure	38
4.1	The language	38
4.2	Hoare logic for partial correctness	41
4.3	Termination	45
4.4	Hoare logic for total correctness	47

1 Introduction

These are the theories underlying the publications [2, 1]. They should be consulted for explanatory text. The local variable declaration construct in [2] has been generalized; see Section 2.1.

2 Hoare Logics for While

theory *Lang* **imports** *Main* **begin**

2.1 The language

We start by declaring a type of states:

typedecl *state*

Our approach is completely parametric in the state space. We define expressions (*bexp*) as functions from states to the booleans:

types *bexp* = *state* \Rightarrow *bool*

Instead of modelling the syntax of boolean expressions, we model their semantics. The (abstract and concrete) syntax of our programming is defined as a recursive datatype:

```
datatype com = Do (state  $\Rightarrow$  state set)
  | Semi com com          (-; - [60, 60] 10)
  | Cond bexp com com    (IF - THEN - ELSE - 60)
  | While bexp com       (WHILE - DO - 60)
  | Local (state  $\Rightarrow$  state) com (state  $\Rightarrow$  state  $\Rightarrow$  state)
    (LOCAL -; -; - [0,0,60] 60)
```

Statements in this language are called *commands*. They are modelled as terms of type *com*. *Do f* represents an atomic nondeterministic command that changes the state from *s* to some element of *f s*. Thus the command that does nothing, often called **skip**, can be represented by *Do* ($\lambda s. \{s\}$). Again we have chosen to model the semantics rather than the syntax, which simplifies matters enormously. Of course it means that we can no longer talk about certain syntactic matters, but that is just fine.

The constructors *Semi*, *Cond* and *While* represent sequential composition, conditional and while-loop. The annotations allow us to write

$$c_1; c_2 \quad \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 \quad \text{WHILE } b \text{ DO } c$$

instead of *Semi* *c*₁ *c*₂, *Cond* *b* *c*₁ *c*₂ and *While* *b* *c*.

The command *LOCAL f; c; g* applies function *f* to the state, executes *c*, and then combines initial and final state via function *g*. More below. The semantics of commands is defined inductively by a so-called big-step semantics.

inductive

exec :: *state* \Rightarrow *com* \Rightarrow *state* \Rightarrow *bool* (-/ \dashrightarrow / - [50,0,50] 50)

where

$t \in f s \implies s -Do f \rightarrow t$

$\llbracket s_0 -c_1 \rightarrow s_1; s_1 -c_2 \rightarrow s_2 \rrbracket \implies s_0 -c_1;c_2 \rightarrow s_2$

$\mid \llbracket b \ s; \ s -c1 \rightarrow t \rrbracket \Longrightarrow s -IF \ b \ THEN \ c1 \ ELSE \ c2 \rightarrow t$
 $\mid \llbracket \neg b \ s; \ s -c2 \rightarrow t \rrbracket \Longrightarrow s -IF \ b \ THEN \ c1 \ ELSE \ c2 \rightarrow t$

$\mid \neg b \ s \Longrightarrow s -WHILE \ b \ DO \ c \rightarrow s$
 $\mid \llbracket b \ s; \ s -c \rightarrow t; \ t -WHILE \ b \ DO \ c \rightarrow u \rrbracket \Longrightarrow s -WHILE \ b \ DO \ c \rightarrow u$

$\mid f \ s -c \rightarrow t \Longrightarrow s -LOCAL \ f; \ c; \ g \rightarrow g \ s \ t$

Assuming that the state is a function from variables to values, the declaration of a new local variable x with initial value a can be modelled as $LOCAL \ (\lambda s. \ s(x := a \ s)); \ c; \ (\lambda s \ t. \ t(x := s \ x))$.

lemma *exec-Do-iff*[*iff*]: $(s -Do \ f \rightarrow t) = (t \in f \ s)$
by(*auto elim: exec.cases intro:exec.intros*)

lemma [*iff*]: $(s -c; d \rightarrow u) = (\exists t. \ s -c \rightarrow t \wedge t -d \rightarrow u)$
by(*best elim: exec.cases intro:exec.intros*)

lemma [*iff*]: $(s -IF \ b \ THEN \ c \ ELSE \ d \rightarrow t) =$
 $(s -if \ b \ s \ then \ c \ else \ d \rightarrow t)$
apply *auto*
apply(*blast elim: exec.cases intro:exec.intros*)
done

lemma [*iff*]: $(s -LOCAL \ f; \ c; \ g \rightarrow u) = (\exists t. \ f \ s -c \rightarrow t \wedge u = g \ s \ t)$
by(*fastsimp elim: exec.cases intro:exec.intros*)

lemma *unfold-while*:
 $(s -WHILE \ b \ DO \ c \rightarrow u) =$
 $(s -IF \ b \ THEN \ c; \ WHILE \ b \ DO \ c \ ELSE \ Do(\lambda s. \ \{s\}) \rightarrow u)$
by(*auto elim: exec.cases intro:exec.intros split:split-if-asm*)

lemma *while-lemma*[*rule-format*]:
 $s -w \rightarrow t \Longrightarrow !b \ c. \ w = WHILE \ b \ DO \ c \wedge P \ s \wedge$
 $(!s \ s'. \ P \ s \wedge b \ s \wedge s -c \rightarrow s' \longrightarrow P \ s') \longrightarrow P \ t \wedge \neg b \ t$
apply(*erule exec.induct*)
apply *clarify+*
defer
apply *clarify+*
apply(*subgoal-tac P t*)
apply *blast*
apply *blast*
done

lemma *while-rule*:
 $\llbracket s -WHILE \ b \ DO \ c \rightarrow t; \ P \ s; \ \forall s \ s'. \ P \ s \wedge b \ s \wedge s -c \rightarrow s' \longrightarrow P \ s' \rrbracket$
 $\Longrightarrow P \ t \wedge \neg b \ t$
apply(*drule while-lemma*)
prefer 2 **apply** *assumption*

apply *blast*
done

end

theory *Hoare* **imports** *Lang* **begin**

2.2 Hoare logic for partial correctness

We continue our semantic approach by modelling assertions just like boolean expressions, i.e. as functions:

types *assn* = *state* \Rightarrow *bool*

Hoare triples are triples of the form $\{P\} c \{Q\}$, where the assertions P and Q are the so-called pre and postconditions. Such a triple is *valid* (denoted by \models) iff every (terminating) execution starting in a state satisfying P ends up in a state satisfying Q :

constdefs

hoare-valid :: *assn* \Rightarrow *com* \Rightarrow *assn* \Rightarrow *bool* ($\models \{(1-)\} / (-) / \{(1-)\}$ 50)
 $\models \{P\}c\{Q\} \equiv \forall s t. s -c \rightarrow t \longrightarrow P s \longrightarrow Q t$

This notion of validity is called *partial correctness* because it does not require termination of c .

Provability in Hoare logic is indicated by \vdash and defined inductively:

inductive

hoare :: *assn* \Rightarrow *com* \Rightarrow *assn* \Rightarrow *bool* ($\vdash \{(1-)\} / (-) / \{(1-)\}$ 50)

where

$\vdash \{\lambda s. \forall t \in f s. P t\} Do f \{P\}$

$\mid \llbracket \vdash \{P\}c1\{Q\}; \vdash \{Q\}c2\{R\} \rrbracket \Longrightarrow \vdash \{P\} c1;c2 \{R\}$

$\mid \llbracket \vdash \{\lambda s. P s \wedge b s\} c1 \{Q\}; \vdash \{\lambda s. P s \wedge \neg b s\} c2 \{Q\} \rrbracket$
 $\Longrightarrow \vdash \{P\} IF b THEN c1 ELSE c2 \{Q\}$

$\mid \vdash \{\lambda s. P s \wedge b s\} c \{P\} \Longrightarrow \vdash \{P\} WHILE b DO c \{\lambda s. P s \wedge \neg b s\}$

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

$\mid \llbracket \bigwedge s. P s \Longrightarrow P' s (f s); \forall s. \vdash \{P' s\} c \{Q \circ (g s)\} \rrbracket \Longrightarrow$
 $\vdash \{P\} LOCAL f;c;g \{Q\}$

Soundness is proved by induction on the derivation of $\vdash \{P\} c \{Q\}$:

theorem *hoare-sound*: $\vdash \{P\}c\{Q\} \Longrightarrow \models \{P\}c\{Q\}$

apply(*unfold hoare-valid-def*)

apply(*erule hoare.induct*)

apply *blast*

```

    apply blast
  apply clarsimp
  apply clarify
  apply (drule while-rule)
  prefer 3
  apply (assumption, assumption, blast)
  apply blast
  apply clarify
  apply (erule allE)
  apply clarify
  apply (erule allE)
  apply (erule allE)
  apply (erule impE)
  apply assumption
  apply simp
  apply (erule mp)
  apply (simp)
done

```

Completeness is not quite as straightforward, but still easy. The proof is best explained in terms of the *weakest precondition*:

```

constdefs
  wp :: com ⇒ assn ⇒ assn
  wp c Q ≡ λs. ∀t. s -c→ t ⟶ Q t

```

Dijkstra calls this the weakest *liberal* precondition to emphasize that it corresponds to partial correctness. We use “weakest precondition” all the time and let the context determine if we talk about partial or total correctness — the latter is introduced further below.

The following lemmas about *wp* are easily derived:

```

lemma [simp]: wp (Do f) Q = (λs. ∀t ∈ f s. Q(t))
  apply (unfold wp-def)
  apply (rule ext)
  apply blast
done

```

```

lemma [simp]: wp (c1;c2) R = wp c1 (wp c2 R)
  apply (unfold wp-def)
  apply (rule ext)
  apply blast
done

```

```

lemma [simp]:
  wp (IF b THEN c1 ELSE c2) Q = (λs. wp (if b s then c1 else c2) Q s)
  apply (unfold wp-def)
  apply (rule ext)
  apply auto
done

```

```

lemma wp-while:
  wp (WHILE b DO c) Q =
    (λs. if b s then wp (c; WHILE b DO c) Q s else Q s)
apply(rule ext)
apply(unfold wp-def)
apply auto
apply(blast intro:exec.intros)
apply(simp add:unfold-while)
apply(blast intro:exec.intros)
apply(simp add:unfold-while)
done

```

```

lemma [simp]:
  wp (LOCAL f;c;g) Q = (λs. wp c (Q o (g s)) (f s))
apply(unfold wp-def)
apply(rule ext)
apply auto
done

```

```

lemma strengthen-pre:  $\llbracket \forall s. P' s \longrightarrow P s; \vdash \{P\}c\{Q\} \rrbracket \Longrightarrow \vdash \{P'\}c\{Q\}$ 
by(erule hoare.Conseq, assumption, blast)

```

```

lemma weaken-post:  $\llbracket \vdash \{P\}c\{Q\}; \forall s. Q s \longrightarrow Q' s \rrbracket \Longrightarrow \vdash \{P\}c\{Q'\}$ 
apply(rule hoare.Conseq)
apply(fast, assumption, assumption)
done

```

By induction on c one can easily prove

```

lemma wp-is-pre[rule-format]:  $\vdash \{wp\ c\ Q\} c \{Q\}$ 
apply (induct c arbitrary: Q)
apply simp-all

```

```

apply(blast intro:hoare.Do hoare.Conseq)

```

```

apply(blast intro:hoare.Semi hoare.Conseq)

```

```

apply(blast intro:hoare.If hoare.Conseq)

```

```

apply(rule weaken-post)
apply(rule hoare.While)
apply(rule strengthen-pre)
prefer 2
apply blast
apply(clarify)
apply(drule expand-fun-eq[THEN iffD1, OF wp-while, THEN spec, THEN iffD1])
apply simp
apply(clarify)
apply(drule expand-fun-eq[THEN iffD1, OF wp-while, THEN spec, THEN iffD1])
apply(simp split:split-if-asm)

```

apply(*fast intro!*: *hoare.Local*)
done

from which completeness follows more or less directly via the rule of consequence:

theorem *hoare-relative-complete*: $\models \{P\}c\{Q\} \implies \vdash \{P\}c\{Q\}$
apply (*rule strengthen-pre*[*OF - wp-is-pre*])
apply(*unfold hoare-valid-def wp-def*)
apply *blast*
done

end

theory *Termi* **imports** *Lang* **begin**

2.3 Termination

Although partial correctness appeals because of its simplicity, in many cases one would like the additional assurance that the command is guaranteed to terminate if started in a state that satisfies the precondition. Even to express this we need to define when a command is guaranteed to terminate. We can do this without modifying our existing semantics by merely adding a second inductively defined judgement $c \downarrow s$ that expresses guaranteed termination of c started in state s :

inductive

termi :: *com* \Rightarrow *state* \Rightarrow *bool* (**infixl** \downarrow 50)

where

$f s \neq \{\}$ \implies *Do* $f \downarrow s$

| $\llbracket c_1 \downarrow s_0; \forall s_1. s_0 -c_1 \rightarrow s_1 \longrightarrow c_2 \downarrow s_1 \rrbracket \implies (c_1; c_2) \downarrow s_0$

| $\llbracket b s; c_1 \downarrow s \rrbracket \implies$ *IF* b *THEN* c_1 *ELSE* $c_2 \downarrow s$

| $\llbracket \neg b s; c_2 \downarrow s \rrbracket \implies$ *IF* b *THEN* c_1 *ELSE* $c_2 \downarrow s$

| $\neg b s \implies$ *WHILE* b *DO* $c \downarrow s$

| $\llbracket b s; c \downarrow s; \forall t. s -c \rightarrow t \longrightarrow$ *WHILE* b *DO* $c \downarrow t \rrbracket \implies$ *WHILE* b *DO* $c \downarrow s$

| $c \downarrow f s \implies$ *LOCAL* $f; c; g \downarrow s$

lemma [*iff*]: $Do f \downarrow s = (f s \neq \{\})$

apply(*rule iffI*)

prefer 2

apply(*best intro:termi.intros*)

apply(*erule termi.cases*)

apply *blast+*
done

lemma [*iff*]: $((c_1; c_2) \downarrow s_0) = (c_1 \downarrow s_0 \wedge (\forall s_1. s_0 -c_1 \rightarrow s_1 \longrightarrow c_2 \downarrow s_1))$
apply(*rule iffI*)
prefer 2
apply(*best intro:termi.intros*)
apply(*erule termi.cases*)
apply *blast+*
done

lemma [*iff*]: $(IF\ b\ THEN\ c_1\ ELSE\ c_2\ \downarrow\ s) = ((if\ b\ s\ then\ c_1\ else\ c_2)\ \downarrow\ s)$
apply *simp*
apply(*rule conjI*)
apply(*rule impI*)
apply(*rule iffI*)
prefer 2
apply(*blast intro:termi.intros*)
apply(*erule termi.cases*)
apply *blast+*
apply(*rule impI*)
apply(*rule iffI*)
prefer 2
apply(*blast intro:termi.intros*)
apply(*erule termi.cases*)
apply *blast+*
done

lemma [*iff*]: $(LOCAL\ f; c; g\ \downarrow\ s) = (c\ \downarrow\ f\ s)$
by(*fast elim: termi.cases intro:termi.intros*)

lemma *termi-while-lemma*[*rule-format*]:
 $w \downarrow fk \implies$
 $(\forall k\ b\ c. fk = f\ k \wedge w = WHILE\ b\ DO\ c \wedge (\forall i. f\ i -c \rightarrow f(Suc\ i)) \longrightarrow (\exists i. \neg b(f\ i)))$
apply(*erule termi.induct*)
apply *simp-all*
apply *blast*
apply *blast*
done

lemma *termi-while*:
 $\llbracket (WHILE\ b\ DO\ c) \downarrow f\ k; \forall i. f\ i -c \rightarrow f(Suc\ i) \rrbracket \implies \exists i. \neg b(f\ i)$
by(*blast intro:termi-while-lemma*)

lemma *wf-termi*: $wf\ \{(t, s). WHILE\ b\ DO\ c\ \downarrow\ s \wedge b\ s \wedge s -c \rightarrow t\}$
apply(*subst wf-iff-no-infinite-down-chain*)
apply(*rule notI*)

```

apply clarsimp
apply(insert termi-while)
apply blast
done

end

```

```

theory HoareTotal imports Hoare Termi begin

```

2.4 Hoare logic for total correctness

Now that we have termination, we can define total validity, \models_t , as partial validity and guaranteed termination:

constdefs

```

hoare-tvalid :: assn  $\Rightarrow$  com  $\Rightarrow$  assn  $\Rightarrow$  bool ( $\models_t$   $\{(1-)\}/$   $(-)/$   $\{(1-)\}$  50)
 $\models_t \{P\}c\{Q\} \equiv \models \{P\}c\{Q\} \wedge (\forall s. P\ s \longrightarrow c\downarrow s)$ 

```

Proveability 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

```

thoare :: assn  $\Rightarrow$  com  $\Rightarrow$  assn  $\Rightarrow$  bool ( $\vdash_t$  ( $\{(1-)\}/$   $(-)/$   $\{(1-)\}$ ) 50)

```

where

```

| Do:  $\vdash_t \{\lambda s. (\forall t \in f\ s. P\ t) \wedge f\ s \neq \{\}\} Do\ f\ \{P\}$ 
| Semi:  $\llbracket \vdash_t \{P\}c\{Q\}; \vdash_t \{Q\}d\{R\} \rrbracket \Longrightarrow \vdash_t \{P\}c;d\{R\}$ 
| If:  $\llbracket \vdash_t \{\lambda s. P\ s \wedge b\ s\}c\{Q\}; \vdash_t \{\lambda s. P\ s \wedge \neg b\ s\}d\{Q\} \rrbracket \Longrightarrow$ 
 $\vdash_t \{P\} IF\ b\ THEN\ c\ ELSE\ d\ \{Q\}$ 
| While:
 $\llbracket wf\ r; \forall s'. \vdash_t \{\lambda s. P\ s \wedge b\ s \wedge s' = s\} c\ \{\lambda s. P\ s \wedge (s, s') \in r\} \rrbracket$ 
 $\Longrightarrow \vdash_t \{P\} WHILE\ b\ DO\ c\ \{\lambda s. P\ s \wedge \neg b\ s\}$ 
| Conseq:  $\llbracket \forall s. P'\ s \longrightarrow P\ s; \vdash_t \{P\}c\{Q\}; \forall s. Q\ s \longrightarrow Q'\ s \rrbracket \Longrightarrow$ 
 $\vdash_t \{P'\}c\{Q'\}$ 

| Local:  $(\llbracket !s. P\ s \Longrightarrow P'\ s\ (f\ s) \rrbracket \Longrightarrow \forall p. \vdash_t \{P'\ p\} c\ \{Q\ o\ (g\ p)\} \Longrightarrow$ 
 $\vdash_t \{P\} LOCAL\ f;c;g\ \{Q\}$ 

```

The *While*-rule is like the one for partial correctness but it requires additionally that with every execution of the loop body a wellfounded relation (*wf r*) on the state space decreases.

The soundness theorem

```

theorem  $\vdash_t \{P\}c\{Q\} \Longrightarrow \models_t \{P\}c\{Q\}$ 
apply(unfold hoare-tvalid-def hoare-valid-def)
apply(erule thoare.induct)
apply blast
apply blast

```

```

  apply clarsimp
  defer
  apply blast
  apply(rule conjI)
  apply clarify
  apply(erule allE)
  apply clarify
  apply(erule allE, erule allE, erule impE, erule asm-rl)
  apply simp
  apply(erule mp)
  apply(simp)
  apply blast
  apply(rule conjI)
  apply(rule allI)
  apply(erule wf-induct)
  apply clarify
  apply(drule unfold-while[THEN iffD1])
  apply (simp split add:split-if-asm)
  apply blast
  apply(rule allI)
  apply(erule wf-induct)
  apply clarify
  apply(case-tac b x)
  apply (blast intro: termi.WhileTrue)
  apply (erule termi.WhileFalse)
done

```

In the *While*-case we perform a local proof by wellfounded induction over the given relation r .

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:

constdefs

```

wpt :: com ⇒ assn ⇒ assn (wpt)
wpt c Q ≡ λs. wp c Q s ∧ c↓s

```

lemmas *wp-defs* = *wp-def wpt-def*

lemma [*simp*]: $wp_t (Do f) Q = (\lambda s. (\forall t \in f s. Q t) \wedge f s \neq \{\})$
by (*simp add: wpt-def*)

lemma [*simp*]: $wp_t (c_1; c_2) R = wp_t c_1 (wp_t c_2 R)$
apply (*unfold wp-defs*)
apply (*rule ext*)
apply *blast*
done

lemma [*simp*]:

```

  wpt (IF b THEN c1 ELSE c2) Q = (λs. wpt (if b s then c1 else c2) Q s)
apply(unfold wp-defs)
apply(rule ext)
apply auto
done

```

```

lemma [simp]: wpt (LOCAL f;c;g) Q = (λs. wpt c (Q o (g s)) (f s))
apply(unfold wp-defs)
apply(rule ext)
apply auto
done

```

```

lemma strengthen-pre: [ [ ∀ s. P' s → P s; ⊢t {P}c{Q} ] ] ⇒ ⊢t {P'}c{Q}
by(erule thoare.Conseq, assumption, blast)

```

```

lemma weaken-post: [ [ ⊢t {P}c{Q}; ∀ s. Q s → Q' s ] ] ⇒ ⊢t {P}c{Q'}
apply(rule thoare.Conseq)
apply(fast, assumption, assumption)
done

```

```

inductive-cases [elim!]: WHILE b DO c ↓ s

```

```

lemma wp-is-pre[rule-format]: ⊢t {wpt c Q} c {Q}
apply (induct c arbitrary: Q)
  apply simp-all
  apply(blast intro:thoare.Do thoare.Conseq)
  apply(blast intro:thoare.Semi thoare.Conseq)
  apply(blast intro:thoare.If thoare.Conseq)
  defer
  apply(fastsimp intro!: thoare.Local)
apply(rename-tac b c Q)
apply(rule weaken-post)
  apply(rule-tac b=b and c=c in thoare.While)
  apply(rule-tac b=b and c=c in wf-termi)
  defer
  apply (simp add:wp-defs unfold-while)
apply(rule allI)
apply(rule strengthen-pre)
  prefer 2
  apply fast
apply(clarsimp simp add: wp-defs)
apply(blast intro:exec.intros)
done

```

The *While*-case is interesting because we now have to furnish a suitable wellfounded relation. Of course the execution of the loop body directly yields the required relation. The actual completeness theorem follows directly, in the same manner as for partial correctness.

```

theorem ⊢t {P}c{Q} ⇒ ⊢t {P}c{Q}

```

```

apply (rule strengthen-pre[OF - wp-is-pre])
apply(unfold hoare-tvalid-def hoare-valid-def wp-defs)
apply blast
done

end

```

3 Hoare Logics for 1 Procedure

```

theory PLang imports Main begin

```

3.1 The language

```

typedecl state

```

```

types bexp = state  $\Rightarrow$  bool

```

```

datatype com = Do (state  $\Rightarrow$  state set)
  | Semi com com      (-; - [60, 60] 10)
  | Cond bexp com com (IF - THEN - ELSE - 60)
  | While bexp com     (WHILE - DO - 60)
  | CALL
  | Local (state  $\Rightarrow$  state) com (state  $\Rightarrow$  state  $\Rightarrow$  state)
  (LOCAL -; -; - [0,0,60] 60)

```

There is only one parameterless procedure in the program. Hence *CALL* does not even need to mention the procedure name. There is no separate syntax for procedure declarations. Instead we declare a HOL constant that represents the body of the one procedure in the program.

```

consts body :: com

```

As before, command execution is described by transitions $s -c \rightarrow t$. The only new rule is the one for *CALL* — it requires no comment:

```

inductive

```

```

  exec :: state  $\Rightarrow$  com  $\Rightarrow$  state  $\Rightarrow$  bool (-/ ->/ - [50,0,50] 50)

```

```

where

```

```

  Do:    t  $\in$  f s  $\Longrightarrow$  s -Do f  $\rightarrow$  t

```

```

  | Semi:  [ [ s0 -c1  $\rightarrow$  s1; s1 -c2  $\rightarrow$  s2 ]
             $\Longrightarrow$  s0 -c1;c2  $\rightarrow$  s2 ]

```

```

  | IfTrue:  [ [ b s; s -c1  $\rightarrow$  t ]  $\Longrightarrow$  s -IF b THEN c1 ELSE c2  $\rightarrow$  t
  | IfFalse: [ [  $\neg$ b s; s -c2  $\rightarrow$  t ]  $\Longrightarrow$  s -IF b THEN c1 ELSE c2  $\rightarrow$  t

```

```

  | WhileFalse:  $\neg$ b s  $\Longrightarrow$  s -WHILE b DO c  $\rightarrow$  s
  | WhileTrue:  [ [ b s; s -c  $\rightarrow$  t; t -WHILE b DO c  $\rightarrow$  u ]
                 $\Longrightarrow$  s -WHILE b DO c  $\rightarrow$  u

```

| $s - \text{body} \rightarrow t \implies s - \text{CALL} \rightarrow t$

| $\text{Local}: f s - c \rightarrow t \implies s - \text{LOCAL } f; c; g \rightarrow g s t$

lemma [iff]: $(s - \text{Do } f \rightarrow t) = (t \in f s)$
by(*auto elim: exec.cases intro:exec.intros*)

lemma [iff]: $(s - c; d \rightarrow u) = (\exists t. s - c \rightarrow t \wedge t - d \rightarrow u)$
by(*auto elim: exec.cases intro:exec.intros*)

lemma [iff]: $(s - \text{IF } b \text{ THEN } c \text{ ELSE } d \rightarrow t) =$
 $(s - \text{if } b \text{ s then } c \text{ else } d \rightarrow t)$
apply(*rule iffI*)
apply(*auto elim: exec.cases intro:exec.intros*)
apply(*auto intro:exec.intros split:split-if-asm*)
done

lemma *unfold-while*:
 $(s - \text{WHILE } b \text{ DO } c \rightarrow u) =$
 $(s - \text{IF } b \text{ THEN } c; \text{WHILE } b \text{ DO } c \text{ ELSE } \text{Do}(\lambda s. \{s\}) \rightarrow u)$
by(*auto elim: exec.cases intro:exec.intros split:split-if-asm*)

lemma [iff]: $(s - \text{CALL} \rightarrow t) = (s - \text{body} \rightarrow t)$
by(*blast elim: exec.cases intro:exec.intros*)

lemma [iff]: $(s - \text{LOCAL } f; c; g \rightarrow u) = (\exists t. f s - c \rightarrow t \wedge u = g s t)$
by(*fastsimp elim: exec.cases intro:exec.intros*)

lemma [simp]: $\neg b s \implies s - \text{WHILE } b \text{ DO } c \rightarrow s$
by(*fast intro:exec.intros*)

lemma *WhileI*: $\llbracket b s; s - c \rightarrow t; t - \text{WHILE } b \text{ DO } c \rightarrow u \rrbracket \implies s - \text{WHILE } b \text{ DO } c \rightarrow u$
by(*fastsimp elim:exec.WhileTrue*)

This semantics turns out not to be fine-grained enough. The soundness proof for the Hoare logic below proceeds by induction on the call depth during execution. To make this work we define a second semantics $s - c - n \rightarrow t$ which expresses that the execution uses at most n nested procedure invocations, where n is a natural number. The rules are straightforward: n is just passed around, except for procedure calls, where it is decremented:

inductive

execn :: *state* \Rightarrow *com* \Rightarrow *nat* \Rightarrow *state* \Rightarrow *bool* (-/ ---->/ - [50,0,0,50] 50)

where

$t \in f s \implies s - \text{Do } f - n \rightarrow t$

$| \llbracket s0 -c1-n \rightarrow s1; s1 -c2-n \rightarrow s2 \rrbracket \Longrightarrow s0 -c1;c2-n \rightarrow s2$
 $| \llbracket b\ s; s -c1-n \rightarrow t \rrbracket \Longrightarrow s -IF\ b\ THEN\ c1\ ELSE\ c2-n \rightarrow t$
 $| \llbracket \neg b\ s; s -c2-n \rightarrow t \rrbracket \Longrightarrow s -IF\ b\ THEN\ c1\ ELSE\ c2-n \rightarrow t$
 $| \neg b\ s \Longrightarrow s -WHILE\ b\ DO\ c-n \rightarrow s$
 $| \llbracket b\ s; s -c-n \rightarrow t; t -WHILE\ b\ DO\ c-n \rightarrow u \rrbracket \Longrightarrow s -WHILE\ b\ DO\ c-n \rightarrow u$
 $| s -body-n \rightarrow t \Longrightarrow s -CALL-Suc\ n \rightarrow t$
 $| f\ s -c-n \rightarrow t \Longrightarrow s -LOCAL\ f; c; g-n \rightarrow g\ s\ t$

lemma [*iff*]: $(s -Do\ f-n \rightarrow t) = (t \in f\ s)$
by(*auto elim: execn.cases intro:execn.intros*)

lemma [*iff*]: $(s -c1;c2-n \rightarrow u) = (\exists t. s -c1-n \rightarrow t \wedge t -c2-n \rightarrow u)$
by(*best elim: execn.cases intro:execn.intros*)

lemma [*iff*]: $(s -IF\ b\ THEN\ c\ ELSE\ d-n \rightarrow t) =$
 $(s -if\ b\ s\ then\ c\ else\ d-n \rightarrow t)$

apply *auto*
apply(*blast elim: execn.cases intro:execn.intros*)
done

lemma [*iff*]: $(s -CALL- 0 \rightarrow t) = False$
by(*blast elim: execn.cases intro:execn.intros*)

lemma [*iff*]: $(s -CALL-Suc\ n \rightarrow t) = (s -body-n \rightarrow t)$
by(*blast elim: execn.cases intro:execn.intros*)

lemma [*iff*]: $(s -LOCAL\ f; c; g-n \rightarrow u) = (\exists t. f\ s -c-n \rightarrow t \wedge u = g\ s\ t)$
by(*auto elim: execn.cases intro:execn.intros*)

By induction on $s -c-m \rightarrow t$ we show monotonicity w.r.t. the call depth:

lemma *exec-mono*[*rule-format*]: $s -c-m \rightarrow t \Longrightarrow \forall n. m \leq n \longrightarrow s -c-n \rightarrow t$
apply(*erule execn.induct*)
apply(*blast*)
apply(*blast*)
apply(*simp*)
apply(*simp*)
apply(*simp add:execn.intros*)
apply(*blast intro:execn.intros*)
apply(*clarify*)
apply(*rename-tac m*)
apply(*case-tac m*)
apply *simp*
apply *simp*

apply *blast*
done

With the help of this lemma we prove the expected relationship between the two semantics:

lemma *exec-iff-execn*: $(s -c \rightarrow t) = (\exists n. s -c-n \rightarrow t)$
apply(*rule iffI*)
apply(*erule exec.induct*)
 apply *blast*
 apply *clarify*
 apply(*rename-tac m n*)
 apply(*rule-tac x = max m n in exI*)
 apply(*fastsimp intro:exec.intros exec-mono simp add:max-def*)
 apply *fastsimp*
 apply *fastsimp*
 apply(*blast intro:execn.intros*)
 apply *clarify*
 apply(*rename-tac m n*)
 apply(*rule-tac x = max m n in exI*)
 apply(*fastsimp elim:execn.WhileTrue exec-mono simp add:max-def*)
 apply *blast*
 apply *blast*
apply(*erule exE, erule execn.induct*)
 apply *blast*
 apply *blast*
 apply *fastsimp*
 apply *fastsimp*
 apply(*erule exec.WhileFalse*)
 apply(*blast intro: exec.intros*)
 apply *blast*
 apply *blast*
done

lemma *while-lemma*[*rule-format*]:
 $s -w-n \rightarrow t \implies !b c. w = \text{WHILE } b \text{ DO } c \wedge P s \wedge$
 $(!s s'. P s \wedge b s \wedge s -c-n \rightarrow s' \implies P s') \implies P t \wedge \neg b t$
apply(*erule execn.induct*)
apply *clarify+*
defer
apply *clarify+*
apply(*subgoal-tac P t*)
apply *blast*
apply *blast*
done

lemma *while-rule*:
 $\llbracket s - \text{WHILE } b \text{ DO } c -n \rightarrow t; P s; \bigwedge s s'. \llbracket P s; b s; s -c-n \rightarrow s' \rrbracket \implies P s \rrbracket$
 $\implies P t \wedge \neg b t$

```

apply(drule while-lemma)
prefer 2 apply assumption
apply blast
done

end

```

```

theory PHoare imports PLang begin

```

3.2 Hoare logic for partial correctness

Taking auxiliary variables seriously means that assertions must now depend on them as well as on the state. Initially we do not fix the type of auxiliary variables but parameterize the type of assertions with a type variable $'a$:

```

types  $'a$  assn =  $'a \Rightarrow state \Rightarrow bool$ 

```

The second major change is the need to reason about Hoare triples in a context: proofs about recursive procedures are conducted by induction where we assume that all *CALLS* satisfy the given pre/postconditions and have to show that the body does as well. The assumption is stored in a context, which is a set of Hoare triples:

```

types  $'a$  cntxt = ( $'a$  assn  $\times$  com  $\times$   $'a$  assn)set

```

In the presence of only a single procedure the context will always be empty or a singleton set. With multiple procedures, larger sets can arise.

Now that we have contexts, validity becomes more complicated. Ordinary validity (w.r.t. partial correctness) is still what it used to be, except that we have to take auxiliary variables into account as well:

```

constdefs

```

```

valid ::  $'a$  assn  $\Rightarrow$  com  $\Rightarrow$   $'a$  assn  $\Rightarrow$  bool ( $\models$   $\{(1-)\}/ (-)/ \{(1-)\}$  50)
 $\models \{P\}c\{Q\} \equiv \forall s t. s -c \rightarrow t \longrightarrow (\forall z. P z s \longrightarrow Q z t)$ 

```

Auxiliary variables are always denoted by z .

Validity of a context and validity of a Hoare triple in a context are defined as follows:

```

constdefs

```

```

valids ::  $'a$  cntxt  $\Rightarrow$  bool ( $\models -$  50)
 $\models C \equiv \forall (P,c,Q) \in C. \models \{P\}c\{Q\}$ 

```

```

cvalid ::  $'a$  cntxt  $\Rightarrow$   $'a$  assn  $\Rightarrow$  com  $\Rightarrow$   $'a$  assn  $\Rightarrow$  bool ( $- \models / \{(1-)\}/ (-)/ \{(1-)\}$  50)

```

```

 $C \models \{P\}c\{Q\} \equiv \models C \longrightarrow \models \{P\}c\{Q\}$ 

```

```

declare valids-def[simp]

```

Note that $\{\} \models \{P\} c \{Q\}$ is equivalent to $\models \{P\} c \{Q\}$.

Unfortunately, this is not the end of it. As we have two semantics, $-c \rightarrow$ and $-c-n \rightarrow$, we also need a second notion of validity parameterized with the recursion depth n :

constdefs

$$\begin{aligned} nvalid &:: nat \Rightarrow 'a\ assn \Rightarrow com \Rightarrow 'a\ assn \Rightarrow bool\ (\models\ \{(1-\)/\ (-)\}\ \{(1-\)\}\ 50) \\ \models_n\ \{P\}c\{Q\} &\equiv \forall s\ t.\ s\ -c-n \rightarrow t \longrightarrow (\forall z.\ P\ z\ s \longrightarrow Q\ z\ t) \end{aligned}$$

$$\begin{aligned} nvalids &:: nat \Rightarrow 'a\ cntxt \Rightarrow bool\ (\models\ \{-\}\ -\ 50) \\ \models_n\ C &\equiv \forall (P,c,Q) \in C.\ \models_n\ \{P\}c\{Q\} \end{aligned}$$

$$\begin{aligned} cvalid &:: 'a\ cntxt \Rightarrow nat \Rightarrow 'a\ assn \Rightarrow com \Rightarrow 'a\ assn \Rightarrow bool\ (\models\ \{-\}\ \{(1-\)/\ (-)\}\ \{(1-\)\}\ 50) \\ C \models_n\ \{P\}c\{Q\} &\equiv \models_n\ C \longrightarrow \models_n\ \{P\}c\{Q\} \end{aligned}$$

Finally we come to the proof system for deriving triples in a context:

inductive

$$hoare :: 'a\ cntxt \Rightarrow 'a\ assn \Rightarrow com \Rightarrow 'a\ assn \Rightarrow bool\ (\vdash\ \{(1-\)/\ (-)\}\ \{(1-\)\}\ 50)$$

where

$$C \vdash \{\lambda z\ s.\ \forall t \in f\ s.\ P\ z\ t\}\ Do\ f\ \{P\}$$

$$\llbracket C \vdash \{P\}c1\{Q\}; C \vdash \{Q\}c2\{R\} \rrbracket \Longrightarrow C \vdash \{P\}\ c1;c2\ \{R\}$$

$$\llbracket C \vdash \{\lambda z\ s.\ P\ z\ s \wedge b\ s\}c1\{Q\}; C \vdash \{\lambda z\ s.\ P\ z\ s \wedge \neg b\ s\}c2\{Q\} \rrbracket \Longrightarrow C \vdash \{P\}\ IF\ b\ THEN\ c1\ ELSE\ c2\ \{Q\}$$

$$\llbracket C \vdash \{\lambda z\ s.\ P\ z\ s \wedge b\ s\}\ c\ \{P\} \rrbracket \Longrightarrow C \vdash \{P\}\ WHILE\ b\ DO\ c\ \{\lambda z\ s.\ P\ z\ s \wedge \neg b\ s\}$$

$$\llbracket C \vdash \{P'\}c\{Q'\}; \forall s\ t.\ (\forall z.\ P'\ z\ s \longrightarrow Q'\ z\ t) \longrightarrow (\forall z.\ P\ z\ s \longrightarrow Q\ z\ t) \rrbracket \Longrightarrow C \vdash \{P\}c\{Q\}$$

$$\llbracket (P,CALL,Q) \rrbracket \vdash \{P\}body\{Q\} \Longrightarrow \{\}\ \vdash \{P\}\ CALL\ \{Q\}$$

$$\begin{aligned} &\llbracket (P,CALL,Q) \rrbracket \vdash \{P\}\ CALL\ \{Q\} \\ &\llbracket \forall s'. C \vdash \{\lambda z\ s.\ P\ z\ s' \wedge s = f\ s'\}\ c\ \{\lambda z\ t.\ Q\ z\ (g\ s'\ t)\} \rrbracket \Longrightarrow \\ &\quad C \vdash \{P\}\ LOCAL\ f;c;g\ \{Q\} \end{aligned}$$

$$\begin{aligned} \text{abbreviation } hoare1 &:: 'a\ cntxt \Rightarrow 'a\ assn \times com \times 'a\ assn \Rightarrow bool\ (\vdash\ \{-\}\ -) \text{ where} \\ C \vdash x &\equiv C \vdash \{fst\ x\}fst\ (snd\ x)\{snd\ (snd\ x)\} \end{aligned}$$

The first four rules are familiar, except for their adaptation to auxiliary variables. The *CALL* rule embodies induction and has already been motivated above. Note that it is only applicable if the context is empty. This shows that we never need nested induction. For the same reason the assumption rule (the last rule) is stated with just a singleton context.

The rule of consequence is explained in the accompanying paper.

lemma *strengthen-pre*:

$\llbracket \forall z s. P' z s \longrightarrow P z s; C \vdash \{P\}c\{Q\} \rrbracket \implies C \vdash \{P'\}c\{Q\}$
by(*rule hoare.Conseq, assumption, blast*)

lemmas *valid-defs = valid-def validates-def cvalid-def
nvalid-def nvalids-def cvalid-def*

theorem *hoare-sound*: $C \vdash \{P\}c\{Q\} \implies C \models \{P\}c\{Q\}$

requires a generalization: $\forall n. C \models_n \{P\} c \{Q\}$ is proved instead, from which the actual theorem follows directly via lemma *exec-iff-execn* in §???. The generalization is proved by induction on c . The reason for the generalization is that soundness of the *CALL* rule is proved by induction on the maximal call depth, i.e. n .

apply(*subgoal-tac* $\forall n. C \models_n \{P\}c\{Q\}$)
apply(*unfold valid-defs exec-iff-execn*[*THEN eq-reflection*])
apply *fast*
apply(*erule hoare.induct*)
 apply *simp*
 apply *fast*
 apply *simp*
 apply *clarify*
 apply(*erule while-rule*)
 prefer 3
 apply (*assumption, assumption*)
 apply *fast*
 apply *fast*
 prefer 2
 apply *simp*
apply(*rule allI, rule impI*)
apply(*induct-tac n*)
 apply *blast*
 apply *clarify*
 apply (*simp*(*no-asm-use*))
 apply *blast*
apply *auto*
done

The completeness proof employs the notion of a *most general triple* (or *most general formula*):

constdefs
MGT :: *com* \Rightarrow *state assn* \times *com* \times *state assn*
MGT $c \equiv (\lambda z s. z = s, c, \lambda z t. z -c \rightarrow t)$

declare *MGT-def*[*simp*]

Note that the type of z has been identified with *state*. This means that for every state variable there is an auxiliary variable, which is simply there to record the value of the program variables before execution of a command. This is exactly what, for example, VDM offers by allowing you to refer to the pre-value of a variable in a postcondition. The intuition behind *MGT c*

is that it completely describes the operational behaviour of c . It is easy to see that, in the presence of the new consequence rule, $\{\} \vdash MGT\ c$ implies completeness:

lemma *MGT-implies-complete*:
 $\{\} \vdash MGT\ c \implies \{\} \models \{P\}c\{Q\} \implies \{\} \vdash \{P\}c\{Q::state\ assn\}$
apply (*simp add: MGT-def*)
apply (*erule hoare.Conseq*)
apply (*simp add: valid-defs*)
done

In order to discharge $\{\} \vdash MGT\ c$ one proves

lemma *MGT-lemma*: $C \vdash MGT\ CALL \implies C \vdash MGT\ c$
apply (*simp*)
apply (*induct-tac c*)
apply (*rule strengthen-pre[OF - hoare.Do]*)
apply *blast*
apply (*blast intro:hoare.Semi hoare.Conseq*)
apply (*rule hoare.If*)
apply (*erule hoare.Conseq*)
apply *simp*
apply (*erule hoare.Conseq*)
apply *simp*
prefer 2
apply *simp*
apply (*rename-tac b c*)
apply (*rule hoare.Conseq*)
apply (*rule-tac P = $\lambda z\ s. (z, s) \in (\{(s, t). b\ s \wedge s -c \rightarrow t\})^*$*
in *hoare.While*)
apply (*erule hoare.Conseq*)
apply (*blast intro:rtrancl-into-rtrancl*)
apply *clarsimp*
apply (*rename-tac s t*)
apply (*erule-tac x = s in allE*)
apply *clarsimp*
apply (*erule converse-rtrancl-induct*)
apply *simp*
apply (*fast elim:exec.WhileTrue*)
apply (*fastsimp intro: hoare.Local elim!: hoare.Conseq*)
done

The proof is by induction on c . In the *While*-case it is easy to show that $\lambda z\ t. (z, t) \in \{(s, t). b\ s \wedge s -c \rightarrow t\}^*$ is invariant. The precondition $\lambda z\ s. z=s$ establishes the invariant and a reflexive transitive closure induction shows that the invariant conjoined with $\neg b\ t$ implies the postcondition $\lambda z. exec\ z\ (WHILE\ b\ DO\ c)$. The remaining cases are trivial.

Using the *MGT-lemma* (together with the *CALL* and the assumption rule) one can easily derive

lemma *MGT-CALL*: $\{\} \vdash MGT\ CALL$

```

apply(simp add: MGT-def)
apply (rule hoare.Call)
apply (rule hoare.Conseq[OF MGT-lemma[simplified], OF hoare.Asm])
apply (fast intro:exec.intros)
done

```

Using the *MGT-lemma* once more we obtain $\{\} \vdash MGT\ c$ and thus by *MGT-implies-complete* completeness.

```

theorem  $\{\} \models \{P\}c\{Q\} \implies \{\} \vdash \{P\}c\{Q::state\ assn\}$ 
apply(erule MGT-implies-complete[OF MGT-lemma[OF MGT-CALL]])
done

```

end

theory *PTermi* **imports** *PLang* **begin**

3.3 Termination

inductive

termi :: *com* \Rightarrow *state* \Rightarrow *bool* (**infixl** \downarrow 50)

where

```

  Do[iff]:  $f\ s \neq \{\} \implies Do\ f\ \downarrow\ s$ 
| Semi[intro!]:  $\llbracket c1\ \downarrow\ s0; \bigwedge s1. s0 -c1 \rightarrow s1 \implies c2\ \downarrow\ s1 \rrbracket$ 
   $\implies (c1;c2)\ \downarrow\ s0$ 

| IfTrue[intro,simp]:  $\llbracket b\ s; c1\ \downarrow\ s \rrbracket \implies IF\ b\ THEN\ c1\ ELSE\ c2\ \downarrow\ s$ 
| IfFalse[intro,simp]:  $\llbracket \neg b\ s; c2\ \downarrow\ s \rrbracket \implies IF\ b\ THEN\ c1\ ELSE\ c2\ \downarrow\ s$ 

| WhileFalse:  $\neg b\ s \implies WHILE\ b\ DO\ c\ \downarrow\ s$ 

| WhileTrue:  $\llbracket b\ s; c\ \downarrow\ s; \bigwedge t. s -c \rightarrow t \implies WHILE\ b\ DO\ c\ \downarrow\ t \rrbracket$ 
   $\implies WHILE\ b\ DO\ c\ \downarrow\ s$ 
| body  $\downarrow\ s \implies CALL\ \downarrow\ s$ 

| Local:  $c\ \downarrow\ f\ s \implies LOCAL\ f;c;g\ \downarrow\ s$ 

```

lemma [*iff*]: $(Do\ f\ \downarrow\ s) = (f\ s \neq \{\})$

apply(rule *iffI*)

prefer 2

apply(best intro:*termi.intros*)

apply(erule *termi.cases*)

apply *blast+*

done

lemma [*iff*]: $((c1;c2)\ \downarrow\ s0) = (c1\ \downarrow\ s0 \wedge (\forall s1. s0 -c1 \rightarrow s1 \longrightarrow c2\ \downarrow\ s1))$

apply(rule *iffI*)

prefer 2

apply(best intro:*termi.intros*)

apply(*erule termi.cases*)
apply *blast+*
done

lemma [*iff*]: (*IF b THEN c1 ELSE c2* \downarrow *s*) =
 $((\text{if } b \text{ s then } c1 \text{ else } c2) \downarrow s)$

apply *simp*
apply(*rule conjI*)
apply(*rule impI*)
apply(*rule iffI*)
prefer 2
apply(*blast intro:termi.intros*)
apply(*erule termi.cases*)
apply *blast+*
apply(*rule impI*)
apply(*rule iffI*)
prefer 2
apply(*blast intro:termi.intros*)
apply(*erule termi.cases*)
apply *blast+*
done

lemma [*iff*]: (*CALL* \downarrow *s*) = (*body* \downarrow *s*)
by(*fast elim: termi.cases intro:termi.intros*)

lemma [*iff*]: (*LOCAL f;c;g* \downarrow *s*) = (*c* \downarrow *f s*)
by(*fast elim: termi.cases intro:termi.intros*)

lemma *termi-while-lemma*[*rule-format*]:

$w \downarrow fk \implies$
 $(\forall k \ b \ c. \ fk = f \ k \wedge \ w = \text{WHILE } b \ \text{DO } c \wedge (\forall i. \ f \ i \ -c \rightarrow f(\text{Suc } i))$
 $\longrightarrow (\exists i. \ \neg b(f \ i)))$

apply(*erule termi.induct*)
apply *simp-all*
apply *blast*
apply *blast*
done

lemma *termi-while*:

$\llbracket (\text{WHILE } b \ \text{DO } c) \downarrow f \ k; \ \forall i. \ f \ i \ -c \rightarrow f(\text{Suc } i) \rrbracket \implies \exists i. \ \neg b(f \ i)$
by(*blast intro:termi-while-lemma*)

lemma *wf-termi*: *wf* $\{(t,s). \ \text{WHILE } b \ \text{DO } c \downarrow s \wedge b \ s \wedge s \ -c \rightarrow t\}$

apply(*subst wf-iff-no-infinite-down-chain*)
apply(*rule notI*)
apply *clarsimp*
apply(*insert termi-while*)
apply *blast*
done

end

theory *PHoareTotal* **imports** *PHoare PTermi* **begin**

3.4 Hoare logic for total correctness

Validity is defined as expected:

constdefs

$$\begin{aligned} tvalid &:: 'a\ assn \Rightarrow com \Rightarrow 'a\ assn \Rightarrow bool \ (\models_t \{(1-)\} / (-) / \{(1-)\} \ 50) \\ \models_t \{P\}c\{Q\} &\equiv \models \{P\}c\{Q\} \wedge (\forall z\ s. P\ z\ s \longrightarrow c\downarrow s) \end{aligned}$$

$$\begin{aligned} ctvalid &:: 'a\ cntxt \Rightarrow 'a\ assn \Rightarrow com \Rightarrow 'a\ assn \Rightarrow bool \\ &\quad ((- / \models_t \{(1-)\} / (-) / \{(1-)\}) \ 50) \end{aligned}$$

$$C \models_t \{P\}c\{Q\} \equiv (\forall (P',c',Q') \in C. \models_t \{P'\}c'\{Q'\}) \longrightarrow \models_t \{P\}c\{Q\}$$

inductive

$$\begin{aligned} thoare &:: 'a\ cntxt \Rightarrow 'a\ assn \Rightarrow com \Rightarrow 'a\ assn \Rightarrow bool \\ &\quad ((- \vdash_t / (\{(1-)\} / (-) / \{(1-)\})) \ [50,0,0,0] \ 50) \end{aligned}$$

where

$$\begin{aligned} | Do: & C \vdash_t \{\lambda z\ s. (\forall t \in f\ s. P\ z\ t) \wedge f\ s \neq \{\}\} \ Do\ f\ \{P\} \\ | Semi: & \llbracket C \vdash_t \{P\}c1\{Q\}; C \vdash_t \{Q\}c2\{R\} \rrbracket \Longrightarrow C \vdash_t \{P\} \ c1;c2\ \{R\} \\ | If: & \llbracket C \vdash_t \{\lambda z\ s. P\ z\ s \wedge b\ s\}c\{Q\}; C \vdash_t \{\lambda z\ s. P\ z\ s \wedge \sim b\ s\}d\{Q\} \rrbracket \Longrightarrow \\ & C \vdash_t \{P\} \ IF\ b\ THEN\ c\ ELSE\ d\ \{Q\} \end{aligned}$$

| *While*:

$$\begin{aligned} &\llbracket wf\ r; \forall s'. C \vdash_t \{\lambda z\ s. P\ z\ s \wedge b\ s \wedge s' = s\} \ c\ \{\lambda z\ s. P\ z\ s \wedge (s,s') \in r\} \rrbracket \\ &\Longrightarrow C \vdash_t \{P\} \ WHILE\ b\ DO\ c\ \{\lambda z\ s. P\ z\ s \wedge \neg b\ s\} \end{aligned}$$

| *Call*:

$$\begin{aligned} &\llbracket wf\ r; \forall s'. \{(\lambda z\ s. P\ z\ s \wedge (s,s') \in r, \ CALL, Q)\} \\ &\quad \vdash_t \{\lambda z\ s. P\ z\ s \wedge s = s'\} \ body\ \{Q\} \rrbracket \\ &\Longrightarrow \{\} \vdash_t \{P\} \ CALL\ \{Q\} \end{aligned}$$

| *Asm*: $\{(P, CALL, Q)\} \vdash_t \{P\} \ CALL\ \{Q\}$

| *Conseq*:

$$\begin{aligned} &\llbracket C \vdash_t \{P'\}c\{Q'\}; \\ &\quad (\forall s\ t. (\forall z. P'\ z\ s \longrightarrow Q'\ z\ t) \longrightarrow (\forall z. P\ z\ s \longrightarrow Q\ z\ t)) \wedge \\ &\quad (\forall s. (\exists z. P\ z\ s) \longrightarrow (\exists z. P'\ z\ s)) \rrbracket \\ &\Longrightarrow C \vdash_t \{P\}c\{Q\} \end{aligned}$$

| *Local*: $\llbracket \forall s'. C \vdash_t \{\lambda z\ s. P\ z\ s' \wedge s = f\ s'\} \ c\ \{\lambda z\ t. Q\ z\ (g\ s'\ t)\} \rrbracket \Longrightarrow$
 $C \vdash_t \{P\} \ LOCAL\ f;c;g\ \{Q\}$

abbreviation *hoare1* :: $'a\ cntxt \Rightarrow 'a\ assn \times com \times 'a\ assn \Rightarrow bool \ (- \vdash_t -)$ **where**

$$C \vdash_t x \equiv C \vdash_t \{fst\ x\}fst\ (snd\ x)\{snd\ (snd\ x)\}$$

The side condition in our rule of consequence looks quite different from the one by Kleymann, but the two are in fact equivalent:

$$\begin{aligned} \text{lemma } & ((\forall s t. (\forall z. P' z s \longrightarrow Q' z t) \longrightarrow (\forall z. P z s \longrightarrow Q z t)) \wedge \\ & (\forall s. (\exists z. P z s) \longrightarrow (\exists z. P' z s))) \\ & = (\forall z s. P z s \longrightarrow (\forall t. \exists z'. P' z' s \wedge (Q' z' t \longrightarrow Q z t))) \end{aligned}$$

by *blast*

The key difference to the work by Kleymann (and America and de Boer) is that soundness and completeness are shown for arbitrary, i.e. unbounded nondeterminism. This is a significant extension and appears to have been an open problem. The details are found below and are explained in a separate paper [1].

lemma *strengthen-pre*:

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

by(*rule thoare.Conseq, assumption, blast*)

lemma *weaken-post*:

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

by(*erule thoare.Conseq, blast*)

theorems *tvalid-defs = tvalid-def ctvalid-def valid-defs*

lemma [*iff*]:

$$(\models_t \{\lambda z s. \exists n. P n z s\}c\{Q\}) = (\forall n. \models_t \{P n\}c\{Q\})$$

apply(*unfold tvalid-defs*)

apply *fast*

done

lemma [*iff*]:

$$(\models_t \{\lambda z s. P z s \wedge P'\}c\{Q\}) = (P' \longrightarrow \models_t \{P\}c\{Q\})$$

apply(*unfold tvalid-defs*)

apply *fast*

done

lemma [*iff*]: $(\models_t \{P\}CALL\{Q\}) = (\models_t \{P\}body\{Q\})$

apply(*unfold tvalid-defs*)

apply *fast*

done

theorem $C \vdash_t \{P\}c\{Q\} \Longrightarrow C \models_t \{P\}c\{Q\}$

apply(*erule thoare.induct*)

apply(*simp only:tvalid-defs*)

apply *fast*

apply(*simp only:tvalid-defs*)

apply *fast*

apply(*simp only:tvalid-defs*)

apply *clarsimp*

```

prefer 3
apply(simp add:tvalid-defs)
prefer 3
apply(simp only:tvalid-defs)
apply blast
apply(simp only:tvalid-defs)
apply(rule impI, rule conjI)
apply(rule allI)
apply(erule wf-induct)
apply clarify
apply(drule unfold-while[THEN iffD1])
apply (simp split add:split-if-asm)
apply fast
apply(rule allI, rule allI)
apply(erule wf-induct)
apply clarify
apply(case-tac b x)
prefer 2
apply (erule termi.WhileFalse)
apply(rule termi.WhileTrue, assumption)
apply fast
apply (subgoal-tac (t,x):r)
apply fast
apply blast
apply(simp (no-asm-use) add:ctvalid-def)
apply(subgoal-tac !n.  $\models_t \{\lambda z s. P z s \ \& \ s=n\}$  body {Q})
apply(simp (no-asm-use) add:tvalid-defs)
apply blast
apply(rule allI)
apply(erule wf-induct)
apply(unfold tvalid-defs)
apply fast
apply fast
done

```

constdefs MGT_t :: $com \Rightarrow state \ assn \times com \times state \ assn$

$MGT_t \ c \equiv (\lambda z s. z = s \wedge c \downarrow s, c, \lambda z t. z -c \rightarrow t)$

declare MGT_t -def[simp]

lemma *MGT-implies-complete*:

$\{\} \vdash_t MGT_t \ c \Longrightarrow \{\} \models_t \{P\}c\{Q\} \Longrightarrow \{\} \vdash_t \{P\}c\{Q::state \ assn\}$

apply(simp add: MGT_t -def)

apply (erule thoare.Conseq)

apply(simp add: tvalid-defs)

apply blast

done

lemma *while-termiE*: $\llbracket WHILE \ b \ DO \ c \ \downarrow \ s; \ b \ s \rrbracket \Longrightarrow c \ \downarrow \ s$

```

by(erule termi.cases, auto)

lemma while-termiE2:
  [| WHILE b DO c ↓ s; b s; s -c→ t |] ⇒ WHILE b DO c ↓ t
by(erule termi.cases, auto)

lemma MGT-lemma: C ⊢t MGTt CALL ⇒ C ⊢t MGTt c
apply (simp)
apply(induct-tac c)
  apply (rule strengthen-pre[OF - thoare.Do])
  apply blast
  apply(rule-tac Q = λz s. z -com1→s & com2↓s in thoare.Semi)
  apply(erule thoare.Conseq)
  apply fast
  apply(erule thoare.Conseq)
  apply fast
  apply(rule thoare.If)
  apply(erule thoare.Conseq)
  apply simp
  apply(erule thoare.Conseq)
  apply simp
  defer
  apply simp
  apply(fast intro:thoare.Local elim!: thoare.Conseq)
  apply(rename-tac b c)
  apply(rule-tac P' = λz s. (z,s) ∈ ({(s,t). b s ∧ s -c→ t})* ∧
    WHILE b DO c ↓ s in thoare.Conseq)
  apply(rule-tac thoare.While[OF wf-termi])
  apply(rule allI)
  apply(erule thoare.Conseq)
  apply(fastsimp intro:rtrancl-into-rtrancl dest:while-termiE while-termiE2)
  apply(rule conjI)
  apply clarsimp
  apply(erule-tac x = s in allE)
  apply clarsimp
  apply(erule converse-rtrancl-induct)
  apply simp
  apply(fast elim:exec.WhileTrue)
  apply(fast intro: rtrancl-refl)
done

inductive-set
  exec1 :: ((com list × state) × (com list × state))set
  and exec1' :: (com list × state) ⇒ (com list × state) ⇒ bool (- → - [81,81]
100)
where
  cs0 → cs1 ≡ (cs0,cs1) : exec1

```

| *Do*[*iff*]: $t \in f s \implies ((Do f) \# cs, s) \rightarrow (cs, t)$

| *Semi*[*iff*]: $((c1; c2) \# cs, s) \rightarrow (c1 \# c2 \# cs, s)$

| *IfTrue*: $b s \implies ((IF b THEN c1 ELSE c2) \# cs, s) \rightarrow (c1 \# cs, s)$

| *IfFalse*: $\neg b s \implies ((IF b THEN c1 ELSE c2) \# cs, s) \rightarrow (c2 \# cs, s)$

| *WhileFalse*: $\neg b s \implies ((WHILE b DO c) \# cs, s) \rightarrow (cs, s)$

| *WhileTrue*: $b s \implies ((WHILE b DO c) \# cs, s) \rightarrow (c \# (WHILE b DO c) \# cs, s)$

| *Call*[*iff*]: $(CALL \# cs, s) \rightarrow (body \# cs, s)$

| *Local*[*iff*]: $((LOCAL f; c; g) \# cs, s) \rightarrow (c \# Do(\lambda t. \{g s t\}) \# cs, f s)$

abbreviation

exectr :: $(com\ list \times state) \Rightarrow (com\ list \times state) \Rightarrow bool \quad (- \rightarrow^* - [81, 81] 100)$
where $cs0 \rightarrow^* cs1 \equiv (cs0, cs1) : exec1 \hat{*}$

inductive-cases *exec1E*[*elim!*]:

$([], s) \rightarrow (cs', s')$
 $(Do f \# cs, s) \rightarrow (cs', s')$
 $((c1; c2) \# cs, s) \rightarrow (cs', s')$
 $((IF b THEN c1 ELSE c2) \# cs, s) \rightarrow (cs', s')$
 $((WHILE b DO c) \# cs, s) \rightarrow (cs', s')$
 $(CALL \# cs, s) \rightarrow (cs', s')$
 $((LOCAL f; c; g) \# cs, s) \rightarrow (cs', s')$

lemma [*iff*]: $\neg ([], s) \rightarrow u$
by (*induct u*) *blast*

lemma *app-exec*: $(cs, s) \rightarrow (cs', s') \implies (cs @ cs2, s) \rightarrow (cs' @ cs2, s')$
apply (*erule exec1.induct*)
apply (*simp-all del:fun-upd-apply*)
apply (*blast intro:exec1.intros*)
done

lemma *app-execs*: $(cs, s) \rightarrow^* (cs', s') \implies (cs @ cs2, s) \rightarrow^* (cs' @ cs2, s')$
apply (*erule rtrancl-induct2*)
apply *blast*
apply (*blast intro:app-exec rtrancl-trans*)
done

lemma *exec-impl-execs*[*rule-format*]:
 $s -c \rightarrow s' \implies \forall cs. (c \# cs, s) \rightarrow^* (cs, s')$
apply (*erule exec.induct*)
apply *blast*
apply (*blast intro:rtrancl-trans*)
apply (*blast intro:exec1.IfTrue rtrancl-trans*)
apply (*blast intro:exec1.IfFalse rtrancl-trans*)

```

    apply(blast intro:exec1.WhileFalse rtrancl-trans)
    apply(blast intro:exec1.WhileTrue rtrancl-trans)
    apply(blast intro: rtrancl-trans)
  apply(blast intro: rtrancl-trans)
done

```

inductive

```

  execs :: state ⇒ com list ⇒ state ⇒ bool (-/ ==>/ - [50,0,50] 50)
  where
    s = [] ⇒ s
  | s -c→ t ⇒ t = cs ⇒ u ⇒ s = c#cs ⇒ u

```

inductive-cases [elim!]:

```

  s = [] ⇒ t
  s = c#cs ⇒ t

```

```

theorem exec1s-impl-execs: (cs,s) →* ([],t) ⇒ s = cs ⇒ t
apply(erule converse-rtrancl-induct2)
  apply(rule execs.intros)
  apply(erule exec1.cases)
  apply(blast intro:execs.intros)
  apply(blast intro:execs.intros)
  apply(fastsimp intro:execs.intros)
  apply(fastsimp intro:execs.intros)
  apply(blast intro:execs.intros exec.intros)
  apply(blast intro:execs.intros exec.intros)
  apply(blast intro:execs.intros exec.intros)
  apply(blast intro:execs.intros exec.intros)
done

```

```

theorem exec1s-impl-exec: ([c],s) →* ([],t) ⇒ s -c→ t
by(blast dest: exec1s-impl-execs)

```

consts termis :: com list ⇒ state ⇒ bool (**infixl** ↓ 60)

primrec

```

[] ↓ s = True
c#cs ↓ s = (c ↓ s ∧ (∀ t. s -c→ t → cs ↓ t))

```

lemma exec1-pres-termis: (cs,s) → (cs',s') ⇒ cs ↓ s → cs' ↓ s'

```

apply(erule exec1.induct)
  apply(simp-all)
  apply blast
  apply(blast intro:while-termiE while-termiE2 exec.WhileTrue)
apply blast
done

```

```

lemma execs-pres-termis: (cs,s) →* (cs',s') ⇒ cs ↓ s → cs' ↓ s'
apply(erule rtrancl-induct2)

```

apply *blast*
apply(*blast dest:exec1-pres-termis*)
done

lemma *execs-pres-termi*: $\llbracket ([c],s) \rightarrow^* (c'\#cs',s'); c\downarrow s \rrbracket \Longrightarrow c'\downarrow s'$
apply(*insert execs-pres-termis[of [c] - c'\#cs',simplified]*)
apply *blast*
done

constdefs
termi-call-steps :: (state × state)set
termi-call-steps ≡ {(t,s). *body*↓s ∧ (∃ cs. ([*body*], s) →* (CALL # cs, t))}

lemma *lem*:
 $!y. (a,y):r^+ \longrightarrow P a \longrightarrow P y \Longrightarrow ((b,a) : \{(y,x). P x \wedge (x,y):r^+\}^+) = ((b,a) : \{(y,x). P x \wedge (x,y):r^+\})$
apply(*rule iffI*)
apply *clarify*
apply(*erule trancl-induct*)
apply *blast*
apply(*blast intro:trancl-trans*)
apply *clarify*
apply(*erule trancl-induct*)
apply *blast*
apply(*blast intro:trancl-trans*)
done

lemma *renumber-aux*:
 $\llbracket \forall i. (a,f i) : r^* \wedge (f i, f(Suc i)) : r; (a,b) : r^* \rrbracket \Longrightarrow b = f 0 \longrightarrow (\exists f. f 0 = a \ \& \ (\forall i. (f i, f(Suc i)) : r))$
apply(*erule converse-rtrancl-induct*)
apply *blast*
apply(*clarsimp*)
apply(*rule-tac x=λi. case i of 0 ⇒ y | Suc i ⇒ fa i in exI*)
apply *simp*
apply *clarify*
apply(*case-tac i*)
apply *simp-all*
done

lemma *renumber*:
 $\forall i. (a,f i) : r^* \wedge (f i, f(Suc i)) : r \Longrightarrow \exists f. f 0 = a \ \& \ (\forall i. (f i, f(Suc i)) : r)$
by(*blast dest:renumber-aux*)

constdefs *inf* :: com list ⇒ state ⇒ bool
inf cs s ≡ ∃ f. f 0 = (cs,s) ∧ (∀ i. f i → f(Suc i))

```

lemma [iff]:  $\neg \text{inf } [] s$ 
apply(unfold inf-def)
apply clarify
apply(erule-tac x = 0 in allE)
apply simp
done

```

```

lemma [iff]:  $\neg \text{inf } [Do f] s$ 
apply(unfold inf-def)
apply clarify
apply(frule-tac x = 0 in spec)
apply(erule-tac x = 1 in allE)
apply(case-tac fa (Suc 0))
apply clarsimp
done

```

```

lemma [iff]:  $\text{inf } ((c1;c2)\#cs) s = \text{inf } (c1\#c2\#cs) s$ 
apply(unfold inf-def)
apply(rule iffI)
apply clarify
apply(rule-tac x =  $\lambda i. f(Suc i)$  in exI)
apply(frule-tac x = 0 in spec)
apply(case-tac f (Suc 0))
apply clarsimp
apply clarify
apply(rule-tac x =  $\lambda i. \text{case } i \text{ of } 0 \Rightarrow ((c1;c2)\#cs,s) \mid \text{Suc } i \Rightarrow f i$  in exI)
apply(simp split:nat.split)
done

```

```

lemma [iff]:  $\text{inf } ((IF b THEN c1 ELSE c2)\#cs) s =$ 
 $\text{inf } ((if b s then c1 else c2)\#cs) s$ 
apply(unfold inf-def)
apply(rule iffI)
apply clarsimp
apply(frule-tac x = 0 in spec)
apply (case-tac f (Suc 0))
apply(rule conjI)
apply clarsimp
apply(rule-tac x =  $\lambda i. f(Suc i)$  in exI)
apply clarsimp
apply clarsimp
apply(rule-tac x =  $\lambda i. f(Suc i)$  in exI)
apply clarsimp
apply clarsimp
apply(rule-tac x =  $\lambda i. \text{case } i \text{ of } 0 \Rightarrow ((IF b THEN c1 ELSE c2)\#cs,s) \mid \text{Suc } i \Rightarrow f i$  in exI)
apply(simp add: exec1.intros split:nat.split)
done

```

```

lemma [simp]:
  inf ((WHILE b DO c)#cs) s =
    (if b s then inf (c#(WHILE b DO c)#cs) s else inf cs s)
apply(unfold inf-def)
apply(rule iffI)
apply clarsimp
apply(frule-tac x = 0 in spec)
apply (case-tac f (Suc 0))
apply(rule conjI)
apply clarsimp
apply(rule-tac x =  $\lambda i. f(\text{Suc } i)$  in exI)
apply clarsimp
apply clarsimp
apply(rule-tac x =  $\lambda i. f(\text{Suc } i)$  in exI)
apply clarsimp
apply (clarsimp split:if-splits)
apply(rule-tac x =  $\lambda i. \text{case } i \text{ of } 0 \Rightarrow ((\text{WHILE } b \text{ DO } c)\#cs,s) \mid \text{Suc } i \Rightarrow f \ i$  in
exI)
apply(simp add: exec1.intros split:nat.split)
apply(rule-tac x =  $\lambda i. \text{case } i \text{ of } 0 \Rightarrow ((\text{WHILE } b \text{ DO } c)\#cs,s) \mid \text{Suc } i \Rightarrow f \ i$  in
exI)
apply(simp add: exec1.intros split:nat.split)
done

```

```

lemma [iff]: inf (CALL#cs) s = inf (body#cs) s
apply(unfold inf-def)
apply(rule iffI)
apply clarsimp
apply(frule-tac x = 0 in spec)
apply (case-tac f (Suc 0))
apply clarsimp
apply(rule-tac x =  $\lambda i. f(\text{Suc } i)$  in exI)
apply clarsimp
apply clarsimp
apply(rule-tac x =  $\lambda i. \text{case } i \text{ of } 0 \Rightarrow (\text{CALL}\#cs,s) \mid \text{Suc } i \Rightarrow f \ i$  in exI)
apply(simp add: exec1.intros split:nat.split)
done

```

```

lemma [iff]: inf ((LOCAL f;c;g)#cs) s =
  inf (c#Do( $\lambda t. \{g \ s \ t\}$ )#cs) (f s)
apply(unfold inf-def)
apply(rule iffI)
apply clarsimp
apply(rename-tac F)
apply(frule-tac x = 0 in spec)
apply (case-tac F (Suc 0))
apply clarsimp
apply(rule-tac x =  $\lambda i. F(\text{Suc } i)$  in exI)
apply clarsimp

```

apply (*clarsimp*)
apply(*rename-tac F*)
apply(*rule-tac x = λi. case i of 0 ⇒ ((LOCAL f;c;g)#cs,s) | Suc i ⇒ F i in exI*)
apply(*simp add: exec1.intros split:nat.split*)
done

lemma *exec1-only1-aux*: $(ccs,s) \rightarrow (cs',t) \implies$
 $\forall c cs. ccs = c\#cs \longrightarrow (\exists cs1. cs' = cs1 @ cs)$
apply(*erule exec1.induct*)
apply *blast*
apply *force+*
done

lemma *exec1-only1*: $(c\#cs,s) \rightarrow (cs',t) \implies \exists cs1. cs' = cs1 @ cs$
by(*blast dest:exec1-only1-aux*)

lemma *exec1-drop-suffix-aux*:
 $(cs12,s) \rightarrow (cs1'2,s') \implies !cs1 cs2 cs1'$
 $cs12 = cs1@cs2 \ \& \ cs1'2 = cs1'@cs2 \ \& \ cs1 \neq [] \longrightarrow (cs1,s) \rightarrow (cs1',s')$
apply(*erule exec1.induct*)
apply (*force intro:exec1.intros simp add: neq-Nil-conv*)
done

lemma *exec1-drop-suffix*:
 $(cs1@cs2,s) \rightarrow (cs1'@cs2,s') \implies cs1 \neq [] \implies (cs1,s) \rightarrow (cs1',s')$
by(*blast dest:exec1-drop-suffix-aux*)

lemma *execs-drop-suffix*[*rule-format(no-asm)*]:
 $\llbracket f\ 0 = (c\#cs,s); !i. f(i) \rightarrow f(Suc\ i) \rrbracket \implies$
 $(!i < k. p\ i \neq [] \ \& \ fst(f\ i) = p\ i@cs) \longrightarrow fst(f\ k) = p\ k@cs$
 $\longrightarrow ([c],s) \rightarrow^* (p\ k, snd(f\ k))$
apply(*induct-tac k*)
apply *simp*
apply (*clarsimp*)
apply(*erule rtrancl-into-rtrancl*)
apply(*erule-tac x = n in allE*)
apply(*erule-tac x = n in allE*)
apply(*case-tac f n*)
apply(*case-tac f(Suc n)*)
apply *simp*
apply(*blast dest:exec1-drop-suffix*)
done

lemma *execs-drop-suffix0*:
 $\llbracket f\ 0 = (c\#cs,s); !i. f(i) \rightarrow f(Suc\ i); !i < k. p\ i \neq [] \ \& \ fst(f\ i) = p\ i@cs;$
 $fst(f\ k) = cs; p\ k = [] \rrbracket \implies ([c],s) \rightarrow^* ([], snd(f\ k))$
apply(*erule execs-drop-suffix, assumption, assumption*)
apply *simp*

apply simp
done

lemma skolemize1: $\forall x. P x \longrightarrow (\exists y. Q x y) \Longrightarrow \exists f. \forall x. P x \longrightarrow Q x (f x)$
apply(rule-tac $x = \lambda x. \text{SOME } y. Q x y$ **in** exI)
apply(fast intro:someI2)
done

lemma least-aux: $\llbracket f 0 = (c \# cs, s); \forall i. f i \rightarrow f (Suc i);$
 $fst(f k) = cs; !i < k. fst(f i) \neq cs \rrbracket$
 $\Longrightarrow \forall i \leq k. (\exists p. (p \neq []) = (i < k) \ \& \ fst(f i) = p @ cs)$

apply(rule allI)
apply(induct-tac i)
apply simp
apply (rule ccontr)
apply simp
apply clarsimp
apply(drule order-le-imp-less-or-eq)
apply(erule disjE)
prefer 2
apply simp
apply simp
apply(erule-tac $x = n$ **in** allE)
apply(erule-tac $x = Suc n$ **in** allE)
apply(case-tac f n)
apply(case-tac f(Suc n))
apply simp
apply(rename-tac sn csn1 sn1)
apply (clarsimp simp add: neq-Nil-conv)
apply(drule exec1-only1)
apply (clarsimp simp add: neq-Nil-conv)
apply(erule disjE)
apply clarsimp
apply clarsimp
apply(case-tac cs1)
apply simp
apply simp
done

lemma least-lem: $\llbracket f 0 = (c \# cs, s); !i. f i \rightarrow f(Suc i); EX i. fst(f i) = cs \rrbracket$
 $\Longrightarrow EX k. fst(f k) = cs \ \& \ ([c], s) \rightarrow^* ([], snd(f k))$
apply(rule-tac $x = \text{LEAST } i. fst(f i) = cs$ **in** exI)
apply(rule conjI)
apply(fast intro: LeastI)
apply(subgoal-tac
 $!i <= \text{LEAST } i. fst(f i) = cs. EX p. ((p \neq []) = (i < (\text{LEAST } i. fst(f i) = cs)))$
 $\ \& \ fst(f i) = p @ cs$)
apply(drule skolemize1)
apply clarify

```

apply(rename-tac p)
apply(erule-tac p=p in execs-drop-suffix0, assumption)
  apply (blast dest:order-less-imp-le)
  apply(fast intro: LeastI)
apply(erule thin-rl)
apply(erule-tac x = LEAST j. fst (f j) = fst (f i) in allE)
apply blast
apply(erule least-aux, assumption)
  apply(fast intro: LeastI)
apply clarify
apply(drule not-less-Least)
apply blast
done

```

```

lemma skolemize2:  $\forall x. \exists y. P x y \implies \exists f. \forall x. P x (f x)$ 
apply(rule-tac x =  $\lambda x. \text{SOME } y. P x y$  in exI)
apply(fast intro:someI2)
done

```

```

lemma inf-cases:  $\text{inf } (c\#cs) s \implies \text{inf } [c] s \vee (\exists t. s -c \rightarrow t \wedge \text{inf } cs t)$ 
apply(unfold inf-def)
apply (clarsimp del: disjCI)
apply(case-tac EX i. fst(f i) = cs)
  apply(rule disjI2)
  apply(drule least-lem, assumption, assumption)
  apply clarify
  apply(drule exec1s-impl-exec)
  apply(case-tac f k)
  apply simp
  apply (rule exI, rule conjI, assumption)
  apply(rule-tac x= $\lambda i. f(i+k)$  in exI)
  apply (clarsimp)
apply(rule disjI1)
apply simp
apply(subgoal-tac  $\forall i. \exists p. p \neq [] \wedge \text{fst}(f i) = p@cs$ )
  apply(drule skolemize2)
  apply clarify
  apply(rename-tac p)
  apply(rule-tac x =  $\lambda i. (p i, \text{snd}(f i))$  in exI)
  apply(rule conjI)
  apply(erule-tac x = 0 in allE, erule conjE)
  apply simp
  apply clarify
  apply(erule-tac x = i in allE)
  apply(erule-tac x = i in allE)
  apply(frule-tac x = i in spec)
  apply(erule-tac x = Suc i in allE)
  apply(case-tac f i)
  apply(case-tac f (Suc i))

```

```

apply clarsimp
apply(blast intro:exec1-drop-suffix)
apply(clarify)
apply(induct-tac i)
  apply force
apply clarsimp
apply(case-tac p)
  apply blast
apply(erule-tac x=n in allE)
apply(erule-tac x=Suc n in allE)
apply(case-tac f n)
apply(case-tac f (Suc n))
apply clarsimp
apply(drule exec1-only1)
apply clarsimp
done

```

lemma *termi-impl-not-inf*: $c \downarrow s \implies \neg \text{inf } [c] s$
apply(*erule termi.induct*)

```

  apply clarify

```

```

  apply(blast dest:inf-cases)

```

```

  apply clarsimp
apply clarsimp

```

```

  apply clarsimp
apply(fastsimp dest:inf-cases)

```

```

apply blast

```

```

apply(blast dest:inf-cases)
done

```

lemma *termi-impl-no-inf-chain*:

```

   $c \downarrow s \implies \neg (\exists f. f 0 = ([c], s) \wedge (\forall i::\text{nat}. (f i, f(i+1)) : \text{exec1}^{\wedge+}))$ 
apply(subgoal-tac wf(\{(y,x). ([c],s) \to^* x \& x \to y\}^{\wedge+}))
apply(simp only:wf-iff-no-infinite-down-chain)
apply(erule contrapos-nn)
apply clarify
apply(subgoal-tac !i. ([c], s) \to^* f i)
prefer 2
apply(rule allI)
apply(induct-tac i)
  apply simp
apply simp
apply(blast intro: trancl-into-rtrancl rtrancl-trans)
apply(rule-tac x=f in exI)

```

```

apply clarify
apply(drule-tac  $x=i$  in spec)
apply(subst lem)
apply(blast intro: trancl-into-rtrancl rtrancl-trans)
apply clarsimp
apply(rule wf-trancl)
apply(simp only:wf-iff-no-infinite-down-chain)
apply(clarify)
apply simp
apply(drule renumber)
apply(fold inf-def)
apply(simp add: termi-impl-not-inf)
done

```

```

consts cseq :: (nat  $\Rightarrow$  state)  $\Rightarrow$  nat  $\Rightarrow$  com list
primrec
cseq S 0 = []
cseq S (Suc i) = (SOME cs. ([body], S i)  $\rightarrow^*$  (CALL # cs, S(i+1))) @ cseq S i

```

```

lemma wf-termi-call-steps: wf termi-call-steps
apply(unfold termi-call-steps-def)
apply(simp only:wf-iff-no-infinite-down-chain)
apply(clarify)
apply(rename-tac S)
apply simp
apply(subgoal-tac  $\exists$  Cs. Cs 0 = [] & ( $\forall$  i. (body # Cs i, S i)  $\rightarrow^*$  (CALL # Cs(i+1), S(i+1))))
prefer 2
apply(rule-tac  $x =$  cseq S in exI)
apply clarsimp
apply(erule-tac  $x=i$  in allE)
apply(clarify)
apply(erule-tac  $P =$   $\lambda$ cs. ([body], S i)  $\rightarrow^*$  (CALL # cs, S(Suc i)) in someI2)
apply(fastsimp dest:app-execs)
apply clarify
apply(subgoal-tac  $\forall$  i. ((body # Cs i, S i), (body # Cs(i+1), S(i+1))) : exec1  $\hat{+}$ )
prefer 2
apply(blast intro:rtrancl-into-trancl1)
apply(subgoal-tac  $\exists$  f. f 0 = ([body], S 0)  $\wedge$  ( $\forall$  i. (f i, f(i+1)) : exec1  $\hat{+}$ ))
prefer 2
apply(rule-tac  $x =$   $\lambda$ i. (body # Cs i, S i) in exI)
apply blast
apply(blast dest:termi-impl-no-inf-chain)
done

```

```

lemma CALL-lemma:
 $\{(\lambda z s. (z=s \wedge \text{body} \downarrow s) \wedge (s,t) \in \text{termi-call-steps}, \text{CALL}, \lambda z s. z - \text{body} \rightarrow s)\} \vdash_t$ 
 $\{\lambda z s. (z=s \wedge \text{body} \downarrow t) \wedge (\exists cs. ([\text{body}], t) \rightarrow^* (c \# cs, s))\} c \{\lambda z s. z - c \rightarrow s\}$ 
apply(induct-tac c)

```

```

apply (rule strengthen-pre[OF - thoare.Do])
apply(blast dest: execs-pres-termi)

apply(rename-tac c1 c2)
apply(rule-tac Q =  $\lambda z s. \text{body} \downarrow t \ \& \ (\text{EX } cs. ([\text{body}], t) \rightarrow^* (c2 \# cs, s)) \ \& \ z$ 
- c1  $\rightarrow$  s & c2  $\downarrow$  s in thoare.Semi)
apply(erule thoare.Conseq)
apply(rule conjI)
apply clarsimp
apply(subgoal-tac s -c1  $\rightarrow$  ta)
prefer 2
apply(blast intro: exec1.Semi exec-impl-execs rtrancl-trans)
apply(subgoal-tac ([body], t)  $\rightarrow^*$  (c2 # cs, ta))
prefer 2
apply(blast intro:exec1.Semi[THEN r-into-rtrancl] exec-impl-execs rtrancl-trans)
apply(subgoal-tac ([body], t)  $\rightarrow^*$  (c2 # cs, ta))
prefer 2
apply(blast intro: exec-impl-execs rtrancl-trans)
apply(blast intro:exec-impl-execs rtrancl-trans execs-pres-termi)
apply(fast intro: exec1.Semi rtrancl-trans)
apply(erule thoare.Conseq)
apply blast

prefer 3
apply(simp only:termi-call-steps-def)
apply(rule thoare.Conseq[OF thoare.Asm])
apply(blast dest: execs-pres-termi)

apply(rule thoare.If)
apply(erule thoare.Conseq)
apply simp
apply(blast intro: exec1.IfTrue rtrancl-trans)
apply(erule thoare.Conseq)
apply simp
apply(blast intro: exec1.IfFalse rtrancl-trans)

defer
apply simp
apply(rule thoare.Local)
apply(rule allI)
apply(erule thoare.Conseq)
apply (clarsimp)
apply(rule conjI)
apply (clarsimp)
apply(drule rtrancl-trans[OF - r-into-rtrancl[OF exec1.Local]])
apply(fast)
apply (clarsimp)
apply(drule rtrancl-trans[OF - r-into-rtrancl[OF exec1.Local]])

```

```

apply blast
apply(rename-tac b c)
apply(rule-tac  $P' = \lambda z s. (z, s) \in (\{(s, t). b s \wedge s - c \rightarrow t\})^* \wedge \text{body} \downarrow t \wedge$ 
 $(\exists cs. ([\text{body}], t) \rightarrow^* ((\text{WHILE } b \text{ DO } c) \# cs, s))$  in thoare.Conseq)
apply(rule-tac thoare.While[OF wf-termi])
apply(rule allI)
apply(erule thoare.Conseq)
apply clarsimp
apply(rule conjI)
apply clarsimp
apply(rule conjI)
apply(blast intro: rtrancl-trans exec1.WhileTrue)
apply(rule conjI)
apply(rule exI, rule rtrancl-trans, assumption)
apply(blast intro: exec1.WhileTrue exec-impl-execs rtrancl-trans)
apply(rule conjI)
apply(blast intro:execs-pres-termi)
apply(blast intro: exec1.WhileTrue exec-impl-execs rtrancl-trans)
apply(blast intro: exec1.WhileTrue exec-impl-execs rtrancl-trans)
apply(rule conjI)
apply clarsimp
apply(erule-tac  $x = s$  in allE)
apply clarsimp
apply(erule impE)
apply blast
apply clarify
apply(erule-tac  $a=s$  in converse-rtrancl-induct)
apply simp
apply(fast elim:exec.WhileTrue)
apply(fast intro: rtrancl-refl)
done

```

lemma *CALL-cor*:

```

 $\{(\lambda z s. (z=s \wedge \text{body} \downarrow s) \wedge (s, t) \in \text{termi-call-steps}, \text{CALL}, \lambda z s. z - \text{body} \rightarrow s)\} \vdash_t$ 
 $\{\lambda z s. (z=s \wedge \text{body} \downarrow s) \wedge s = t\} \text{body} \{\lambda z s. z - \text{body} \rightarrow s\}$ 
apply(rule strengthen-pre[OF - CALL-lemma])
apply blast
done

```

lemma *MGT-CALL*: $\{\} \vdash_t \text{MGT}_t \text{CALL}$

```

apply(simp add: MGTt-def)
apply(blast intro:thoare.Call wf-termi-call-steps CALL-cor)
done

```

theorem $\{\} \models_t \{P\}c\{Q\} \implies \{\} \vdash_t \{P\}c\{Q::\text{state assn}\}$

```

apply(erule MGT-implies-complete[OF MGT-lemma[OF MGT-CALL]])
done

```

end

4 Hoare Logics for Mutually Recursive Procedure

theory *PsLang* imports *Main* begin

4.1 The language

typedecl *state*

typedecl *pname*

types *bexp* = *state* \Rightarrow *bool*

datatype

com = *Do state* \Rightarrow *state set*
| *Semi com com* (*-;* - [60, 60] 10)
| *Cond bexp com com* (*IF - THEN - ELSE -* 60)
| *While bexp com* (*WHILE - DO -* 60)
| *CALL pname*
| *Local (state* \Rightarrow *state) com (state* \Rightarrow *state* \Rightarrow *state)*
(*LOCAL -;* -; - [0,0,60] 60)

consts *body* :: *pname* \Rightarrow *com*

We generalize from a single procedure to a whole set of procedures following the ideas of von Oheimb [3]. The basic setup is modified only in a few places:

- We introduce a new basic type *pname* of procedure names.
- Constant *body* is now of type *pname* \Rightarrow *com*.
- The *CALL* command now has an argument of type *pname*, the name of the procedure that is to be called.

inductive

exec :: *state* \Rightarrow *com* \Rightarrow *state* \Rightarrow *bool* (*-/ -->/* - [50,0,50] 50)

where

Do: $t \in f s \Longrightarrow s -Do f \rightarrow t$

| *Semi*: $\llbracket s0 -c1 \rightarrow s1; s1 -c2 \rightarrow s2 \rrbracket \Longrightarrow s0 -c1;c2 \rightarrow s2$

| *IfTrue*: $\llbracket b s; s -c1 \rightarrow t \rrbracket \Longrightarrow s -IF b THEN c1 ELSE c2 \rightarrow t$

| *IfFalse*: $\llbracket \neg b s; s -c2 \rightarrow t \rrbracket \Longrightarrow s -IF b THEN c1 ELSE c2 \rightarrow t$

| *WhileFalse*: $\neg b s \Longrightarrow s -WHILE b DO c \rightarrow s$

| *WhileTrue*: $\llbracket b s; s -c \rightarrow t; t -WHILE b DO c \rightarrow u \rrbracket$

$\implies s - \text{WHILE } b \text{ DO } c \rightarrow u$

| *Call*: $s - \text{body } p \rightarrow t \implies s - \text{CALL } p \rightarrow t$

| *Local*: $f s - c \rightarrow t \implies s - \text{LOCAL } f; c; g \rightarrow g s t$

lemma [*iff*]: $(s - \text{Do } f \rightarrow t) = (t \in f s)$
by(*auto elim: exec.cases intro:exec.intros*)

lemma [*iff*]: $(s - c; d \rightarrow u) = (\exists t. s - c \rightarrow t \wedge t - d \rightarrow u)$
by(*auto elim: exec.cases intro:exec.intros*)

lemma [*iff*]: $(s - \text{IF } b \text{ THEN } c \text{ ELSE } d \rightarrow t) =$
 $(s - \text{if } b \text{ s then } c \text{ else } d \rightarrow t)$

apply(*rule iffI*)
apply(*auto elim: exec.cases intro:exec.intros*)
apply(*auto intro:exec.intros split:split-if-asm*)
done

lemma [*iff*]: $(s - \text{CALL } p \rightarrow t) = (s - \text{body } p \rightarrow t)$
by(*blast elim: exec.cases intro:exec.intros*)

lemma [*iff*]: $(s - \text{LOCAL } f; c; g \rightarrow u) = (\exists t. f s - c \rightarrow t \wedge u = g s t)$
by(*fastsimp elim: exec.cases intro:exec.intros*)

inductive

execn :: *state* \Rightarrow *com* \Rightarrow *nat* \Rightarrow *state* \Rightarrow *bool* (-/ \dashrightarrow / - [50,0,0,50] 50)

where

Do: $t \in f s \implies s - \text{Do } f - n \rightarrow t$

| *Semi*: $\llbracket s0 - c0 - n \rightarrow s1; s1 - c1 - n \rightarrow s2 \rrbracket \implies s0 - c0; c1 - n \rightarrow s2$

| *IfTrue*: $\llbracket b s; s - c0 - n \rightarrow t \rrbracket \implies s - \text{IF } b \text{ THEN } c0 \text{ ELSE } c1 - n \rightarrow t$

| *IfFalse*: $\llbracket \neg b s; s - c1 - n \rightarrow t \rrbracket \implies s - \text{IF } b \text{ THEN } c0 \text{ ELSE } c1 - n \rightarrow t$

| *WhileFalse*: $\neg b s \implies s - \text{WHILE } b \text{ DO } c - n \rightarrow s$

| *WhileTrue*: $\llbracket b s; s - c - n \rightarrow t; t - \text{WHILE } b \text{ DO } c - n \rightarrow u \rrbracket$
 $\implies s - \text{WHILE } b \text{ DO } c - n \rightarrow u$

| *Call*: $s - \text{body } p - n \rightarrow t \implies s - \text{CALL } p - \text{Suc } n \rightarrow t$

| *Local*: $f s - c - n \rightarrow t \implies s - \text{LOCAL } f; c; g - n \rightarrow g s t$

lemma [*iff*]: $(s - \text{Do } f - n \rightarrow t) = (t \in f s)$
by(*auto elim: execn.cases intro:execn.intros*)

lemma [*iff*]: $(s - c1; c2 - n \rightarrow u) = (\exists t. s - c1 - n \rightarrow t \wedge t - c2 - n \rightarrow u)$

```

by(best elim: execn.cases intro:execn.intros)

lemma [iff]: (s -IF b THEN c ELSE d-n→ t) =
  (s -if b s then c else d-n→ t)
apply auto
apply(blast elim: execn.cases intro:execn.intros)+
done

lemma [iff]: (s -CALL p- 0→ t) = False
by(blast elim: execn.cases intro:execn.intros)

lemma [iff]: (s -CALL p-Suc n→ t) = (s -body p-n→ t)
by(blast elim: execn.cases intro:execn.intros)

lemma [iff]: (s -LOCAL f; c; g-n→ u) = (∃ t. f s -c-n→ t ∧ u = g s t)
by(auto elim: execn.cases intro:execn.intros)

lemma exec-mono[rule-format]: s -c-m→ t ⇒ ∀ n. m ≤ n → s -c-n→ t
apply(erule execn.induct)
  apply(blast)
  apply(blast)
  apply(simp)
  apply(simp)
  apply(simp add:execn.intros)
  apply(blast intro:execn.intros)
  apply(clarify)
  apply(rename-tac m)
  apply(case-tac m)
  apply simp
  apply simp
  apply blast
done

lemma exec-iff-execn: (s -c→ t) = (∃ n. s -c-n→ t)
apply(rule iffI)
  apply(erule exec.induct)
    apply blast
    apply clarify
    apply(rename-tac m n)
    apply(rule-tac x = max m n in exI)
    apply(fastsimp intro:exec.intros exec-mono simp add:max-def)
    apply fastsimp
    apply fastsimp
    apply(blast intro:execn.intros)
  apply clarify
  apply(rename-tac m n)
  apply(rule-tac x = max m n in exI)
  apply(fastsimp elim:execn.WhileTrue exec-mono simp add:max-def)

```

```

apply blast
apply blast
apply(erule exE, erule execn.induct)
  apply blast
  apply blast
  apply fastsimp
  apply fastsimp
  apply(erule exec.WhileFalse)
  apply(blast intro: exec.intros)
apply blast
apply blast
done

```

```

lemma while-lemma[rule-format]:
   $s -w-n \rightarrow t \implies !b\ c.\ w = \text{WHILE } b\ \text{DO } c \wedge P\ s \wedge$ 
   $(!s\ s'.\ P\ s \wedge b\ s \wedge s -c-n \rightarrow s' \longrightarrow P\ s') \longrightarrow P\ t \wedge \neg b\ t$ 
apply(erule execn.induct)
apply clarify+
defer
apply clarify+
apply(subgoal-tac P t)
apply blast
apply blast
done

```

```

lemma while-rule:
   $\llbracket s - \text{WHILE } b\ \text{DO } c -n \rightarrow t; P\ s; \bigwedge s\ s'.\ \llbracket P\ s; b\ s; s -c-n \rightarrow s' \rrbracket \implies P\ s \rrbracket$ 
   $\implies P\ t \wedge \neg b\ t$ 
apply(erule while-lemma)
prefer 2 apply assumption
apply blast
done

```

end

theory *PsHoare* **imports** *PsLang* **begin**

4.2 Hoare logic for partial correctness

```

types 'a assn = 'a  $\Rightarrow$  state  $\Rightarrow$  bool
  'a cntxt = ('a assn  $\times$  com  $\times$  'a assn)set

```

```

constdefs
  valid :: 'a assn  $\Rightarrow$  com  $\Rightarrow$  'a assn  $\Rightarrow$  bool ( $\models$   $\{(1-)\}/ (-)/ \{(1-)\}$  50)
   $\models \{P\}c\{Q\} \equiv \forall s\ t\ z.\ s -c \rightarrow t \longrightarrow P\ z\ s \longrightarrow Q\ z\ t$ 

  valids :: 'a cntxt  $\Rightarrow$  bool ( $\models$  - 50)

```

$$\models D \equiv \forall (P, c, Q) \in D. \models \{P\}c\{Q\}$$

$$\begin{aligned} nvalid &:: nat \Rightarrow 'a\ assn \Rightarrow com \Rightarrow 'a\ assn \Rightarrow bool (\models - \{(1-\)/ (-)/ \{(1-\)\} 50) \\ \models_n \{P\}c\{Q\} &\equiv \forall s\ t\ z. s -c-n \rightarrow t \longrightarrow P\ z\ s \longrightarrow Q\ z\ t \end{aligned}$$

$$\begin{aligned} nvalids &:: nat \Rightarrow 'a\ cntxt \Rightarrow bool (\models' - / - 50) \\ \models_n C &\equiv \forall (P, c, Q) \in C. \models_n \{P\}c\{Q\} \end{aligned}$$

We now need an additional notion of validity $C \models D$ where D is a set as well. The reason is that we can now have mutually recursive procedures whose correctness needs to be established by simultaneous induction. Instead of sets of Hoare triples we may think of conjunctions. We define both $C \models D$ and its relativized version:

constdefs

$$\begin{aligned} cvalids &:: 'a\ cntxt \Rightarrow 'a\ cntxt \Rightarrow bool (- \models / - 50) \\ C \models D &\equiv \models C \longrightarrow \models D \end{aligned}$$

$$\begin{aligned} cvalids &:: 'a\ cntxt \Rightarrow nat \Rightarrow 'a\ cntxt \Rightarrow bool (- \models' - / - 50) \\ C \models_n D &\equiv \models_n C \longrightarrow \models_n D \end{aligned}$$

Our Hoare logic now defines judgements of the form $C \Vdash D$ where both C and D are (potentially infinite) sets of Hoare triples; $C \vdash \{P\}c\{Q\}$ is simply an abbreviation for $C \Vdash \{(P, c, Q)\}$.

inductive

$$\begin{aligned} hoare &:: 'a\ cntxt \Rightarrow 'a\ cntxt \Rightarrow bool (- \Vdash / - 50) \\ \text{and } hoare3 &:: 'a\ cntxt \Rightarrow 'a\ assn \Rightarrow com \Rightarrow 'a\ assn \Rightarrow bool (- \vdash / (\{(1-\)/ (-)/ \{(1-\)\} 50) \end{aligned}$$

where

$$\begin{aligned} C \vdash \{P\}c\{Q\} &\equiv C \Vdash \{(P, c, Q)\} \\ | Do: & C \vdash \{\lambda z\ s. \forall t \in f\ s. P\ z\ t\} Do\ f\ \{P\} \\ | Semi: & [C \vdash \{P\}c\{Q\}; C \vdash \{Q\}d\{R\}] \Longrightarrow C \vdash \{P\}c;d\ \{R\} \\ | If: & [C \vdash \{\lambda z\ s. P\ z\ s \wedge b\ s\}c\{Q\}; C \vdash \{\lambda z\ s. P\ z\ s \wedge \neg b\ s\}d\{Q\}] \Longrightarrow \\ & C \vdash \{P\} IF\ b\ THEN\ c\ ELSE\ d\ \{Q\} \\ | While: & C \vdash \{\lambda z\ s. P\ z\ s \wedge b\ s\} c\ \{P\} \Longrightarrow \\ & C \vdash \{P\} WHILE\ b\ DO\ c\ \{\lambda z\ s. P\ z\ s \wedge \neg b\ s\} \\ | Conseq: & [C \vdash \{P'\}c\{Q'\}; \\ & \forall s\ t. (\forall z. P'\ z\ s \longrightarrow Q'\ z\ t) \longrightarrow (\forall z. P\ z\ s \longrightarrow Q\ z\ t)] \Longrightarrow \\ & C \vdash \{P\}c\{Q\} \\ | Call: & [\forall (P, c, Q) \in C. \exists p. c = CALL\ p; \\ & C \Vdash \{(P, b, Q). \exists p. (P, CALL\ p, Q) \in C \wedge b = body\ p\}] \\ & \Longrightarrow \{ \} \Vdash C \\ | Asm: & (P, CALL\ p, Q) \in C \Longrightarrow C \vdash \{P\} CALL\ p\ \{Q\} \\ | ConjI: & \forall (P, c, Q) \in D. C \vdash \{P\}c\{Q\} \Longrightarrow C \Vdash D \\ | ConjE: & [C \Vdash D; (P, c, Q) \in D] \Longrightarrow C \vdash \{P\}c\{Q\} \end{aligned}$$

$\llbracket \forall z s. P' z s \longrightarrow P z s; C \vdash \{P\}c\{Q\} \rrbracket \Longrightarrow C \vdash \{P'\}c\{Q\}$
by(rule hoare.Conseq, assumption, blast)

lemma *MGT-implies-complete*:

$\{\} \Vdash \{MGT\ c\} \Longrightarrow \models \{P\}c\{Q\} \Longrightarrow \{\} \vdash \{P\}c\{Q::state\ assn\}$
apply(unfold MGT-def)
apply (erule hoare.Conseq)
apply(simp add: valid-defs)
done

lemma *MGT-lemma*: $\forall p. C \Vdash \{MGT(CALL\ p)\} \Longrightarrow C \Vdash \{MGT\ c\}$

apply (simp)
apply(induct-tac c)
apply (rule strengthen-pre[OF - hoare.Do])
apply blast
apply simp
apply (rule hoare.Semi)
apply blast
apply (rule hoare.Conseq)
apply blast
apply blast
apply clarsimp
apply(rule hoare.If)
apply(rule hoare.Conseq)
apply blast
apply simp
apply(rule hoare.Conseq)
apply blast
apply simp
prefer 2
apply simp
apply(rename-tac b c)
apply(rule hoare.Conseq)
apply(rule-tac $P = \lambda z s. (z,s) \in (\{(s,t). b\ s \wedge s -c \rightarrow t\})^*$
in hoare.While)
apply(erule hoare.Conseq)
apply(blast intro:rtrancl-into-rtrancl)
apply clarsimp
apply(rename-tac s t)
apply(erule-tac $x = s$ **in** allE)
apply clarsimp
apply(erule converse-rtrancl-induct)
apply(blast intro:exec.intros)
apply(fast elim:exec.WhileTrue)

apply(fastsimp intro: hoare.Local elim!: hoare.Conseq)
done

lemma *MGT-body*: $(P, CALL\ p, Q) = MGT\ (CALL\ pa) \Longrightarrow C \Vdash \{MGT\ (body)$

```

p)}  $\implies C \vdash \{P\} \text{ body } p \{Q\}$ 
apply clarsimp
done

declare MGT-def[simp del]

lemma MGT-CALL:  $\{\} \Vdash \{mgt. \exists p. mgt = MGT(CALL\ p)\}$ 
apply (rule hoare.Call)
  apply(fastsimp simp add:MGT-def)
apply(rule hoare.ConjI)
apply clarsimp
apply (erule MGT-body)
apply(rule MGT-lemma)
apply(unfold MGT-def)
apply(fast intro: hoare.Asm)
done

theorem Complete:  $\models \{P\}c\{Q\} \implies \{\} \vdash \{P\}c\{Q::state\ assn\}$ 
apply(rule MGT-implies-complete)
prefer 2
apply assumption
apply (rule MGT-lemma)
apply(rule allI)
apply(unfold MGT-def)
apply(rule hoare.ConjE[OF MGT-CALL])
apply(simp add:MGT-def)
done

end

```

```

theory PsTermi imports PsLang begin

```

4.3 Termination

```

inductive

```

```

  termi :: com  $\Rightarrow$  state  $\Rightarrow$  bool (infixl  $\downarrow$  50)

```

```

where

```

```

  Do[iff]:  $f\ s \neq \{\} \implies Do\ f \downarrow s$ 
  | Semi[intro!]:  $\llbracket c1 \downarrow s0; \bigwedge s1. s0 -c1 \rightarrow s1 \implies c2 \downarrow s1 \rrbracket$ 
     $\implies (c1;c2) \downarrow s0$ 

  | IfTrue[intro,simp]:  $\llbracket b\ s; c1 \downarrow s \rrbracket \implies IF\ b\ THEN\ c1\ ELSE\ c2 \downarrow s$ 
  | IfFalse[intro,simp]:  $\llbracket \neg b\ s; c2 \downarrow s \rrbracket \implies IF\ b\ THEN\ c1\ ELSE\ c2 \downarrow s$ 

  | WhileFalse:  $\neg b\ s \implies WHILE\ b\ DO\ c \downarrow s$ 

  | WhileTrue:  $\llbracket b\ s; c \downarrow s; \bigwedge t. s -c \rightarrow t \implies WHILE\ b\ DO\ c \downarrow t \rrbracket$ 
     $\implies WHILE\ b\ DO\ c \downarrow s$ 

```

| *body* $p \downarrow s \implies CALL\ p \downarrow s$

| *Local*: $c \downarrow f\ s \implies LOCAL\ f;c;g \downarrow s$

lemma [*iff*]: $(Do\ f \downarrow s) = (f\ s \neq \{\})$
apply(*rule iffI*)
prefer 2
apply(*best intro:termi.intros*)
apply(*erule termi.cases*)
apply *blast+*
done

lemma [*iff*]: $((c1;c2) \downarrow s0) = (c1 \downarrow s0 \wedge (\forall s1. s0 -c1 \rightarrow s1 \longrightarrow c2 \downarrow s1))$
apply(*rule iffI*)
prefer 2
apply(*best intro:termi.intros*)
apply(*erule termi.cases*)
apply *blast+*
done

lemma [*iff*]: $(IF\ b\ THEN\ c1\ ELSE\ c2 \downarrow s) =$
 $((if\ b\ s\ then\ c1\ else\ c2) \downarrow s)$
apply *simp*
apply(*rule conjI*)
apply(*rule impI*)
apply(*rule iffI*)
prefer 2
apply(*blast intro:termi.intros*)
apply(*erule termi.cases*)
apply *blast+*
apply(*rule impI*)
apply(*rule iffI*)
prefer 2
apply(*blast intro:termi.intros*)
apply(*erule termi.cases*)
apply *blast+*
done

lemma [*iff*]: $(CALL\ p \downarrow s) = (body\ p \downarrow s)$
by(*fast elim: termi.cases intro:termi.intros*)

lemma [*iff*]: $(LOCAL\ f;c;g \downarrow s) = (c \downarrow f\ s)$
by(*fast elim: termi.cases intro:termi.intros*)

lemma *termi-while-lemma*[*rule-format*]:
 $w \downarrow fk \implies$
 $(\forall k\ b\ c. fk = f\ k \wedge w = WHILE\ b\ DO\ c \wedge (\forall i. f\ i -c \rightarrow f(Suc\ i))$
 $\longrightarrow (\exists i. \neg b(f\ i)))$

apply(*erule termi.induct*)
apply *simp-all*
apply *blast*
apply *blast*
done

lemma *termi-while*:

$\llbracket (WHILE\ b\ DO\ c) \downarrow f\ k; \forall i. f\ i \rightarrow c \rightarrow f(Suc\ i) \rrbracket \implies \exists i. \neg b(f\ i)$
by(*blast intro:termi-while-lemma*)

lemma *wf-termi*: *wf* $\{(t,s). WHILE\ b\ DO\ c \downarrow s \wedge b\ s \wedge s \rightarrow t\}$

apply(*subst wf-iff-no-infinite-down-chain*)
apply(*rule notI*)
apply *clarsimp*
apply(*insert termi-while*)
apply *blast*
done

end

theory *PsHoareTotal* **imports** *PsHoare PsTermi* **begin**

4.4 Hoare logic for total correctness

constdefs

tvalid :: 'a *assn* \Rightarrow *com* \Rightarrow 'a *assn* \Rightarrow *bool* ($\models_t \{(1-)\} / (-) / \{(1-)\} 50$)
 $\models_t \{P\}c\{Q\} \equiv \models \{P\}c\{Q\} \wedge (\forall z\ s. P\ z\ s \longrightarrow c\downarrow s)$

valids :: 'a *cntxt* \Rightarrow *bool* ($\models_t - 50$)
 $\models_t D \equiv \forall (P,c,Q) \in D. \models_t \{P\}c\{Q\}$

ctvalid :: 'a *cntxt* \Rightarrow 'a *assn* \Rightarrow *com* \Rightarrow 'a *assn* \Rightarrow *bool*
 $((- / \models_t \{(1-)\} / (-) / \{(1-)\}) 50)$
 $C \models_t \{P\}c\{Q\} \equiv \models_t C \longrightarrow \models_t \{P\}c\{Q\}$

cvalids :: 'a *cntxt* \Rightarrow 'a *cntxt* \Rightarrow *bool* ($- \models_t / - 50$)
 $C \models_t D \equiv \models_t C \longrightarrow \models_t D$

inductive

thoare :: 'a *cntxt* \Rightarrow 'a *cntxt* \Rightarrow *bool* ($(- \models_t / -) 50$)
and *thoare'* :: 'a *cntxt* \Rightarrow 'a *assn* \Rightarrow *com* \Rightarrow 'a *assn* \Rightarrow *bool*
 $((- \models_t / (\{(1-)\} / (-) / \{(1-)\})) [50,0,0,0] 50)$

where

$C \models_t \{P\}c\{Q\} \equiv C \models_t \{(P,c,Q)\}$
| *Do*: $C \models_t \{\lambda z\ s. (\forall t \in f\ s. P\ z\ t) \wedge f\ s \neq \{\}\} Do\ f\ \{P\}$
| *Semi*: $\llbracket C \models_t \{P\}c1\{Q\}; C \models_t \{Q\}c2\{R\} \rrbracket \implies C \models_t \{P\} c1;c2 \{R\}$
| *If*: $\llbracket C \models_t \{\lambda z\ s. P\ z\ s \wedge b\ s\}c\{Q\}; C \models_t \{\lambda z\ s. P\ z\ s \wedge \sim b\ s\}d\{Q\} \rrbracket \implies$

$C \vdash_t \{P\} \text{ IF } b \text{ THEN } c \text{ ELSE } d \{Q\}$
| *While*:
 $\llbracket \text{wf } r; \forall s'. C \vdash_t \{\lambda z s. P z s \wedge b s \wedge s' = s\} c \{\lambda z s. P z s \wedge (s, s') \in r\} \rrbracket$
 $\implies C \vdash_t \{P\} \text{ WHILE } b \text{ DO } c \{\lambda z s. P z s \wedge \neg b s\}$

| *Call*:
 $\llbracket \text{wf } r;$
 $\quad \forall q \text{ pre.}$
 $\quad (\bigcup p. \{(\lambda z s. P p z s \wedge ((p, s), (q, \text{pre})) \in r, \text{CALL } p, Q p)\})$
 $\quad \vdash_t \{\lambda z s. P q z s \wedge s = \text{pre}\} \text{ body } q \{Q q\} \rrbracket$
 $\implies \{ \} \vdash_t \bigcup p. \{(P p, \text{CALL } p, Q p)\}$

| *Asm*: $(P, \text{CALL } p, Q) \in C \implies C \vdash_t \{P\} \text{ CALL } p \{Q\}$

| *Conseq*:
 $\llbracket C \vdash_t \{P'\} c \{Q'\};$
 $\quad (\forall s t. (\forall z. P' z s \longrightarrow Q' z t) \longrightarrow (\forall z. P z s \longrightarrow Q z t)) \wedge$
 $\quad (\forall s. (\exists z. P z s) \longrightarrow (\exists z. P' z s)) \rrbracket$
 $\implies C \vdash_t \{P\} c \{Q\}$

| *ConjI*: $\forall (P, c, Q) \in D. C \vdash_t \{P\} c \{Q\} \implies C \vdash_t D$
| *ConjE*: $\llbracket C \vdash_t D; (P, c, Q) \in D \rrbracket \implies C \vdash_t \{P\} c \{Q\}$

| *Local*: $\llbracket \forall s'. C \vdash_t \{\lambda z s. P z s' \wedge s = f s'\} c \{\lambda z t. Q z (g s' t)\} \rrbracket \implies$
 $C \vdash_t \{P\} \text{ LOCAL } f; g \{Q\}$

monos *split-beta*

lemma *strengthen-pre*:

$\llbracket \forall z s. P' z s \longrightarrow P z s; C \vdash_t \{P\} c \{Q\} \rrbracket \implies C \vdash_t \{P'\} c \{Q\}$
by(*rule thoare. Conseq, assumption, blast*)

lemma *weaken-post*:

$\llbracket C \vdash_t \{P\} c \{Q\}; \forall z s. Q z s \longrightarrow Q' z s \rrbracket \implies C \vdash_t \{P\} c \{Q'\}$
by(*erule thoare. Conseq, blast*)

theorems *tvalid-defs = tvalid-def ctvalid-def valids-def cvalids-def valid-defs*

lemma [*iff*]:

$(\models_t \{\lambda z s. \exists n. P n z s\} c \{Q\}) = (\forall n. \models_t \{P n\} c \{Q\})$

apply(*unfold tvalid-defs*)

apply *fast*

done

lemma [*iff*]:

$(\models_t \{\lambda z s. P z s \wedge P'\} c \{Q\}) = (P' \longrightarrow \models_t \{P\} c \{Q\})$

apply(*unfold tvalid-defs*)

apply *fast*

done

lemma [*iff*]: $(\models_t \{P\} \text{CALL } p\{Q\}) = (\models_t \{P\} \text{body } p\{Q\})$
apply(*unfold tvalid-defs*)
apply *fast*
done

lemma *unfold-while*:
 $(s - \text{WHILE } b \text{ DO } c \rightarrow u) =$
 $(s - \text{IF } b \text{ THEN } c; \text{WHILE } b \text{ DO } c \text{ ELSE Do}(\lambda s. \{s\}) \rightarrow u)$
by(*auto elim: exec.cases intro: exec.intros split: split-if-asm*)

theorem $C \Vdash_t D \implies C \models_t D$
apply(*erule thoare.induct*)
apply(*simp only: tvalid-defs*)
apply *fast*
apply(*simp add: tvalid-defs*)
apply *fast*
apply(*simp only: tvalid-defs*)
apply *clarsimp*
prefer 3
apply(*simp add: tvalid-defs*)
apply *fast*
prefer 3
apply(*simp add: tvalid-defs*)
apply *blast*
apply(*simp add: tvalid-defs*)
apply(*rule impI, rule conjI*)
apply(*rule allI*)
apply(*erule wf-induct*)
apply *clarify*
apply(*drule unfold-while[THEN iffD1]*)
apply (*simp split add: split-if-asm*)
apply *fast*
apply(*rule allI, rule allI*)
apply(*erule wf-induct*)
apply *clarify*
apply(*case-tac b x*)
prefer 2
apply (*erule termi.WhileFalse*)
apply(*rule termi.WhileTrue, assumption*)
apply *fast*
apply (*subgoal-tac (t,x):r*)
apply *fast*
apply *blast*

defer

apply(*simp add: tvalid-defs*)
apply *fast*

```

apply(simp (no-asm-use) add:tvalid-defs)
apply fast

apply(simp add:tvalid-defs)
apply fast

apply(simp (no-asm-use) add:valids-def cvalid-def cvalids-def)
apply(rule allI)
apply(rename-tac q)
apply(subgoal-tac  $\forall pre. \models_t \{\lambda z s. P (fst(q,pre)) z s \ \& \ s=(snd(q,pre))\}$  body
(fst(q,pre)) {Q (fst(q,pre))})
  apply(simp (no-asm-use) add:tvalid-defs)
  apply fast
apply(rule allI)
apply(erule-tac wf-induct)
apply(simp add:split-paired-all)
apply(rename-tac q pre)
apply(erule allE, erule allE, erule conjE, erule impE)
  prefer 2
  apply assumption
apply(rotate-tac 1, erule thin-rl)
apply(unfold tvalid-defs)
apply fast
done

constdefs MGTt :: com  $\Rightarrow$  state assn  $\times$  com  $\times$  state assn
  MGTt c  $\equiv$  ( $\lambda z s. z = s \wedge c \downarrow s, c, \lambda z t. z -c \rightarrow t$ )
declare MGTt-def[simp]

lemma MGT-implies-complete:
  {}  $\vdash_t \{MGT_t c\} \Longrightarrow \{\} \models_t \{P\}c\{Q\} \Longrightarrow \{\} \vdash_t \{P\}c\{Q::state\ assn\}$ 
apply(unfold MGTt-def)
apply (erule thoare.Conseq)
apply(simp add: tvalid-defs)
apply blast
done

lemma while-termiE:  $\llbracket WHILE\ b\ DO\ c \downarrow s; b\ s \rrbracket \Longrightarrow c \downarrow s$ 
by(erule termi.cases, auto)

lemma while-termiE2:  $\llbracket WHILE\ b\ DO\ c \downarrow s; b\ s; s -c \rightarrow t \rrbracket \Longrightarrow WHILE\ b\ DO\ c \downarrow t$ 
by(erule termi.cases, auto)

lemma MGT-lemma:  $\forall p. \{\} \vdash_t \{MGT_t(CALL\ p)\} \Longrightarrow \{\} \vdash_t \{MGT_t\ c\}$ 
apply (simp)
apply(induct-tac c)
  apply (rule strengthen-pre[OF - thoare.Do])

```

```

apply blast
apply(rule-tac  $Q = \lambda z s. z -com1 \rightarrow s \ \& \ com2 \downarrow s$  in thoare.Semi)
apply(erule thoare.Conseq)
apply fast
apply(erule thoare.Conseq)
apply fast
apply(rule thoare.If)
apply(erule thoare.Conseq)
apply simp
apply(erule thoare.Conseq)
apply simp
defer
apply simp
apply(fast intro:thoare.Local elim!: thoare.Conseq)
apply(rename-tac  $b \ c$ )
apply(rule-tac  $P' = \lambda z s. (z,s) \in (\{(s,t). b \ s \wedge s -c \rightarrow t\})^* \wedge$ 
 $WHILE \ b \ DO \ c \downarrow s$  in thoare.Conseq)
apply(rule-tac thoare.While[OF wf-termi])
apply(rule allI)
apply(erule thoare.Conseq)
apply(fastsimp intro:rtrancl-into-rtrancl dest:while-termiE while-termiE2)
apply(rule conjI)
apply clarsimp
apply(erule-tac  $x = s$  in allE)
apply clarsimp
apply(erule converse-rtrancl-induct)
apply(erule exec.WhileFalse)
apply(fast elim:exec.WhileTrue)
apply(fast intro: rtrancl-refl)
done

```

inductive-set

```

exec1 :: ((com list  $\times$  state)  $\times$  (com list  $\times$  state))set
and exec1' :: (com list  $\times$  state)  $\Rightarrow$  (com list  $\times$  state)  $\Rightarrow$  bool ( $- \rightarrow -$  [81,81]
100)

```

where

```

 $cs0 \rightarrow cs1 \equiv (cs0, cs1) : exec1$ 

```

```

| Do[iff]:  $t \in f \ s \Longrightarrow ((Do \ f)\#cs,s) \rightarrow (cs,t)$ 

```

```

| Semi[iff]:  $((c1;c2)\#cs,s) \rightarrow (c1\#c2\#cs,s)$ 

```

```

| IfTrue:  $b \ s \Longrightarrow ((IF \ b \ THEN \ c1 \ ELSE \ c2)\#cs,s) \rightarrow (c1\#cs,s)$ 

```

```

| IfFalse:  $\neg b \ s \Longrightarrow ((IF \ b \ THEN \ c1 \ ELSE \ c2)\#cs,s) \rightarrow (c2\#cs,s)$ 

```

```

| WhileFalse:  $\neg b \ s \Longrightarrow ((WHILE \ b \ DO \ c)\#cs,s) \rightarrow (cs,s)$ 

```

```

| WhileTrue:  $b \ s \Longrightarrow ((WHILE \ b \ DO \ c)\#cs,s) \rightarrow (c\#(WHILE \ b \ DO \ c)\#cs,s)$ 

```

| *Call*[*iff*]: (*CALL* *p*#*cs,s*) → (*body* *p*#*cs,s*)

| *Local*[*iff*]: ((*LOCAL* *f*;*c*;*g*)#*cs,s*) → (*c* # *Do*($\lambda t. \{g\ s\ t\}$)#*cs, f s*)

abbreviation

exectr :: (*com list* × *state*) ⇒ (*com list* × *state*) ⇒ *bool* (- →* - [81,81] 100)
where *cs0* →* *cs1* ≡ (*cs0,cs1*) : *exec1* ^*

inductive-cases *exec1E*[*elim!*]:

([],*s*) → (*cs',s'*)
(*Do* *f*#*cs,s*) → (*cs',s'*)
(*c1*;*c2*)#*cs,s*) → (*cs',s'*)
(*IF* *b* *THEN* *c1* *ELSE* *c2*)#*cs,s*) → (*cs',s'*)
(*WHILE* *b* *DO* *c*)#*cs,s*) → (*cs',s'*)
(*CALL* *p*#*cs,s*) → (*cs',s'*)
(*LOCAL* *f*;*c*;*g*)#*cs,s*) → (*cs',s'*)

lemma [*iff*]: ¬ ([],*s*) → *u*
by (*induct* *u*) *blast*

lemma *app-exec*: (*cs,s*) → (*cs',s'*) ⇒ (*cs@cs2,s*) → (*cs'@cs2,s'*)
apply(*erule* *exec1.induct*)
apply(*simp-all* *del:fun-upd-apply*)
apply(*blast* *intro:exec1.intros*)
done

lemma *app-execs*: (*cs,s*) →* (*cs',s'*) ⇒ (*cs@cs2,s*) →* (*cs'@cs2,s'*)
apply(*erule* *rtrancl-induct2*)
apply *blast*
apply(*blast* *intro:app-exec* *rtrancl-trans*)
done

lemma *exec-impl-execs*[*rule-format*]:
s -c→ s' ⇒ ∀ cs. (c#cs,s) → (cs,s')*
apply(*erule* *exec.induct*)
apply *blast*
apply(*blast* *intro:rtrancl-trans*)
apply(*blast* *intro:exec1.IfTrue* *rtrancl-trans*)
apply(*blast* *intro:exec1.IfFalse* *rtrancl-trans*)
apply(*blast* *intro:exec1.WhileFalse* *rtrancl-trans*)
apply(*blast* *intro:exec1.WhileTrue* *rtrancl-trans*)
apply(*blast* *intro: rtrancl-trans*)
apply(*blast* *intro: rtrancl-trans*)
done

inductive

execs :: *state* ⇒ *com list* ⇒ *state* ⇒ *bool* (-/ ==>/ - [50,0,50] 50)
where
s = [] ⇒ s

| $s - c \rightarrow t \implies t = cs \Rightarrow u \implies s = c\#cs \Rightarrow u$

inductive-cases [*elim!*]:

$s = [] \Rightarrow t$

$s = c\#cs \Rightarrow t$

theorem *exec1s-impl-execs*: $(cs, s) \rightarrow^* ([], t) \implies s = cs \Rightarrow t$

apply(*erule converse-rtrancl-induct2*)

apply(*rule execs.intros*)

apply(*erule exec1.cases*)

apply(*blast intro:execs.intros*)

apply(*blast intro:execs.intros*)

apply(*fastsimp intro:execs.intros*)

apply(*fastsimp intro:execs.intros*)

apply(*blast intro:execs.intros exec.intros*)+

done

theorem *exec1s-impl-exec*: $([c], s) \rightarrow^* ([], t) \implies s - c \rightarrow t$

by(*blast dest: exec1s-impl-execs*)

consts *termis* :: *com list* \Rightarrow *state* \Rightarrow *bool* (**infixl** \Downarrow 60)

primrec

$[] \Downarrow s = \text{True}$

$c\#cs \Downarrow s = (c \downarrow s \wedge (\forall t. s - c \rightarrow t \longrightarrow cs \Downarrow t))$

lemma *exec1-pres-termis*: $(cs, s) \rightarrow (cs', s') \implies cs \Downarrow s \longrightarrow cs' \Downarrow s'$

apply(*erule exec1.induct*)

apply(*simp-all del:fun-upd-apply*)

apply *blast*

apply(*blast intro:exec.WhileFalse*)

apply(*blast intro:while-termiE while-termiE2 exec.WhileTrue*)

apply *blast*

done

lemma *execs-pres-termis*: $(cs, s) \rightarrow^* (cs', s') \implies cs \Downarrow s \longrightarrow cs' \Downarrow s'$

apply(*erule rtrancl-induct2*)

apply *blast*

apply(*blast dest:exec1-pres-termis*)

done

lemma *execs-pres-termi*: $\llbracket ([c], s) \rightarrow^* (c'\#cs', s'); c \downarrow s \rrbracket \implies c' \downarrow s'$

apply(*insert execs-pres-termis[of [c] - c'\#cs',simplified]*)

apply *blast*

done

constdefs

termi-call-steps :: $((pname \times state) \times (pname \times state))set$

termi-call-steps \equiv
 $\{((q,t),(p,s)). \text{body } p \downarrow s \wedge (\exists cs. ([\text{body } p], s) \rightarrow^* (\text{CALL } q\# \text{ cs}, t))\}$

lemma *lem*:

$\!y. (a,y):r^+ \longrightarrow P a \longrightarrow P y \implies ((b,a) : \{(y,x). P x \wedge (x,y):r^+\}^+) = ((b,a) : \{(y,x). P x \wedge (x,y):r^+\})$

apply(*rule iffI*)
apply *clarify*
apply(*erule trancl-induct*)
apply *blast*
apply(*blast intro:trancl-trans*)
apply *clarify*
apply(*erule trancl-induct*)
apply *blast*
apply(*blast intro:trancl-trans*)
done

lemma *renumber-aux*:

$\llbracket \forall i. (a,f i) : r^* \wedge (f i, f(\text{Suc } i)) : r; (a,b) : r^* \rrbracket \implies b = f 0 \longrightarrow (\exists f. f 0 = a \ \& \ (\forall i. (f i, f(\text{Suc } i)) : r))$

apply(*erule converse-rtrancl-induct*)
apply *blast*
apply(*clarsimp*)
apply(*rule-tac x= $\lambda i. \text{case } i \text{ of } 0 \Rightarrow y \mid \text{Suc } i \Rightarrow f a \ i$ in exI*)
apply *simp*
apply *clarify*
apply(*case-tac i*)
apply *simp-all*
done

lemma *renumber*:

$\forall i. (a,f i) : r^* \wedge (f i, f(\text{Suc } i)) : r \implies \exists f. f 0 = a \ \& \ (\forall i. (f i, f(\text{Suc } i)) : r)$
by(*blast dest:renumber-aux*)

constdefs *inf* :: *com list* \Rightarrow *state* \Rightarrow *bool*
inf *cs s* == $\exists f. f 0 = (cs,s) \wedge (\forall i. f i \rightarrow f(\text{Suc } i))$

lemma [*iff*]: $\neg \text{inf } [] \ s$
apply(*unfold inf-def*)
apply *clarify*
apply(*erule-tac x = 0 in allE*)
apply *simp*
done

lemma [*iff*]: $\neg \text{inf } [\text{Do } f] \ s$
apply(*unfold inf-def*)
apply *clarify*

```

apply(frule-tac  $x = 0$  in spec)
apply(erule-tac  $x = 1$  in allE)
apply (case-tac fa (Suc 0))
apply clarsimp
done

```

```

lemma [iff]: inf ((c1;c2)#cs) s = inf (c1#c2#cs) s
apply(unfold inf-def)
apply(rule iffI)
apply clarify
apply(rule-tac  $x = \lambda i. f(\text{Suc } i)$  in exI)
apply(frule-tac  $x = 0$  in spec)
apply (case-tac f (Suc 0))
apply clarsimp
apply clarify
apply(rule-tac  $x = \lambda i. \text{case } i \text{ of } 0 \Rightarrow ((c1;c2)\#cs,s) \mid \text{Suc } i \Rightarrow f \ i$  in exI)
apply(simp split:nat.split)
done

```

```

lemma [iff]: inf ((IF b THEN c1 ELSE c2)#cs) s =
               inf ((if b s then c1 else c2)#cs) s
apply(unfold inf-def)
apply(rule iffI)
apply clarsimp
apply(frule-tac  $x = 0$  in spec)
apply (case-tac f (Suc 0))
apply(rule conjI)
apply clarsimp
apply(rule-tac  $x = \lambda i. f(\text{Suc } i)$  in exI)
apply clarsimp
apply clarsimp
apply(rule-tac  $x = \lambda i. f(\text{Suc } i)$  in exI)
apply clarsimp
apply clarsimp
apply(rule-tac  $x = \lambda i. \text{case } i \text{ of } 0 \Rightarrow ((IF \ b \ THEN \ c1 \ ELSE \ c2)\#cs,s) \mid \text{Suc } i \Rightarrow f \ i$  in exI)
apply(simp add: exec1.intros split:nat.split)
done

```

```

lemma [simp]:
  inf ((WHILE b DO c)#cs) s =
    (if b s then inf (c#(WHILE b DO c)#cs) s else inf cs s)
apply(unfold inf-def)
apply(rule iffI)
apply clarsimp
apply(frule-tac  $x = 0$  in spec)
apply (case-tac f (Suc 0))
apply(rule conjI)
apply clarsimp

```

```

apply(rule-tac x =  $\lambda i. f(\text{Suc } i)$  in exI)
apply clarsimp
apply clarsimp
apply(rule-tac x =  $\lambda i. f(\text{Suc } i)$  in exI)
apply clarsimp
apply (clarsimp split:if-splits)
apply(rule-tac x =  $\lambda i. \text{case } i \text{ of } 0 \Rightarrow ((\text{WHILE } b \text{ DO } c)\#cs,s) \mid \text{Suc } i \Rightarrow f \ i$  in
exI)
apply(simp add: exec1.intros split:nat.split)
apply(rule-tac x =  $\lambda i. \text{case } i \text{ of } 0 \Rightarrow ((\text{WHILE } b \text{ DO } c)\#cs,s) \mid \text{Suc } i \Rightarrow f \ i$  in
exI)
apply(simp add: exec1.intros split:nat.split)
done

```

```

lemma [iff]: inf (CALL p#cs) s = inf (body p#cs) s
apply(unfold inf-def)
apply(rule iffI)
apply clarsimp
apply(frule-tac x = 0 in spec)
apply (case-tac f (Suc 0))
apply clarsimp
apply(rule-tac x =  $\lambda i. f(\text{Suc } i)$  in exI)
apply clarsimp
apply clarsimp
apply(rule-tac x =  $\lambda i. \text{case } i \text{ of } 0 \Rightarrow (\text{CALL } p\#cs,s) \mid \text{Suc } i \Rightarrow f \ i$  in exI)
apply(simp add: exec1.intros split:nat.split)
done

```

```

lemma [iff]: inf ((LOCAL f;c;g)#cs) s =
inf (c#Do( $\lambda t. \{g \ s \ t\}$ )#cs) (f s)
apply(unfold inf-def)
apply(rule iffI)
apply clarsimp
apply(rename-tac F)
apply(frule-tac x = 0 in spec)
apply (case-tac F (Suc 0))
apply clarsimp
apply(rule-tac x =  $\lambda i. F(\text{Suc } i)$  in exI)
apply clarsimp
apply (clarsimp)
apply(rename-tac F)
apply(rule-tac x =  $\lambda i. \text{case } i \text{ of } 0 \Rightarrow ((\text{LOCAL } f;c;g)\#cs,s) \mid \text{Suc } i \Rightarrow F \ i$  in
exI)
apply(simp add: exec1.intros split:nat.split)
done

```

```

lemma exec1-only1-aux: (ccs,s)  $\rightarrow$  (cs',t)  $\implies$ 
 $\forall c \ cs. \ ccs = c\#cs \longrightarrow (\exists cs1. cs' = cs1 \ @ \ cs)$ 
apply(erule exec1.induct)

```

apply *force+*
done

lemma *exec1-only1*: $(c \# cs, s) \rightarrow (cs', t) \implies \exists cs1. cs' = cs1 @ cs$
by(*blast dest:exec1-only1-aux*)

lemma *exec1-drop-suffix-aux*:
 $(cs12, s) \rightarrow (cs1'2, s') \implies !cs1 cs2 cs1'$
 $cs12 = cs1 @ cs2 \ \& \ cs1'2 = cs1' @ cs2 \ \& \ cs1 \neq [] \longrightarrow (cs1, s) \rightarrow (cs1', s')$
apply(*erule exec1.induct*)
apply (*force intro:exec1.intros simp add: neq-Nil-conv*)
done

lemma *exec1-drop-suffix*:
 $(cs1 @ cs2, s) \rightarrow (cs1' @ cs2, s') \implies cs1 \neq [] \implies (cs1, s) \rightarrow (cs1', s')$
by(*blast dest:exec1-drop-suffix-aux*)

lemma *execs-drop-suffix*[*rule-format(no-asm)*]:
 $\llbracket f \ 0 = (c \# cs, s); !i. f(i) \rightarrow f(Suc \ i) \rrbracket \implies$
 $(!i < k. p \ i \neq [] \ \& \ fst(f \ i) = p \ i @ cs) \longrightarrow fst(f \ k) = p \ k @ cs$
 $\longrightarrow ([c], s) \rightarrow^* (p \ k, snd(f \ k))$
apply(*induct-tac k*)
apply *simp*
apply (*clarsimp*)
apply(*erule rtrancl-into-rtrancl*)
apply(*erule-tac x = n in allE*)
apply(*erule-tac x = n in allE*)
apply(*case-tac f n*)
apply(*case-tac f(Suc n)*)
apply *simp*
apply(*blast dest:exec1-drop-suffix*)
done

lemma *execs-drop-suffix0*:
 $\llbracket f \ 0 = (c \# cs, s); !i. f(i) \rightarrow f(Suc \ i); !i < k. p \ i \neq [] \ \& \ fst(f \ i) = p \ i @ cs;$
 $fst(f \ k) = cs; p \ k = [] \rrbracket \implies ([c], s) \rightarrow^* ([], snd(f \ k))$
apply(*drule execs-drop-suffix, assumption, assumption*)
apply *simp*
apply *simp*
done

lemma *skolemize1*: $\forall x. P \ x \longrightarrow (\exists y. Q \ x \ y) \implies \exists f. \forall x. P \ x \longrightarrow Q \ x \ (f \ x)$
apply(*rule-tac x = $\lambda x. SOME \ y. Q \ x \ y$ in exI*)
apply(*fast intro:someI2*)
done

lemma *least-aux*: $\llbracket f \ 0 = (c \ \# \ cs, \ s); \forall i. f \ i \rightarrow f \ (Suc \ i);$
 $fst(f \ k) = cs; !i < k. fst(f \ i) \neq cs \rrbracket$
 $\implies \forall i \leq k. (\exists p. (p \neq []) = (i < k) \ \& \ fst(f \ i) = p @ cs)$

```

apply(rule allI)
apply(induct-tac i)
  apply simp
  apply (rule ccontr)
  apply simp
apply clarsimp
apply(drule order-le-imp-less-or-eq)
apply(erule disjE)
  prefer 2
  apply simp
apply simp
apply(erule-tac x = n in allE)
apply(erule-tac x = Suc n in allE)
apply(case-tac f n)
apply(case-tac f (Suc n))
apply simp
apply(rename-tac sn csn1 sn1)
apply (clarsimp simp add: neq-Nil-conv)
apply(drule exec1-only1)
apply (clarsimp simp add: neq-Nil-conv)
apply(erule disjE)
  apply clarsimp
apply clarsimp
apply(case-tac cs1)
  apply simp
apply simp
done

```

```

lemma least-lem:  $\llbracket f\ 0 = (c\#\ cs,s); !i. f\ i \rightarrow f(Suc\ i); EX\ i. fst(f\ i) = cs \rrbracket$ 
   $\implies EX\ k. fst(f\ k) = cs \ \& \ ([c],s) \rightarrow^* (\llbracket, snd(f\ k)\rrbracket)$ 
apply(rule-tac x=LEAST i. fst(f i) = cs in exI)
apply(rule conjI)
  apply(fast intro: LeastI)
apply(subgoal-tac
  !i<=LEAST i. fst (f i) = cs. EX p. ((p  $\neq$  []) = (i<(LEAST i. fst (f i) = cs)))
  & fst(f i) = p@cs)
  apply(drule skolemize1)
  apply clarify
  apply(rename-tac p)
  apply(erule-tac p=p in execs-drop-suffix0, assumption)
  apply (blast dest:order-less-imp-le)
  apply(fast intro: LeastI)
apply(erule thin-rl)
apply(erule-tac x = LEAST j. fst (f j) = fst (f i) in allE)
apply blast
apply(erule least-aux,assumption)
  apply(fast intro: LeastI)
apply clarify
apply(drule not-less-Least)

```

apply *blast*
done

lemma *skolemize2*: $\forall x.\exists y. P x y \implies \exists f.\forall x. P x (f x)$
apply(*rule-tac* $x = \lambda x. \text{SOME } y. P x y$ **in** *exI*)
apply(*fast intro:someI2*)
done

lemma *inf-cases*: $\text{inf } (c\#cs) s \implies \text{inf } [c] s \vee (\exists t. s -c \rightarrow t \wedge \text{inf } cs t)$
apply(*unfold inf-def*)
apply (*clarsimp del: disjCI*)
apply(*case-tac* $EX i. \text{fst}(f i) = cs$)
 apply(*rule disjI2*)
 apply(*drule least-lem, assumption, assumption*)
 apply *clarify*
 apply(*drule exec1s-impl-exec*)
 apply(*case-tac* $f k$)
 apply *simp*
 apply (*rule exI, rule conjI, assumption*)
 apply(*rule-tac* $x = \lambda i. f(i+k)$ **in** *exI*)
 apply (*clarsimp*)
apply(*rule disjI1*)
apply *simp*
apply(*subgoal-tac* $\forall i. \exists p. p \neq [] \wedge \text{fst}(f i) = p@cs$)
 apply(*drule skolemize2*)
 apply *clarify*
 apply(*rename-tac* p)
 apply(*rule-tac* $x = \lambda i. (p i, \text{snd}(f i))$ **in** *exI*)
 apply(*rule conjI*)
 apply(*erule-tac* $x = 0$ **in** *alle, erule conjE*)
 apply *simp*
 apply *clarify*
 apply(*erule-tac* $x = i$ **in** *alle*)
 apply(*erule-tac* $x = i$ **in** *alle*)
 apply(*frule-tac* $x = i$ **in** *spec*)
 apply(*erule-tac* $x = \text{Suc } i$ **in** *alle*)
 apply(*case-tac* $f i$)
 apply(*case-tac* $f(\text{Suc } i)$)
 apply *clarsimp*
 apply(*blast intro:exec1-drop-suffix*)
apply(*clarify*)
apply(*induct-tac* i)
 apply *force*
 apply *clarsimp*
 apply(*case-tac* p)
 apply *blast*
 apply(*erule-tac* $x=n$ **in** *alle*)
 apply(*erule-tac* $x=\text{Suc } n$ **in** *alle*)
 apply(*case-tac* $f n$)

```

apply(case-tac f(Suc n))
apply clarsimp
apply(drule exec1-only1)
apply clarsimp
done

```

```

lemma termi-impl-not-inf:  $c \downarrow s \implies \neg \text{inf } [c] s$ 
apply(erule termi.induct)

```

```

  apply clarify

```

```

  apply(blast dest:inf-cases)

```

```

    apply clarsimp
    apply clarsimp

```

```

    apply clarsimp
    apply(fastsimp dest:inf-cases)

```

```

  apply blast

```

```

apply(blast dest:inf-cases)
done

```

```

lemma termi-impl-no-inf-chain:

```

```

   $c \downarrow s \implies \neg(\exists f. f 0 = ([c], s) \wedge (\forall i::\text{nat}. (f i, f(i+1)) : \text{exec1}^{\wedge+}))$ 
apply(subgoal-tac wf(\{(y,x). ([c],s) \to^* x \& x \to y\}^{\wedge+}))
apply(simp only:wf-iff-no-infinite-down-chain)
apply(erule contrapos-nn)
apply clarify
apply(subgoal-tac !i. ([c], s) \to^* f i)
prefer 2
apply(rule allI)
apply(induct-tac i)
  apply simp
  apply simp
  apply(blast intro: trancl-into-rtrancl rtrancl-trans)
apply(rule-tac x=f in exI)
apply clarify
apply(drule-tac x=i in spec)
apply(subst lem)
  apply(blast intro: trancl-into-rtrancl rtrancl-trans)
  apply clarsimp
apply(rule wf-trancl)
apply(simp only:wf-iff-no-infinite-down-chain)
apply(clarify)
apply simp
apply(drule renumber)
apply(fold inf-def)

```

```

apply(simp add: termi-impl-not-inf)
done

consts cseq :: (nat  $\Rightarrow$  pname  $\times$  state)  $\Rightarrow$  nat  $\Rightarrow$  com list
primrec
cseq S 0 = []
cseq S (Suc i) = (SOME cs. ([body(fst(S i))], snd(S i))  $\rightarrow^*$ 
  (CALL(fst(S(i+1)))# cs, snd(S(i+1)))) @ cseq S i

lemma wf-termi-call-steps: wf termi-call-steps
apply(unfold termi-call-steps-def)
apply(simp only:wf-iff-no-infinite-down-chain)
apply(clarify)
apply(rename-tac S)
apply simp
apply(subgoal-tac
   $\exists$  Cs. Cs 0 = [] & ( $\forall i. (body(fst(S i)) \# Cs i, snd(S i)) \rightarrow^*$ 
    (CALL(fst(S(i+1)))# Cs(i+1), snd(S(i+1)))))

prefer 2
apply(rule-tac x = cseq S in exI)
apply clarsimp
apply(erule-tac x=i in allE)
apply clarsimp
apply(rename-tac q t p s cs)
apply(erule-tac P =  $\lambda cs. ([body p], s) \rightarrow^* (CALL q \# cs, t)$  in someI2)
apply(fastsimp dest:app-execs)
apply clarify
apply(subgoal-tac
   $\forall i. ((body(fst(S i)) \# Cs i, snd(S i)), (body(fst(S(i+1))) \# Cs(i+1), snd(S(i+1))))$ 
  : exec1+)
prefer 2
apply(blast intro:rtrancl-into-trancl1)
apply(subgoal-tac  $\exists f. f 0 = ([body(fst(S 0))], snd(S 0)) \wedge (\forall i. (f i, f(i+1)) :$ 
  exec1+)
prefer 2
apply(rule-tac x =  $\lambda i. (body(fst(S i)) \# Cs i, snd(S i))$  in exI)
apply blast
apply(erule-tac x=0 in allE)
apply(simp add:split-def)
apply clarify
apply(drule termi-impl-no-inf-chain)
apply simp
apply blast
done

lemma CALL-lemma:
 $(\bigcup p. \{(\lambda z s. (z=s \wedge body p \downarrow s) \wedge ((p,s),(q,pre)) \in termi-call-steps, CALL p,$ 
   $\lambda z s. z \dashv body p \rightarrow s)\}) \vdash_t$ 
 $\{\lambda z s. (z=s \wedge body q \downarrow pre) \wedge (\exists cs. ([body q],pre) \rightarrow^* (c \# cs, s))\} c$ 

```

```

{ $\lambda z s. z - c \rightarrow s$ }
apply(induct-tac c)

  apply (rule strengthen-pre[OF - thoare.Do])
  apply(blast dest: execs-pres-termi)

apply(rename-tac c1 c2)
apply(rule-tac Q =  $\lambda z s. body\ q \downarrow pre \ \& \ (EX\ cs. ([body\ q],\ pre) \rightarrow^* (c2\ \#\ cs,\ s))$ )
& z - c1  $\rightarrow$  s & c2  $\downarrow$  s in thoare.Semi)
  apply(erule thoare.Conseq)
  apply(rule conjI)
  apply clarsimp
  apply(subgoal-tac s - c1  $\rightarrow$  t)
  prefer 2
  apply(blast intro: exec1.Semi exec-impl-execs rtrancl-trans)
  apply(subgoal-tac ([body q], pre)  $\rightarrow^*$  (c2 # cs, t))
  prefer 2
  apply(blast intro:exec1.Semi[THEN r-into-rtrancl] exec-impl-execs rtrancl-trans)
  apply(subgoal-tac ([body q], pre)  $\rightarrow^*$  (c2 # cs, t))
  prefer 2
  apply(blast intro: exec-impl-execs rtrancl-trans)
  apply(blast intro:exec-impl-execs rtrancl-trans execs-pres-termi)
  apply(fast intro: exec1.Semi rtrancl-trans)
  apply(erule thoare.Conseq)
  apply blast

prefer 3
apply(simp only:termi-call-steps-def)
apply(rule thoare.Conseq[OF thoare.Asm])
  apply blast
apply(blast dest: execs-pres-termi)

apply(rule thoare.If)
  apply(erule thoare.Conseq)
  apply simp
  apply(blast intro: exec1.IfTrue rtrancl-trans)
apply(erule thoare.Conseq)
  apply simp
  apply(blast intro: exec1.IfFalse rtrancl-trans)

defer
apply simp
apply(rule thoare.Local)
apply(rule allI)
apply(erule thoare.Conseq)
apply (clarsimp)
apply(rule conjI)
apply (clarsimp)
apply(drule rtrancl-trans[OF - r-into-rtrancl[OF exec1.Local]])

```

```

apply(fast)
apply (clarsimp)
apply(drule rtrancl-trans[OF - r-into-rtrancl[OF exec1.Local]])
apply blast

apply(rename-tac b c)
apply(rule-tac  $P' = \lambda z s. (z,s) \in (\{(s,t). b s \wedge s -c \rightarrow t\})^* \wedge \text{body } q \downarrow \text{pre} \wedge$ 
 $(\exists cs. ([\text{body } q], \text{pre}) \rightarrow^* ((\text{WHILE } b \text{ DO } c) \# cs, s))$  in thoare.Conseq)
apply(rule-tac thoare.While[OF wf-termi])
apply(rule allI)
apply(erule thoare.Conseq)
apply clarsimp
apply(rule conjI)
apply clarsimp
apply(rule conjI)
apply(blast intro: rtrancl-trans exec1.WhileTrue)
apply(rule conjI)
apply(rule exI, rule rtrancl-trans, assumption)
apply(blast intro: exec1.WhileTrue exec-impl-execs rtrancl-trans)
apply(rule conjI)
apply(blast intro:execs-pres-termi)
apply(blast intro: exec1.WhileTrue exec-impl-execs rtrancl-trans)
apply(blast intro: exec1.WhileTrue exec-impl-execs rtrancl-trans)
apply(rule conjI)
apply clarsimp
apply(erule-tac  $x = s$  in allE)
apply clarsimp
apply(erule impE)
apply blast
apply clarify
apply(erule-tac  $a=s$  in converse-rtrancl-induct)
apply(erule exec.WhileFalse)
apply(fast elim:exec.WhileTrue)
apply(fast intro: rtrancl-refl)
done

```

```

lemma CALL-cor:
 $(\bigcup p. \{(\lambda z s. (z=s \wedge \text{body } p \downarrow s) \wedge ((p,s),(q,pre)) \in \text{termi-call-steps}, \text{CALL } p,$ 
 $\lambda z s. z -\text{body } p \rightarrow s)\} \vdash_t$ 
 $\{\lambda z s. (z=s \wedge \text{body } q \downarrow s) \wedge s = \text{pre}\} \text{body } q \{\lambda z s. z -\text{body } q \rightarrow s\}$ 
apply(rule strengthen-pre[OF - CALL-lemma])
apply blast
done

```

```

lemma MGT-CALL:  $\{\} \vdash_t (\bigcup p. \{MGT_t(\text{CALL } p)\})$ 
apply(simp add: MGT_t-def)
apply(rule thoare.Call)
apply(rule wf-termi-call-steps)
apply clarify

```

apply(rule CALL-cor)
done

lemma MGT-CALL1: $\forall p. \{\} \vdash_t \{MGT_t(CALL p)\}$
by(fastsimp intro:MGT-CALL[THEN ConjE])

theorem $\{\} \models_t \{P\}c\{Q\} \implies \{\} \vdash_t \{P\}c\{Q::state\ assn\}$
apply(erule MGT-implies-complete[OF MGT-lemma[OF MGT-CALL1]])
done

end

References

- [1] T. Nipkow. Hoare logics for recursive procedures and unbounded non-determinism. In J. Bradfield, editor, *Computer Science Logic (CSL 2002)*, volume 2471, pages 103–119, 2002.
- [2] T. Nipkow. Hoare logics in Isabelle/HOL. In H. Schwichtenberg and R. Steinbruggen, editors, *Proof and System-Reliability*, pages 341–367. Kluwer, 2002.
- [3] D. v. Oheimb. Hoare logic for mutual recursion and local variables. In C. P. Rangan, V. Raman, and R. Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science (FST&TCS)*, volume 1738, pages 168–180, 1999.