Abstract Hoare Logics

Tobias Nipkow

March 17, 2025

Abstract

These therories 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

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:

type-synonym $bexp = state \Rightarrow bool$

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 $com = Do\ (state \Rightarrow state\ set)$
$\quad |\ Semi\ \ com\ com \qquad (‹\text{-};\ \text{-}›\ \ [60,\ 60]\ 10)$
$\quad |\ Cond\ \ bexp\ com\ com \qquad (‹IF\ \text{-}\ THEN\ \text{-}\ ELSE\ \text{-}›\ \ 60)$
$\quad |\ While\ bexp\ com \qquad (‹WHILE\ \text{-}\ DO\ \text{-}›\ \ 60)$
$\quad |\ Local\ (state \Rightarrow state)\ com\ (state \Rightarrow state \Rightarrow state)$
$\quad\ (‹LOCAL\ \text{-};\ \text{-};\ \text{-}›\ [0,0,60]\ 60)$

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

$$c_1;\ c_2 \qquad IF\ b\ THEN\ c_1\ ELSE\ c_2 \qquad WHILE\ b\ DO\ c$$

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
$\ \ exec :: state \Rightarrow com \Rightarrow state \Rightarrow bool\ (‹\text{-}/\ \text{--}\text{-}\!\!\to/\ \text{-}›\ [50,0,50]\ 50)$
where
$\ \ t \in f\ s \Longrightarrow s\ {-Do\ f\to}\ t$

$|\ [\![\ s0\ {-c1\to}\ s1;\ s1\ {-c2\to}\ s2\ ]\!] \Longrightarrow s0\ {-c1;c2\to}\ s2$

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

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

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

Assuming that the state is a function from variables to values, the declaration of a new local variable x with inital 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(*fastforce 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*:*if-split-asm*)

lemma *while-lemma*[*rule-format*]:
$s\ -w\rightarrow\ t \Longrightarrow \forall b\ c.\ w = WHILE\ b\ DO\ c \wedge P\ s \wedge$
$(\forall s\ s'.\ P\ s \wedge b\ s \wedge s\ -c\rightarrow\ s' \longrightarrow P\ s') \longrightarrow P\ t \wedge \neg b\ t$
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:

type-synonym $assn = state \Rightarrow bool$

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:

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

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
$hoare :: assn \Rightarrow com \Rightarrow assn \Rightarrow bool$ ($\langle\vdash (\{(1\text{-})\}/\ (\text{-})/\ \{(1\text{-})\})\rangle\ 50$)
where
$\vdash \{\lambda s.\ \forall t \in f\ s.\ P\ t\}\ Do\ f\ \{P\}$

$|\ [\![\vdash \{P\}c1\{Q\};\ \vdash \{Q\}c2\{R\}\ ]\!] \Longrightarrow\ \vdash \{P\}\ c1;c2\ \{R\}$

$|\ [\![\vdash \{\lambda s.\ P\ s \wedge b\ s\}\ c1\ \{Q\};\ \vdash \{\lambda s.\ P\ s \wedge \neg b\ s\}\ c2\ \{Q\}\ ]\!]$
$\Longrightarrow \vdash \{P\}\ IF\ b\ THEN\ c1\ ELSE\ c2\ \{Q\}$

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

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

$|\ [\![\bigwedge s.\ P\ s \Longrightarrow P'\ s\ (f\ s);\ \forall s. \vdash \{P'\ s\}\ c\ \{Q \circ (g\ s)\}\ ]\!] \Longrightarrow$
$\vdash \{P\}\ LOCAL\ f;c;g\ \{Q\}$

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*:

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

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*]: $wp\ (Do\ f)\ Q = (\lambda s.\ \forall\, t \in f\ s.\ Q(t))$
apply(*unfold wp-def*)
apply(*rule ext*)
apply *blast*
done

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

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

lemma *wp-while*:

wp (*WHILE b DO c*) *Q* =
  (λ*s. 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* = (λ*s. wp c* (*Q o* (*g s*)) (*f s*))
apply(*unfold wp-def*)
apply(*rule ext*)
apply *auto*
done

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

lemma *weaken-post*: $[\![\ \vdash \{P\}c\{Q\};\ \forall s.\ Q\ s \longrightarrow Q'\ s\ ]\!] \Longrightarrow \vdash \{P\}c\{Q'\}$
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 fun-eq-iff*[*THEN iffD1*, *OF wp-while*, *THEN spec*, *THEN iffD1*])
apply *simp*
apply(*clarify*)
apply(*drule fun-eq-iff*[*THEN iffD1*, *OF wp-while*, *THEN spec*, *THEN iffD1*])
apply(*simp split*:*if-split-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 termiate 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* ⇒ *state* ⇒ *bool* (**infixl** ‹↓› *50*)
where
$f\ s \neq \{\} \implies Do\ f \downarrow s$

| $[\![\ c_1 \downarrow s_0;\ \forall s_1.\ s_0\ -c_1\rightarrow\ s_1 \longrightarrow c_2 \downarrow s_1\ ]\!] \implies (c_1;c_2) \downarrow s_0$

| $[\![\ \ b\ s;\ c_1 \downarrow s\ ]\!] \implies IF\ b\ THEN\ c_1\ ELSE\ c_2 \downarrow s$
| $[\![\ \neg b\ s;\ c_2 \downarrow s\ ]\!] \implies IF\ b\ THEN\ c_1\ ELSE\ c_2 \downarrow s$

| $\neg b\ s \implies WHILE\ b\ DO\ c \downarrow s$
| $[\![\ b\ s;\ c \downarrow s;\ \forall t.\ s\ -c\rightarrow\ t \longrightarrow WHILE\ b\ DO\ c \downarrow t\ ]\!] \implies WHILE\ b\ DO\ c \downarrow s$

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

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*]:
 $w{\downarrow}fk \Longrightarrow$
 $(\forall k\ b\ c.\ fk = f\ k \wedge w = WHILE\ b\ DO\ c \wedge (\forall i.\ f\ i\ -c\rightarrow\ f(Suc\ i))$
 $\longrightarrow (\exists i.\ \neg b(f\ i)))$
apply(*erule termi.induct*)
apply *simp-all*
apply *blast*
apply *blast*
done

lemma *termi-while*:
 $[\![\ (WHILE\ b\ DO\ c) \downarrow f\ k;\ \ \forall i.\ f\ i\ -c\rightarrow\ f(Suc\ i)\ ]\!] \Longrightarrow \exists i.\ \neg b(f\ i)$
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:

definition
hoare-tvalid :: *assn* ⇒ *com* ⇒ *assn* ⇒ *bool* (‹$\models_t$ {(*1*-)}/ (-)/ {(*1*-)}› *50*) **where**
$\models_t \{P\}c\{Q\} \longleftrightarrow \models \{P\}c\{Q\} \wedge (\forall s.\ P\ s \longrightarrow c{\downarrow}s)$

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* ⇒ *com* ⇒ *assn* ⇒ *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*: $[\![\vdash_t \{P\}c\{Q\}; \vdash_t \{Q\}d\{R\}]\!] \Longrightarrow \vdash_t \{P\}\ c;d\ \{R\}$
| *If*: $[\![\vdash_t \{\lambda s.\ P\ s \wedge b\ s\}c\{Q\}; \vdash_t \{\lambda s.\ P\ s \wedge {\sim}b\ s\}d\{Q\}]\!] \Longrightarrow$
$\vdash_t \{P\}\ IF\ b\ THEN\ c\ ELSE\ d\ \{Q\}$
| *While*:
$[\![wf\ r;\ \forall s'.\ \vdash_t \{\lambda s.\ P\ s \wedge b\ s \wedge s' = s\}\ c\ \{\lambda s.\ P\ s \wedge (s,s') \in r\}]\!]$
$\Longrightarrow \vdash_t \{P\}\ WHILE\ b\ DO\ c\ \{\lambda s.\ P\ s \wedge \neg b\ s\}$
| *Conseq*: $[\![\forall s.\ P'\ s \longrightarrow P\ s; \vdash_t \{P\}c\{Q\}; \forall s.\ Q\ s \longrightarrow Q'\ s]\!] \Longrightarrow$
$\vdash_t \{P'\}c\{Q'\}$

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

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*: *if-split-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:

definition
wpt :: *com* $\Rightarrow$ *assn* $\Rightarrow$ *assn* (‹wp_t›) **where**
$wp_t\ c\ Q = (\lambda s.\ wp\ c\ Q\ s \wedge c{\downarrow}s)$

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*]:
$wp_t\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ Q = (\lambda s.\ wp_t\ (if\ b\ s\ then\ c_1\ else\ c_2)\ Q\ s)$
apply(*unfold wp-defs*)
apply(*rule ext*)

apply *auto*
done

lemma [*simp*]: wp_t $(LOCAL\ f;c;g)\ Q = (\lambda s.\ wp_t\ c\ (Q\ o\ (g\ s))\ (f\ s))$
apply(*unfold wp-defs*)
apply(*rule ext*)
apply *auto*
done

lemma *strengthen-pre*: $[\![\ \forall s.\ P'\ s \longrightarrow P\ s;\ \vdash_t \{P\}c\{Q\}\ ]\!] \Longrightarrow\ \vdash_t \{P'\}c\{Q\}$
by(*erule thoare.Conseq*, *assumption*, *blast*)

lemma *weaken-post*: $[\![\ \vdash_t \{P\}c\{Q\};\ \forall s.\ Q\ s \longrightarrow Q'\ s\ ]\!] \Longrightarrow\ \vdash_t \{P\}c\{Q'\}$
apply(*rule thoare.Conseq*)
apply(*fast*, *assumption*, *assumption*)
done

inductive-cases [*elim*!]: $WHILE\ b\ DO\ c \downarrow s$

lemma *wp-is-pre*[*rule-format*]: $\vdash_t \{wp_t\ c\ Q\}\ c\ \{Q\}$
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(*fastforce 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 $\models_t \{P\}c\{Q\} \Longrightarrow\ \vdash_t \{P\}c\{Q\}$
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*

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

datatype *com* = *Do* (*state* ⇒ *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* ⇒ *state*) *com* (*state* ⇒ *state* ⇒ *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* ⇒ *com* ⇒ *state* ⇒ *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*: $⟦\ b\ s;\ s\ -c\rightarrow\ t;\ t\ -WHILE\ b\ DO\ c\rightarrow\ u\ ⟧$
$\Longrightarrow s\ -WHILE\ b\ DO\ c\rightarrow\ u$

| $s\ -body\rightarrow\ t \Longrightarrow s\ -CALL\rightarrow\ t$

| *Local*: $f\ s\ -c\rightarrow\ t \Longrightarrow s\ -LOCAL\ f;\ c;\ g\rightarrow\ g\ s\ t$

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

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*:*if-split-asm*)

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

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

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

lemma *WhileI*: $[\![b\ s;\ s\ {-}c{\rightarrow}\ t;\ t\ {-}WHILE\ b\ DO\ c{\rightarrow}\ u]\!] \Longrightarrow s\ {-}WHILE\ b\ DO\ c{\rightarrow}\ u$
by(*fastforce 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* ⇒ *com* ⇒ *nat* ⇒ *state* ⇒ *bool* (‹-/ -–-→/ -› [50,0,0,50] 50)
where
$t \in f\ s \Longrightarrow s\ {-}Do\ f{-}n{\rightarrow}\ t$

$|\ [\![\ s0\ {-}c1{-}n{\rightarrow}\ s1;\ s1\ {-}c2{-}n{\rightarrow}\ s2\ ]\!] \Longrightarrow s0\ {-}c1;c2{-}n{\rightarrow}\ s2$

$|\ [\![\ \ b\ s;\ s\ {-}c1{-}n{\rightarrow}\ t\ ]\!] \Longrightarrow s\ {-}IF\ b\ THEN\ c1\ ELSE\ c2{-}n{\rightarrow}\ t$
$|\ [\![\ \neg b\ s;\ s\ {-}c2{-}n{\rightarrow}\ t\ ]\!] \Longrightarrow s\ {-}IF\ b\ THEN\ c1\ ELSE\ c2{-}n{\rightarrow}\ t$

| $\neg b\ s \Longrightarrow s\ -WHILE\ b\ DO\ c-n\rightarrow\ s$
| $[\![b\ s;\ s\ -c-n\rightarrow\ t;\ t\ -WHILE\ b\ DO\ c-n\rightarrow\ u]\!] \Longrightarrow s\ -WHILE\ b\ DO\ c-n\rightarrow\ u$

| $s\ -body-n\rightarrow\ t \Longrightarrow s\ -CALL-Suc\ n\rightarrow\ t$

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

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*]: $s\ -c-m\rightarrow\ t \Longrightarrow \forall\, n.\ m \le n \longrightarrow s\ -c-n\rightarrow\ t$
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(*fastforce intro*:*exec.intros exec-mono simp add*:*max-def*)
apply *fastforce*
apply *fastforce*
apply(*blast intro*:*execn.intros*)
apply *clarify*
apply(*rename-tac m n*)
apply(*rule-tac x = max m n* **in** *exI*)
apply(*fastforce elim*:*execn.WhileTrue exec-mono simp add*:*max-def*)
apply *blast*
apply *blast*
apply(*erule exE*, *erule execn.induct*)
apply *blast*
apply *blast*
apply *fastforce*
apply *fastforce*
apply(*erule exec.WhileFalse*)
apply(*blast intro*: *exec.intros*)
apply *blast*
apply *blast*
done

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

lemma *while-rule*:
$[\![s \ -WHILE\ b\ DO\ c-n\rightarrow \ t;\ P\ s;\ \bigwedge s\ s'.\ [\![P\ s;\ b\ s;\ s \ -c-n\rightarrow \ s'\ ]\!] \Longrightarrow P\ s']\!]$
$\Longrightarrow P\ t \wedge \neg b\ t$
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 $'a$:

type-synonym $'a\ assn = 'a \Rightarrow state \Rightarrow bool$

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 *CALL*s 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:

type-synonym $'a\ cntxt = ('a\ assn \times com \times 'a\ assn)set$

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:

definition
$valid :: 'a\ assn \Rightarrow com \Rightarrow 'a\ assn \Rightarrow bool$ (‹$\models \{(1\text{-})\}/\ (\text{-})/\ \{(1\text{-})\}$› *50*) **where**
$\models \{P\}c\{Q\} \longleftrightarrow (\forall s\ t.\ s\ {-c\rightarrow}\ t \longrightarrow (\forall z.\ P\ z\ s \longrightarrow Q\ z\ t))$

Auxiliary variables are always denoted by z.

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

definition
$valids :: 'a\ cntxt \Rightarrow bool$ (‹$||\!\models$ -› *50*) **where**
[*simp*]: $||\!\models C \equiv (\forall (P,c,Q) \in C.\ \models \{P\}c\{Q\})$

definition
$cvalid :: 'a\ cntxt \Rightarrow 'a\ assn \Rightarrow com \Rightarrow 'a\ assn \Rightarrow bool$ (‹- $\models$/ $\{(1\text{-})\}/\ (\text{-})/\ \{(1\text{-})\}$› *50*) **where**
$C \models \{P\}c\{Q\} \longleftrightarrow ||\!\models C \longrightarrow \models \{P\}c\{Q\}$

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:

definition
$nvalid :: nat \Rightarrow 'a\ assn \Rightarrow com \Rightarrow 'a\ assn \Rightarrow bool$ (‹$\models$- $\{(1\text{-})\}/\ (\text{-})/\ \{(1\text{-})\}$› *50*) **where**

$\models n\ \{P\}c\{Q\} \equiv (\forall s\ t.\ s\ -c-n\rightarrow\ t \longrightarrow (\forall z.\ P\ z\ s \longrightarrow Q\ z\ t))$

definition
nvalids :: *nat* ⇒ *'a cntxt* ⇒ *bool* (‹||=′‐‐/ ‐› *50*) **where**
$||\!\!=\text{-}n\ C \equiv (\forall (P,c,Q) \in C.\ \models n\ \{P\}c\{Q\})$

definition
cnvalid :: *'a cntxt* ⇒ *nat* ⇒ *'a assn* ⇒ *com* ⇒ *'a assn* ⇒ *bool* (‹‐ |=‐/ {(*1*‐)}/ (‐)/ {(*1*‐)}› *50*) **where**
$C \models n\ \{P\}c\{Q\} \longleftrightarrow\ ||\!\!=\text{-}n\ C \longrightarrow\ \models n\ \{P\}c\{Q\}$

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

inductive
hoare :: *'a cntxt* ⇒ *'a assn* ⇒ *com* ⇒ *'a assn* ⇒ *bool* (‹‐ ⊢/ ({(*1*‐)}/ (‐)/ {(*1*‐)})› *50*)
where
$C \vdash \{\lambda z\ s.\ \forall t \in f\ s\ .\ P\ z\ t\}\ Do\ f\ \{P\}$

$|\ [\![\ C \vdash \{P\}c1\{Q\};\ C \vdash \{Q\}c2\{R\}\ ]\!] \Longrightarrow C \vdash \{P\}\ c1;c2\ \{R\}$

$|\ [\![\ C \vdash \{\lambda z\ s.\ P\ z\ s \wedge b\ s\}c1\{Q\};\ \ C \vdash \{\lambda z\ s.\ P\ z\ s \wedge \neg b\ s\}c2\{Q\}\ \ ]\!]$
$\Longrightarrow C \vdash \{P\}\ IF\ b\ THEN\ c1\ ELSE\ c2\ \{Q\}$

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

$|\ [\![\ C \vdash \{P'\}c\{Q'\};\ \ \forall s\ t.\ (\forall z.\ P'\ z\ s \longrightarrow Q'\ z\ t) \longrightarrow (\forall z.\ P\ z\ s \longrightarrow Q\ z\ t)\ ]\!]$
$\Longrightarrow C \vdash \{P\}c\{Q\}$

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

$|\ \{(P,CALL,Q)\} \vdash \{P\}\ CALL\ \{Q\}$
$|\ \ [\![\ \forall s'.\ C \vdash \{\lambda z\ s.\ P\ z\ s' \wedge s = f\ s'\}\ c\ \{\lambda z\ t.\ Q\ z\ (g\ s'\ t)\}\ ]\!] \Longrightarrow$
$C \vdash \{P\}\ LOCAL\ f;c;g\ \{Q\}$

abbreviation *hoare1* :: *'a cntxt* ⇒ *'a assn* × *com* × *'a assn* ⇒ *bool* (‹‐ ⊢ ‐›)
where
$C \vdash x \equiv C \vdash \{fst\ x\}fst\ (snd\ x)\{snd\ (snd\ x)\}$

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*:
$[\![\ \forall z\ s.\ P'\ z\ s \longrightarrow P\ z\ s;\ C\vdash\ \{P\}c\{Q\}\ \ ]\!] \Longrightarrow C\vdash\ \{P'\}c\{Q\}$
by(*rule hoare.Conseq*, *assumption*, *blast*)

lemmas *valid-defs* = *valid-def valids-def cvalid-def*
nvalid-def nvalids-def cnvalid-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*)
 apply *fast*
apply(*erule hoare.induct*)
        apply *simp*
       apply *fast*
      apply *simp*
     apply *clarify*
     apply(*drule 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*):

definition
  $MGT :: com \Rightarrow state\ assn \times com \times state\ assn$ **where**
  $MGT\ c = (\lambda z\ s.\ z = s,\ c,\ \lambda z\ t.\ z\ {-c\rightarrow}\ t)$

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* $P = \lambda z\ s.\ (z,s) \in (\{(s,t).\ b\ s \wedge s\ {-c\rightarrow}\ t\})\hat{}*$
   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(*fastforce 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* ⇒ *state* ⇒ *bool* (**infixl** ‹↓› *50*)
where
  Do[*iff*]: $f\ s \neq \{\} \implies Do\ f \downarrow s$
| *Semi*[*intro*!]: $[\![\ c1 \downarrow s0;\ \bigwedge s1.\ s0\ -c1\rightarrow\ s1 \implies c2 \downarrow s1\ ]\!]$
  $\implies (c1;c2) \downarrow s0$

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

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

| *WhileTrue*: $[\![\ b\ s;\ c \downarrow s;\ \bigwedge t.\ s\ -c\rightarrow\ t \implies WHILE\ b\ DO\ c \downarrow t\ ]\!]$
  $\implies WHILE\ b\ DO\ c \downarrow s$
| $body \downarrow s \implies CALL \downarrow s$

| *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*]: $(IF\ b\ THEN\ c1\ ELSE\ c2 \downarrow s) =$

```
    ((if b s then c1 else c2) ↓ 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 ↓ s) = (body ↓ s)
by(fast elim: termi.cases intro:termi.intros)

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

lemma termi-while-lemma[rule-format]:
 w↓fk ⟹
 (∀ k b c. fk = f k ∧ w = WHILE b DO c ∧ (∀ i. f i −c→ f(Suc i))
        ⟶ (∃ i. ¬b(f i)))
apply(erule termi.induct)
apply simp-all
apply blast
apply blast
done

lemma termi-while:
 ⟦ (WHILE b DO c) ↓ f k;  ∀ i. f i −c→ f(Suc i) ⟧ ⟹ ∃ i. ¬b(f i)
by(blast intro:termi-while-lemma)

lemma wf-termi: wf {(t,s). WHILE b DO c ↓ s ∧ b s ∧ s −c→ 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:

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

definition
$ctvalid :: 'a\ cntxt \Rightarrow 'a\ assn \Rightarrow com \Rightarrow 'a\ assn \Rightarrow bool$
(‹(- /$\models_t$ {(1-)}/ (-)/ {(1-)})› 50) **where**
$C \models_t \{P\}c\{Q\} \longleftrightarrow (\forall (P',c',Q') \in C.\ \models_t \{P'\}c'\{Q'\}) \longrightarrow \models_t \{P\}c\{Q\}$

inductive
$thoare :: 'a\ cntxt \Rightarrow 'a\ assn \Rightarrow com \Rightarrow 'a\ assn \Rightarrow bool$
(‹(- $\vdash_t$/ ({(1-)}/ (-)/ {(1-)}))› [50,0,0,0] 50)
where
Do: $C \vdash_t \{\lambda z\ s.\ (\forall t \in f\ s\ .\ P\ z\ t) \wedge f\ s \neq \{\}\}\ Do\ f\ \{P\}$
| $Semi$: $[\![\ C \vdash_t \{P\}c1\{Q\};\ C \vdash_t \{Q\}c2\{R\}\ ]\!] \Longrightarrow C \vdash_t \{P\}\ c1;c2\ \{R\}$
| If: $[\![\ C \vdash_t \{\lambda z\ s.\ P\ z\ s \wedge b\ s\}c\{Q\};\ C \vdash_t \{\lambda z\ s.\ P\ z\ s \wedge \sim b\ s\}d\{Q\}\ \ ]\!] \Longrightarrow$
$C \vdash_t \{P\}\ IF\ b\ THEN\ c\ ELSE\ d\ \{Q\}$
| $While$:
$[\![wf\ r;\ \ \forall s'.\ C \vdash_t \{\lambda z\ s.\ P\ z\ s \wedge b\ s \wedge s' = s\}\ c\ \{\lambda z\ s.\ P\ z\ s \wedge (s,s') \in r\}]\!]$
$\Longrightarrow C \vdash_t \{P\}\ WHILE\ b\ DO\ c\ \{\lambda z\ s.\ P\ z\ s \wedge \neg b\ s\}$

| $Call$:
$[\![wf\ r;\ \ \forall s'.\ \{(\lambda z\ s.\ P\ z\ s \wedge (s,s') \in r,\ CALL,\ Q)\}$
$\vdash_t \{\lambda z\ s.\ P\ z\ s \wedge s = s'\}\ body\ \{Q\}]\!]$
$\Longrightarrow \{\} \vdash_t \{P\}\ CALL\ \{Q\}$

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

| $Conseq$:
$[\![\ C \vdash_t \{P'\}c\{Q'\};$
$(\forall s\ t.\ (\forall z.\ P'\ z\ s \longrightarrow Q'\ z\ t) \longrightarrow (\forall z.\ P\ z\ s \longrightarrow Q\ z\ t)) \wedge$
$(\forall s.\ (\exists z.\ P\ z\ s) \longrightarrow (\exists z.\ P'\ z\ s))\ ]\!]$
$\Longrightarrow C \vdash_t \{P\}c\{Q\}$

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

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:

lemma $((\forall s\ t.\ (\forall z.\ P'\ z\ s \longrightarrow Q'\ z\ t) \longrightarrow (\forall z.\ P\ z\ s \longrightarrow Q\ z\ t)) \wedge$
$(\forall s.\ (\exists z.\ P\ z\ s) \longrightarrow (\exists z.\ P'\ z\ s)))$

$= (\forall z\ s.\ P\ z\ s \longrightarrow (\forall t.\exists z'.\ P'\ z'\ s \wedge (Q'\ z'\ t \longrightarrow Q\ z\ t)))$
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*:
$[\![\ \forall z\ s.\ P'\ z\ s \longrightarrow P\ z\ s;\ C \vdash_t \{P\}c\{Q\}\ ]\!] \Longrightarrow C \vdash_t \{P'\}c\{Q\}$
by(*rule thoare.Conseq*, *assumption*, *blast*)

lemma *weaken-post*:
$[\![\ C \vdash_t \{P\}c\{Q\};\ \forall z\ s.\ Q\ z\ s \longrightarrow Q'\ z\ s\ ]\!] \Longrightarrow C \vdash_t \{P\}c\{Q'\}$
by(*erule thoare.Conseq*, *blast*)

lemmas *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 $C \vdash_t \{P\}c\{Q\} \implies C \models_t \{P\}c\{Q\}$
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*: *if-split-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* $\forall n.\ \models_t \{\lambda z\ s.\ P\ z\ s \wedge s{=}n\}\ body\ \{Q\}$)
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

definition MGT_t :: $com \Rightarrow state\ assn \times com \times state\ assn$ **where**
[*simp*]: $MGT_t\ c = (\lambda z\ s.\ z = s \wedge c{\downarrow}s,\ c,\ \lambda z\ t.\ z\ {-c\rightarrow}\ t)$

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*: $[\![\ WHILE\ b\ DO\ c \downarrow s;\ b\ s\ ]\!] \Longrightarrow c \downarrow s$
by(*erule termi.cases*, *auto*)

lemma *while-termiE2*:
$[\![\ WHILE\ b\ DO\ c \downarrow s;\ b\ s;\ s\ {-c\rightarrow}\ t\ ]\!] \Longrightarrow WHILE\ b\ DO\ c \downarrow t$
by(*erule termi.cases*, *auto*)

lemma *MGT-lemma*: $C \vdash_t MGT_t\ CALL \Longrightarrow C \vdash_t MGT_t\ c$
apply (*simp*)
apply(*induct-tac c*)
apply (*rule strengthen-pre*[*OF - thoare.Do*])
apply *blast*
apply(*rename-tac com1 com2*)
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* $P' = \lambda z\ s.\ (z,s) \in (\{(s,t).\ b\ s \wedge s\ -c\rightarrow\ t\})\hat{}* \wedge$
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(*fastforce 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* $\times$ *state*) $\times$ (*com list* $\times$ *state*))*set*
and *exec1'* :: (*com list* $\times$ *state*) $\Rightarrow$ (*com list* $\times$ *state*) $\Rightarrow$ *bool* (‹- → -› [*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: ¬b s ⟹ ((IF b THEN c1 ELSE c2)#cs,s) → (c2#cs,s)

| WhileFalse: ¬b s ⟹ ((WHILE b DO c)#cs,s) → (cs,s)
| WhileTrue:   b s ⟹ ((WHILE b DO c)#cs,s) → (c#(WHILE b DO c)#cs,s)

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

| Local[iff]: ((LOCAL f;c;g)#cs,s) → (c # Do(λt. {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#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→ 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(*fastforce intro*:*execs.intros*)
apply(*fastforce 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*)

primrec *termis* :: *com list* ⇒ *state* ⇒ *bool* (**infixl** ‹⇓› *60*) **where**
  []⇓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*: ⟦ ([c],s) →* (c'#cs',s'); c↓s ⟧ ⟹ c'↓s'
apply(*insert execs-pres-termis*[*of* [c] - c'#cs',*simplified*])

apply *blast*
done

definition
termi-call-steps :: $(state \times state)set$ **where**
$termi\text{-}call\text{-}steps = \{(t,s).\ body{\downarrow}s \wedge (\exists\, cs.\ ([body],\ s) \rightarrow^* (CALL\ \#\ cs,\ t))\}$

lemma *lem*:
$\forall\, y.\ (a,y){\in}r^+ \longrightarrow P\ a \longrightarrow P\ y \Longrightarrow ((b,a) \in \{(y,x).\ P\ x \wedge (x,y){:}r\}^+) = ((b,a) \in \{(y,x).\ P\ x \wedge (x,y){\in}r^+\})$
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*:
$[\![\forall\, i.\ (a,f\ i) : r\hat{}{*} \wedge (f\ i,f(Suc\ i)) : r;\ (a,b) : r\hat{}{*}\ ]\!] \Longrightarrow b = f\ 0 \longrightarrow (\exists f.\ f\ 0 = a\ \&\ (\forall\, i.\ (f\ i,\ f(Suc\ i)) : r))$
apply(*erule converse-rtrancl-induct*)
apply *blast*
apply(*clarsimp*)
apply(*rule-tac* $x{=}\lambda i.\ case\ i\ of\ 0 \Rightarrow y \mid Suc\ i \Rightarrow fa\ i$ **in** *exI*)
apply *simp*
apply *clarify*
apply(*case-tac i*)
apply *simp-all*
done

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

definition *inf* :: *com list* $\Rightarrow$ *state* $\Rightarrow$ *bool* **where**
$inf\ cs\ s \longleftrightarrow (\exists f.\ f\ 0 = (cs,s) \wedge (\forall\, i.\ f\ i \rightarrow f(Suc\ i)))$

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

lemma [*iff*]: ¬ *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*]: *inf* ((*c1*;*c2*)#*cs*) *s* = *inf* (*c1*#*c2*#*cs*) *s*
apply(*unfold inf-def*)
apply(*rule iffI*)
apply *clarify*
apply(*rule-tac x* = λ*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* = λ*i*. *case i of 0* ⇒ ((*c1*;*c2*)#*cs*,*s*) | *Suc i* ⇒ *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* = λ*i*. *f*(*Suc i*) **in** *exI*)
apply *clarsimp*
apply *clarsimp*
apply(*rule-tac x* = λ*i*. *f*(*Suc i*) **in** *exI*)
apply *clarsimp*
apply *clarsimp*
apply(*rule-tac x* = λ*i*. *case i of 0* ⇒ ((*IF b THEN c1 ELSE c2*)#*cs*,*s*) | *Suc i* ⇒ *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* = λ*i*. *f*(*Suc i*) **in** *exI*)
apply *clarsimp*
apply *clarsimp*
apply(*rule-tac x* = λ*i*. *f*(*Suc i*) **in** *exI*)
apply *clarsimp*
apply (*clarsimp split*:*if-splits*)
apply(*rule-tac x* = λ*i*. *case i of 0* ⇒ ((*WHILE b DO c*)#*cs*,*s*) | *Suc i* ⇒ *f i* **in** *exI*)
apply(*simp add*: *exec1.intros split*:*nat.split*)
apply(*rule-tac x* = λ*i*. *case i of 0* ⇒ ((*WHILE b DO c*)#*cs*,*s*) | *Suc i* ⇒ *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* = λ*i*. *f*(*Suc i*) **in** *exI*)
apply *clarsimp*
apply *clarsimp*
apply(*rule-tac x* = λ*i*. *case i of 0* ⇒ (*CALL*#*cs*,*s*) | *Suc i* ⇒ *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*(λ*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* = λ*i*. *F*(*Suc i*) **in** *exI*)
apply *clarsimp*
apply (*clarsimp*)
apply(*rename-tac F*)
apply(*rule-tac x* = λ*i*. *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*: $(ccs,s) \rightarrow (cs',t) \Longrightarrow$
$\forall c\ cs.\ ccs = c\#cs \longrightarrow (\exists cs1.\ cs' = cs1\ @\ cs)$
apply(*erule exec1.induct*)
apply *blast*
apply *force+*
done

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

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

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

lemma *execs-drop-suffix*[*rule-format*(*no-asm*)]:
$[\![\ f\ 0 = (c\#cs,s);\forall i.\ f(i) \rightarrow f(Suc\ i)\ ]\!] \Longrightarrow$
$(\forall i{<}k.\ p\ i \neq []\ \&\ fst(f\ i) = p\ i@cs) \longrightarrow fst(f\ k) = p\ k@cs$
$\longrightarrow ([c],s) \rightarrow^* (p\ k,snd(f\ k))$
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*:
$[\![\ f\ 0 = (c\#cs,s);\forall i.\ f(i) \rightarrow f(Suc\ i);\ \forall i{<}k.\ p\ i \neq []\ \&\ fst(f\ i) = p\ i@cs;$
$fst(f\ k) = cs;\ p\ k = []\ ]\!] \Longrightarrow ([c],s) \rightarrow^* ([],snd(f\ k))$
apply(*drule execs-drop-suffix*,*assumption*,*assumption*)
apply *simp*
apply *simp*
done

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

lemma *least-aux*: $[\![f\ 0 = (c \# cs,\ s);\ \forall\, i.\ f\ i \to f\ (Suc\ i);$
$\quad fst(f\ k) = cs;\ \forall\, i{<}k.\ fst(f\ i) \neq cs]\!]$
$\quad \Longrightarrow \forall\, i \leq k.\ (\exists\, p.\ (p \neq []) = (i < k)\ \&\ fst(f\ i) = p\ @\ cs)$
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*: $[\![f\ 0 = (c\#cs,s);\ \forall\, i.\ f\ i \to f(Suc\ i);\ \exists\, i.\ fst(f\ i) = cs\ ]\!]$
$\quad \Longrightarrow \exists\, k.\ fst(f\ k) = cs\ \&\ ([c],s) \to^* ([],snd(f\ k))$
apply(*rule-tac x=LEAST i. fst(f i) = cs* **in** *exI*)
apply(*rule conjI*)
 apply(*fast intro*: *LeastI*)
apply(*subgoal-tac*
 $\forall\, i{\leq}LEAST\ i.\ fst\ (f\ i) = cs.\ \exists\, p.\ ((p \neq []) = (i{<}(LEAST\ i.\ fst\ (f\ i) = cs)))\ \&$
$fst(f\ i) = p@cs$)
 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*: $\forall x.\exists y.\ P\ x\ y \Longrightarrow \exists f.\forall x.\ P\ x\ (f\ x)$
apply(*rule-tac* $x = \lambda x.$ *SOME* $y.\ P\ x\ y$ **in** *exI*)
apply(*fast intro*:*someI2*)
done

lemma *inf-cases*: *inf* $(c\#cs)\ s \Longrightarrow$ *inf* $[c]\ s \lor (\exists t.\ s\ {-c\rightarrow}\ t \land$ *inf cs* $t)$
apply(*unfold inf-def*)
apply (*clarsimp del*: *disjCI*)
apply(*case-tac* $\exists i.$ *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* $\forall i.\ \exists p.\ p \neq [] \land$ *fst*$(f\ i) = p@cs$)
 apply(*drule skolemize2*)
 apply *clarify*
 apply(*rename-tac p*)
 apply(*rule-tac* $x = \lambda i.\ (p\ i,$ *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*: $c \downarrow s \Longrightarrow \neg$ *inf* $[c]$ s
apply(*erule termi.induct*)

apply *clarify*

apply(*blast dest*:*inf-cases*)

apply *clarsimp*
apply *clarsimp*

apply *clarsimp*
apply(*fastforce dest*:*inf-cases*)

apply *blast*

apply(*blast dest*:*inf-cases*)
done

lemma *termi-impl-no-inf-chain*:
$c{\downarrow}s \Longrightarrow \neg(\exists f.\ f\ 0 = ([c],s) \wedge (\forall i{::}nat.\ (f\ i,\ f(i+1)) : exec1\hat{}+))$
apply(*subgoal-tac* $wf(\{(y,x).\ ([c],s) \rightarrow^* x\ \&\ x \rightarrow y\}\hat{}+)$)
apply(*simp only*:*wf-iff-no-infinite-down-chain*)
apply(*erule contrapos-nn*)
apply *clarify*
apply(*subgoal-tac* $\forall i.\ ([c],\ s) \rightarrow^* f\ i$)
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

primrec *cseq* :: (*nat* ⇒ *state*) ⇒ *nat* ⇒ *com list* **where**
  cseq S 0 = []
| *cseq S* (*Suc i*) = (*SOME cs.* ([*body*], *S i*) →* (*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* ∃ *Cs. Cs 0* = [] & (∀ *i.* (*body* # *Cs i*,*S i*) →* (*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* = *λcs.*([*body*],*S i*) →* (*CALL* # *cs*, *S*(*Suc i*)) **in** *someI2*)
 apply(*fastforce dest*:*app-execs*)
apply *clarify*
apply(*subgoal-tac* ∀ *i.* ((*body* # *Cs i*,*S i*), (*body* # *Cs*(*i*+*1*), *S*(*i*+*1*))) : *exec1*^+)
 prefer *2*
 apply(*blast intro*:*rtrancl-into-trancl1*)
apply(*subgoal-tac* ∃*f. f 0* = ([*body*],*S 0*) ∧ (∀ *i.* (*f i*, *f*(*i*+*1*)) : *exec1*^+))
 prefer *2*
 apply(*rule-tac x* = *λi.*(*body*#*Cs i*,*S i*) **in** *exI*)
 apply *blast*
apply(*blast dest*:*termi-impl-no-inf-chain*)
done

lemma *CALL-lemma*:
{(*λz s.* (*z*=*s* ∧ *body*↓*s*) ∧ (*s*,*t*) ∈ *termi-call-steps*, *CALL*, *λz s. z* −*body*→ *s*)} $\vdash_t$
 {*λz s.* (*z*=*s* ∧ *body*↓*t*) ∧ (∃ *cs.* ([*body*],*t*) →* (*c*#*cs*,*s*))} *c* {*λz s. z* −*c*→ *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* = *λz s. body*↓*t* & (∃ *cs.* ([*body*], *t*) →* (*c2*#*cs*,*s*)) & *z* −*c1*→*s* & *c2*↓*s* **in** *thoare.Semi*)
    apply(*erule thoare.Conseq*)

```
     apply(rule conjI)
      apply clarsimp
      apply(subgoal-tac s −c1→ ta)
       prefer 2
       apply(blast intro: exec1.Semi exec-impl-execs rtrancl-trans)
      apply(subgoal-tac ([body], t) →* (c2 # cs, ta))
       prefer 2
     apply(blast intro:exec1.Semi[THEN r-into-rtrancl] exec-impl-execs rtrancl-trans)
      apply(subgoal-tac ([body], t) →* (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' = λz s. (z,s) ∈ ({(s,t). b s ∧ s −c→ t})^* ∧ body ↓ t ∧
         (∃ cs. ([body], t) →* ((WHILE b 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*:
$\{(\lambda z\ s.\ (z{=}s \wedge body{\downarrow}s) \wedge (s,t) \in termi\text{-}call\text{-}steps,\ CALL,\ \lambda z\ s.\ z\ {-}body{\rightarrow}\ s)\} \vdash_t$
$\{\lambda z\ s.\ (z{=}s \wedge body{\downarrow}s) \wedge s = t\}\ body\ \{\lambda z\ s.\ z\ {-}body{\rightarrow}\ s\}$
apply(*rule strengthen-pre*[*OF - CALL-lemma*])
apply *blast*
done

lemma *MGT-CALL*: $\{\} \vdash_t MGT_t\ CALL$
apply(*simp add*: MGT_t*-def*)
apply(*blast intro*:*thoare.Call wf-termi-call-steps CALL-cor*)
done

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

type-synonym $bexp = state \Rightarrow bool$

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

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
$exec :: state \Rightarrow com \Rightarrow state \Rightarrow bool \quad (\langle\text{-/ --}\rightarrow\text{/ -}\rangle\ [50,0,50]\ 50)$
where
$\quad Do: \quad t \in f\ s \Longrightarrow s\ -Do\ f\rightarrow\ t$

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

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

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

$|\ Call: s\ -body\ p\rightarrow\ t \Longrightarrow s\ -CALL\ p\rightarrow\ t$

$|\ Local: f\ s\ -c\rightarrow\ t \Longrightarrow s\ -LOCAL\ f;\ c;\ g\rightarrow\ g\ s\ t$

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

lemma [*iff*]: (*s* −*c*;*d*→ *u*) = (∃ *t*. *s* −*c*→ *t* ∧ *t* −*d*→ *u*)
by(*auto elim*: *exec.cases intro*:*exec.intros*)

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

lemma [*iff*]: (*s* −*CALL p*→ *t*) = (*s* −*body p*→ *t*)
by(*blast elim*: *exec.cases intro*:*exec.intros*)

lemma [*iff*]: (*s* −*LOCAL f*; *c*; *g*→ *u*) = (∃ *t*. *f s* −*c*→ *t* ∧ *u* = *g s t*)
by(*fastforce elim*: *exec.cases intro*:*exec.intros*)

inductive
 execn :: *state* ⇒ *com* ⇒ *nat* ⇒ *state* ⇒ *bool* (‹-/ −-−-→/ -› [*50*,*0*,*0*,*50*] *50*)
where
 Do: *t* ∈ *f s* ⟹ *s* −*Do f*−*n*→ *t*

 | *Semi*: ⟦ *s0* −*c0*−*n*→ *s1*; *s1* −*c1*−*n*→ *s2* ⟧ ⟹ *s0* −*c0*;*c1*−*n*→ *s2*

 | *IfTrue*: ⟦ *b s*; *s* −*c0*−*n*→ *t* ⟧ ⟹ *s* −*IF b THEN c0 ELSE c1*−*n*→ *t*

 | *IfFalse*: ⟦ ¬*b s*; *s* −*c1*−*n*→ *t* ⟧ ⟹ *s* −*IF b THEN c0 ELSE c1*−*n*→ *t*

 | *WhileFalse*: ¬*b s* ⟹ *s* −*WHILE b DO c*−*n*→ *s*

 | *WhileTrue*: ⟦ *b s*; *s* −*c*−*n*→ *t*; *t* −*WHILE b DO c*−*n*→ *u* ⟧
 ⟹ *s* −*WHILE b DO c*−*n*→ *u*

 | *Call*: *s* −*body p*−*n*→ *t* ⟹ *s* −*CALL p*−*Suc n*→ *t*

 | *Local*: *f s* −*c*−*n*→ *t* ⟹ *s* −*LOCAL f*; *c*; *g*−*n*→ *g s t*

lemma [*iff*]: (*s* −*Do f*−*n*→ *t*) = (*t* ∈ *f s*)
by(*auto elim*: *execn.cases intro*:*execn.intros*)

lemma [*iff*]: (*s* −*c1*;*c2*−*n*→ *u*) = (∃ *t*. *s* −*c1*−*n*→ *t* ∧ *t* −*c2*−*n*→ *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*]: $(s \; -CALL \; p-Suc \; n\rightarrow \; t) = (s \; -body \; p-n\rightarrow \; t)$
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*)

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

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(*fastforce intro*:*exec.intros exec-mono simp add*:*max-def*)
apply *fastforce*
apply *fastforce*
apply(*blast intro*:*execn.intros*)
apply *clarify*
apply(*rename-tac m n*)
apply(*rule-tac x = max m n* **in** *exI*)
apply(*fastforce elim*:*execn.WhileTrue exec-mono simp add*:*max-def*)
apply *blast*
apply *blast*
apply(*erule exE*, *erule execn.induct*)
apply *blast*
apply *blast*
apply *fastforce*
apply *fastforce*
apply(*erule exec.WhileFalse*)
apply(*blast intro*: *exec.intros*)

apply *blast*
apply *blast*
done

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

lemma *while-rule*:
$[\![s\ -\mathit{WHILE}\ b\ \mathit{DO}\ c-n\rightarrow\ t;\ P\ s;\ \bigwedge s\ s'.\ [\![P\ s;\ b\ s;\ s\ -c-n\rightarrow\ s'\ ]\!] \Longrightarrow P\ s']\!]$
$\Longrightarrow P\ t \wedge \neg b\ t$
apply(*drule while-lemma*)
prefer *2* **apply** *assumption*
apply *blast*
done

end

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

4.2 Hoare logic for partial correctness

type-synonym $'a\ assn = 'a \Rightarrow state \Rightarrow bool$
type-synonym $'a\ cntxt = ('a\ assn \times com \times 'a\ assn)set$

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

definition
valids :: $'a\ cntxt \Rightarrow bool$ (‹|⊨ -› *50*) **where**
$|\models D \equiv (\forall\, (P,c,Q) \in D.\ \models \{P\}c\{Q\})$

definition
nvalid :: $nat \Rightarrow 'a\ assn \Rightarrow com \Rightarrow 'a\ assn \Rightarrow bool$ (‹⊨- {(1-)}/ (-)/ {(1-)}› *50*)
where
$\models n\ \{P\}c\{Q\} \equiv (\forall\, s\ t\ z.\ s\ -c-n\rightarrow\ t \longrightarrow P\ z\ s \longrightarrow Q\ z\ t)$

definition
nvalids :: $nat \Rightarrow 'a\ cntxt \Rightarrow bool$ (‹|⊨'--/ -› *50*) **where**

$|\models$-$n\ C \equiv (\forall (P,c,Q) \in C.\ \models n\ \{P\}c\{Q\})$

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:

definition
cvalids :: *'a cntxt* ⇒ *'a cntxt* ⇒ *bool* (‹- |⊨/ -› *50*) **where**
$C\ |\models D \longleftrightarrow\ |\models C \longrightarrow\ |\models D$

definition
cnvalids :: *'a cntxt* ⇒ *nat* ⇒ *'a cntxt* ⇒ *bool* (‹- |⊨'--/ -› *50*) **where**
$C\ |\models$-$n\ D \longleftrightarrow\ |\models$-$n\ C \longrightarrow\ |\models$-$n\ D$

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
hoare :: *'a cntxt* ⇒ *'a cntxt* ⇒ *bool* (‹- ⊩/ -› *50*)
and *hoare3* :: *'a cntxt* ⇒ *'a assn* ⇒ *com* ⇒ *'a assn* ⇒ *bool* (‹- ⊢/ ({(1-)}/ (-)/ {(1-)})› *50*)
where
$C \vdash \{P\}c\{Q\}\ \equiv\ C \Vdash \{(P,c,Q)\}$
| *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*: $[\![\ C \Vdash D;\ (P,c,Q) \in D\ ]\!] \Longrightarrow C \vdash \{P\}c\{Q\}$

| *Local*: $[\![\ \forall s'.\ C \vdash \{\lambda z\ s.\ P\ z\ s' \wedge s = f\ s'\}\ c\ \{\lambda z\ t.\ Q\ z\ (g\ s'\ t)\}\ ]\!] \Longrightarrow$
$C \vdash \{P\}\ LOCAL\ f;c;g\ \{Q\}$
monos *split-beta*

lemmas *valid-defs* = *valid-def valids-def cvalids-def*
nvalid-def nvalids-def cnvalids-def

theorem $C \Vdash D \implies C \mid\models D$

As before, we prove a generalization of $C \mid\models D$, namely $\forall n.\ C \mid\models\text{-}n\ D$, by induction on $C \Vdash D$, with an induction on n in the *CALL* case.

apply(*subgoal-tac* $\forall n.\ C \mid\models\text{-}n\ D$)
apply(*unfold valid-defs exec-iff-execn*[*THEN eq-reflection*])
 apply *fast*
apply(*erule hoare.induct*)
      apply *simp*
     apply *simp*
     apply *fast*
    apply *simp*
   apply *clarify*
   apply(*drule while-rule*)
   prefer *3*
   apply (*assumption*, *assumption*)
   apply *simp*
  apply *simp*
  apply *fast*
 apply(*rule allI*, *rule impI*)
 apply(*induct-tac n*)
  apply *force*
 apply *clarify*
 apply(*frule bspec*, *assumption*)
 apply (*simp*(*no-asm-use*))
 apply *fast*
apply *simp*
apply *fast*

apply *simp*
apply *fast*

apply *fast*

apply *fastforce*
done

definition *MGT* :: *com* $\Rightarrow$ *state assn* $\times$ *com* $\times$ *state assn* **where**
 [*simp*]: $MGT\ c = (\lambda z\ s.\ z = s,\ c,\ \lambda z\ t.\ z\ {-c\rightarrow}\ t)$

lemma *strengthen-pre*:
$[\![\ \forall z\ s.\ P'\ z\ s \longrightarrow P\ z\ s;\ C\vdash \{P\}c\{Q\}\ ]\!] \Longrightarrow C\vdash \{P'\}c\{Q\}$
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*: $\forall p.\ C \Vdash \{MGT(CALL\ p)\} \implies C \Vdash \{MGT\ c\}$
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* $P = \lambda z\ s.\ (z,s) \in (\{(s,t).\ b\ s \wedge s\ {-c\rightarrow}\ t\})\hat{}*$
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(*fastforce intro*: *hoare.Local elim*!: *hoare.Conseq*)
done

lemma *MGT-body*: $(P,\ CALL\ p,\ Q) = MGT\ (CALL\ pa) \implies C \Vdash \{MGT\ (body\ p)\} \implies C \vdash \{P\}\ body\ p\ \{Q\}$
apply *clarsimp*
done

declare *MGT-def*[*simp del*]

lemma *MGT-CALL*: $\{\} \Vdash \{mgt.\ \exists p.\ mgt = MGT(CALL\ p)\}$
apply (*rule hoare.Call*)
 apply(*fastforce 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 fun-eq-iff*)
done

end

theory *PsTermi* **imports** *PsLang* **begin**

4.3 Termination

inductive
  termi :: *com* ⇒ *state* ⇒ *bool* (**infixl** ‹↓› *50*)
where
    Do[*iff*]: $f\ s \neq \{\} \implies Do\ f \downarrow s$
  | *Semi*[*intro*!]: $[\![\ c1 \downarrow s0;\ \bigwedge s1.\ s0\ -c1\rightarrow\ s1 \implies c2 \downarrow s1\ ]\!]$
          $\implies (c1;c2) \downarrow s0$

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

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

  | *WhileTrue*: $[\![\ b\ s;\ c \downarrow s;\ \bigwedge t.\ s\ -c\rightarrow\ t \implies WHILE\ b\ DO\ c \downarrow t\ ]\!]$
          $\implies WHILE\ b\ DO\ c \downarrow s$

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

  | *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*]:
$w{\downarrow}fk \Longrightarrow$
$(\forall k\ b\ c.\ fk = f\ k \wedge w = WHILE\ b\ DO\ c \wedge (\forall i.\ f\ i\ -c\rightarrow\ f(Suc\ i))$
$\longrightarrow (\exists i.\ \neg b(f\ i)))$
apply(*erule termi.induct*)
apply *simp-all*
apply *blast*
apply *blast*
done

lemma *termi-while*:

$[\![$ $(WHILE\ b\ DO\ c) \downarrow f\ k;\ \ \forall i.\ f\ i\ -c\rightarrow f(Suc\ i)$ $]\!] \Longrightarrow \exists i.\ \neg b(f\ i)$
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 *PsHoareTotal* **imports** *PsHoare PsTermi* **begin**

4.4 Hoare logic for total correctness

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

definition
$valids :: 'a\ cntxt \Rightarrow bool$ (‹$|\models_t$ -› 50) **where**
$|\models_t D \longleftrightarrow (\forall (P,c,Q) \in D.\ \models_t \{P\}c\{Q\})$

definition
$ctvalid :: 'a\ cntxt \Rightarrow 'a\ assn \Rightarrow com \Rightarrow 'a\ assn \Rightarrow bool$
(‹(- /$\models_t$ {(1-)}/ (-)/ {(1-)})› 50) **where**
$C \models_t \{P\}c\{Q\} \longleftrightarrow |\models_t C \longrightarrow\ \models_t \{P\}c\{Q\}$

definition
$cvalids :: 'a\ cntxt \Rightarrow 'a\ cntxt \Rightarrow bool$ (‹- $|\models_t$/ -› 50) **where**
$C\ |\models_t D \longleftrightarrow |\models_t C \longrightarrow |\models_t D$

inductive
$thoare :: 'a\ cntxt \Rightarrow 'a\ cntxt \Rightarrow bool$ (‹(- $|\vdash_t$/ -)› 50)
and $thoare' :: 'a\ cntxt \Rightarrow 'a\ assn \Rightarrow com \Rightarrow 'a\ assn \Rightarrow bool$
(‹(- $\vdash_t$/ ({(1-)}/ (-)/ {(1-)}))› [50,0,0,0] 50)
where
$C \vdash_t \{P\}c\{Q\}\ \equiv\ C\ |\vdash_t \{(P,c,Q)\}$
| *Do*: $C \vdash_t \{\lambda z\ s.\ (\forall t \in f\ s\ .\ P\ z\ t) \wedge f\ s \neq \{\}\}\ Do\ f\ \{P\}$
| *Semi*: $[\![\ C \vdash_t \{P\}c1\{Q\};\ C \vdash_t \{Q\}c2\{R\}\ ]\!] \Longrightarrow C \vdash_t \{P\}\ c1;c2\ \{R\}$
| *If*: $[\![\ C \vdash_t \{\lambda z\ s.\ P\ z\ s \wedge b\ s\}c\{Q\};\ C \vdash_t \{\lambda z\ s.\ P\ z\ s \wedge {\sim}b\ s\}d\{Q\}\ \ ]\!] \Longrightarrow$
$C \vdash_t \{P\}\ IF\ b\ THEN\ c\ ELSE\ d\ \{Q\}$
| *While*:
$[\![wf\ r;\ \ \forall s'.\ C \vdash_t \{\lambda z\ s.\ P\ z\ s \wedge b\ s \wedge s' = s\}\ c\ \{\lambda z\ s.\ P\ z\ s \wedge (s,s') \in r\}]\!]$
$\Longrightarrow C \vdash_t \{P\}\ WHILE\ b\ DO\ c\ \{\lambda z\ s.\ P\ z\ s \wedge \neg b\ s\}$

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

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

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

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

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

lemma *strengthen-pre*:
$\llbracket\ \forall\, z\ s.\ P'\ z\ s \longrightarrow P\ z\ s;\ C \vdash_t \{P\}c\{Q\}\ \rrbracket \Longrightarrow C \vdash_t \{P'\}c\{Q\}$
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*)

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

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*:*if-split-asm*)

theorem $C \mid\vdash_t D \implies C \mid\models_t D$
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*: *if-split-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 ctvalid-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

definition MGT_t :: *com* $\Rightarrow$ *state assn* $\times$ *com* $\times$ *state assn* **where**
 [*simp*]: $MGT_t\ c = (\lambda z\ s.\ z = s \wedge c{\downarrow}s,\ c,\ \lambda z\ t.\ z\ {-c\rightarrow}\ t)$

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

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

lemma *while-termiE2*: $[\![$ *WHILE b DO c* $\downarrow s$; $b\ s$; $s\ {-c\rightarrow}\ t$ $]\!] \Longrightarrow$ *WHILE b DO c* $\downarrow t$
by(*erule termi.cases*, *auto*)

lemma *MGT-lemma*: $\forall p.\ \{\} \mathrel{|\!\!\vdash_t} \{MGT_t(CALL\ p)\} \implies \{\} \mathrel{|\!\!\vdash_t} \{MGT_t\ c\}$
apply (*simp*)
apply(*induct-tac c*)
  apply (*rule strengthen-pre*[*OF - thoare.Do*])
  apply *blast*
 apply(*rename-tac com1 com2*)
 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' = 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(fastforce 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 × 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 ∈ f s ⟹ ((Do f)#cs,s) → (cs,t)

| Semi[iff]: ((c1;c2)#cs,s) → (c1#c2#cs,s)

| IfTrue:   b s ⟹ ((IF b THEN c1 ELSE c2)#cs,s) → (c1#cs,s)
| IfFalse: ¬b s ⟹ ((IF b THEN c1 ELSE c2)#cs,s) → (c2#cs,s)

| WhileFalse: ¬b s ⟹ ((WHILE b DO c)#cs,s) → (cs,s)
| WhileTrue:   b s ⟹ ((WHILE b DO c)#cs,s) → (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(λt. {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*→ *t* ⟹ *t* =*cs*⇒ *u* ⟹ *s* =*c*#*cs*⇒ *u*

inductive-cases [*elim*!]:
s =[]⇒ *t*
s =*c*#*cs*⇒ *t*

theorem *exec1s-impl-execs*: $(cs,s) \rightarrow^* ([],t) \Longrightarrow s =cs\Rightarrow t$
apply(*erule converse-rtrancl-induct2*)
 apply(*rule execs.intros*)
apply(*erule exec1.cases*)
apply(*blast intro*:*execs.intros*)
apply(*blast intro*:*execs.intros*)
apply(*fastforce intro*:*execs.intros*)
apply(*fastforce intro*:*execs.intros*)
apply(*blast intro*:*execs.intros exec.intros*)+
done

theorem *exec1s-impl-exec*: $([c],s) \rightarrow^* ([],t) \Longrightarrow s -c\rightarrow t$
by(*blast dest*: *exec1s-impl-execs*)

primrec *termis* :: *com list* ⇒ *state* ⇒ *bool* (**infixl** ‹⇓› *60*) **where**
  $[]\Downarrow s = True$
| $c\#cs \Downarrow s = (c\downarrow s \wedge (\forall t.\ s -c\rightarrow t \longrightarrow cs\Downarrow t))$

lemma *exec1-pres-termis*: $(cs,s) \rightarrow (cs',s') \Longrightarrow cs\Downarrow s \longrightarrow cs'\Downarrow s'$
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*: $(cs,s) \rightarrow^* (cs',s') \Longrightarrow cs\Downarrow s \longrightarrow cs'\Downarrow s'$
apply(*erule rtrancl-induct2*)
 apply *blast*
apply(*blast dest*:*exec1-pres-termis*)
done

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

definition *termi-call-steps* :: ((*pname* × *state*) × (*pname* × *state*))*set* **where**
  termi-call-steps =
    $\{((q,t),(p,s)).\ body\ p\downarrow s \wedge (\exists cs.\ ([body\ p],\ s) \rightarrow^* (CALL\ q\#\ cs,\ t))\}$

lemma *lem*:
  $\forall y.\ (a,y)\in r^+ \longrightarrow P\ a \longrightarrow P\ y \Longrightarrow ((b,a) \in \{(y,x).\ P\ x \wedge (x,y)\in r\}^+) = ((b,a) \in \{(y,x).\ P\ x \wedge (x,y)\in r^+\})$
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*:
$[\![\forall i.\ (a,f\ i) \in r\hat{\ }* \wedge (f\ i,f(Suc\ i)) : r;\ (a,b) : r\hat{\ }*\ ]\!] \Longrightarrow b = f\ 0 \longrightarrow (\exists f.\ f\ 0 = a$ $\&\ (\forall i.\ (f\ i,\ f(Suc\ i)) : r))$
apply(*erule converse-rtrancl-induct*)
apply *blast*
apply(*clarsimp*)
apply(*rule-tac x*=$\lambda i.$ *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*:
$\forall i.\ (a,f\ i) : r\hat{\ }* \wedge (f\ i,f(Suc\ i)) : r \Longrightarrow \exists f.\ f\ 0 = a\ \&\ (\forall i.\ (f\ i,\ f(Suc\ i)) : r)$
by(*blast dest*:*renumber-aux*)

definition *inf* :: *com list* ⇒ *state* ⇒ *bool* **where**
$inf\ cs\ s \longleftrightarrow (\exists f.\ f\ 0 = (cs,s) \wedge (\forall i.\ f\ i \rightarrow f(Suc\ i)))$

lemma [*iff*]: ¬ *inf* [] *s*
apply(*unfold inf-def*)
apply *clarify*
apply(*erule-tac x = 0* **in** *allE*)
apply *simp*
done

lemma [*iff*]: ¬ *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*]: *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*(*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* = $\lambda i.$ *case i of 0* ⇒ ((*c1*;*c2*)#*cs*,*s*) | *Suc i* ⇒ *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*(*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* = $\lambda i.$ *case i of 0* ⇒ ((*IF b THEN c1 ELSE c2*)#*cs*,*s*) | *Suc i* ⇒
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*(*Suc i*) **in** *exI*)
apply *clarsimp*
apply *clarsimp*
apply(*rule-tac x* = $\lambda i.$ *f*(*Suc i*) **in** *exI*)
apply *clarsimp*
apply (*clarsimp split*:*if-splits*)
apply(*rule-tac x* = $\lambda i.$ *case i of 0* ⇒ ((*WHILE b DO c*)#*cs*,*s*) | *Suc i* ⇒ *f i* **in**

exI)
apply(*simp add*: *exec1.intros split*:*nat.split*)
apply(*rule-tac x* = *λi. case i of 0* ⇒ ((*WHILE b DO c*)#*cs*,*s*) | *Suc i* ⇒ *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* = *λi. f*(*Suc i*) **in** *exI*)
apply *clarsimp*
apply *clarsimp*
apply(*rule-tac x* = *λi. case i of 0* ⇒ (*CALL p*#*cs*,*s*) | *Suc i* ⇒ *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*(*λt*. {*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* = *λi. F*(*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*) → (*cs'*,*t*) ⟹
∀ *c cs. ccs* = *c*#*cs* ⟶ (∃ *cs1*. *cs'* = *cs1* @ *cs*)
apply(*erule exec1.induct*)
apply *force+*
done

lemma *exec1-only1*: (*c*#*cs*,*s*) → (*cs'*,*t*) ⟹ ∃ *cs1*. *cs'* = *cs1* @ *cs*
by(*blast dest*:*exec1-only1-aux*)

lemma *exec1-drop-suffix-aux*:
(*cs12*,*s*) → (*cs1'2*,*s'*) ⟹ ∀ *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*:
$(cs1@cs2,s) \rightarrow (cs1'@cs2,s') \Longrightarrow cs1 \neq [] \Longrightarrow (cs1,s) \rightarrow (cs1',s')$
by(*blast dest*:*exec1-drop-suffix-aux*)

lemma *execs-drop-suffix*[*rule-format*(*no-asm*)]:
$[\![f\ 0 = (c\#cs,s);\forall i.\ f(i) \rightarrow f(Suc\ i)]\!] \Longrightarrow$
$(\forall i{<}k.\ p\ i \neq []$ & $fst(f\ i) = p\ i@cs) \longrightarrow fst(f\ k) = p\ k@cs$
$\longrightarrow ([c],s) \rightarrow^* (p\ k,snd(f\ k))$
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*:
$[\![f\ 0 = (c\#cs,s);\forall i.\ f(i) \rightarrow f(Suc\ i);\ \forall i{<}k.\ p\ i \neq []$ & $fst(f\ i) = p\ i@cs;$
$fst(f\ k) = cs;\ p\ k = []\ ]\!] \Longrightarrow ([c],s) \rightarrow^* ([],snd(f\ k))$
apply(*drule execs-drop-suffix*,*assumption*,*assumption*)
 apply *simp*
apply *simp*
done

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

lemma *least-aux*: $[\![f\ 0 = (c\ \#\ cs,\ s);\ \forall i.\ f\ i \rightarrow f\ (Suc\ i);$
$fst(f\ k) = cs;\ \forall i{<}k.\ fst(f\ i) \neq cs]\!]$
$\Longrightarrow \forall i \leq k.\ (\exists p.\ (p \neq []) = (i < k)$ & $fst(f\ i) = p\ @\ cs)$
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*: $[\![f\ 0 = (c\#cs,s);\ \forall\, i.\ f\ i \to f(Suc\ i);\ \exists\, i.\ fst(f\ i) = cs\ ]\!]$
$\Longrightarrow \exists\, k.\ fst(f\ k) = cs\ \&\ ([c],s) \to^* ([],snd(f\ k))$
apply(*rule-tac* $x{=}LEAST\ i.\ fst(f\ i) = cs$ **in** *exI*)
apply(*rule conjI*)
apply(*fast intro*: *LeastI*)
apply(*subgoal-tac*
$\forall\, i{\le}LEAST\ i.\ fst\ (f\ i) = cs.\ \exists\, p.\ ((p \neq []) = (i<(LEAST\ i.\ fst\ (f\ i) = cs)))\ \&$
$fst(f\ i) = p@cs$)
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*: $\forall\, x.\exists\, y.\ P\ x\ y \Longrightarrow \exists\, f.\forall\, x.\ P\ x\ (f\ x)$
apply(*rule-tac* $x = \lambda x.\ SOME\ y.\ P\ x\ y$ **in** *exI*)
apply(*fast intro*:*someI2*)
done

lemma *inf-cases*: $inf\ (c\#cs)\ s \Longrightarrow inf\ [c]\ s \lor (\exists\, t.\ s\ {-c\rightarrow}\ t \land inf\ cs\ t)$
apply(*unfold inf-def*)
apply (*clarsimp del*: *disjCI*)
apply(*case-tac* $\exists\, i.\ fst(f\ i) = cs$)
 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* $\forall\, i.\ \exists\, p.\ p \neq [] \land fst(f\ i) = p@cs$)
 apply(*drule skolemize2*)
 apply *clarify*
 apply(*rename-tac p*)
 apply(*rule-tac* $x = \lambda i.\ (p\ i,\ snd(f\ i))$ **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*: $c \downarrow s \Longrightarrow \neg\ inf\ [c]\ s$
apply(*erule termi.induct*)

apply *clarify*

apply(*blast dest*:*inf-cases*)

apply *clarsimp*
apply *clarsimp*

apply *clarsimp*
apply(*fastforce dest*:*inf-cases*)

apply *blast*

apply(*blast dest*:*inf-cases*)
done

lemma *termi-impl-no-inf-chain*:
$c\downarrow s \Longrightarrow \neg(\exists f.\ f\ 0 = ([c],s) \wedge (\forall i::nat.\ (f\ i,\ f(i+1)) : exec1\hat{\ }+))$
apply(*subgoal-tac* $wf(\{(y,x).\ ([c],s) \rightarrow^* x\ \&\ x \rightarrow y\}\hat{\ }+)$)
apply(*simp only*:*wf-iff-no-infinite-down-chain*)
apply(*erule contrapos-nn*)
apply *clarify*
apply(*subgoal-tac* $\forall i.\ ([c],\ s) \rightarrow^* f\ i$)
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

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

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*)) $\to^*$
(*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* = λ*cs*.([*body p*],*s*) $\to^*$ (*CALL q*# *cs*, *t*) **in** *someI2*)
apply(*fastforce 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* = λ*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*. {(λ*z s*. (*z*=*s* $\wedge$ *body p*$\downarrow$*s*) $\wedge$ ((*p*,*s*),(*q*,*pre*)) $\in$ *termi-call-steps*, *CALL p*,
λ*z s*. *z* $-$*body p*$\to$ *s*)}) $\vdash_t$
{λ*z s*. (*z*=*s* $\wedge$ *body q*$\downarrow$*pre*) $\wedge$ ($\exists$ *cs*. ([*body q*],*pre*) $\to^*$ (*c*#*cs*,*s*))} *c*
{λ*z s*. *z* $-$*c*$\to$ *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* = λ*z s*. *body q*$\downarrow$*pre* & ($\exists$ *cs*. ([*body q*], *pre*) $\to^*$ (*c2*#*cs*,*s*)) & *z* $-$*c1*$\to$*s* & *c2*$\downarrow$*s* **in** *thoare.Semi*)

apply(*erule thoare.Conseq*)
apply(*rule conjI*)
apply *clarsimp*
apply(*subgoal-tac s −c1→ t*)
prefer *2*
apply(*blast intro*: *exec1.Semi exec-impl-execs rtrancl-trans*)
apply(*subgoal-tac* ([*body q*], *pre*) →* (*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*) →* (*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* $P' = \lambda z\ s.\ (z,s) \in (\{(s,t).\ b\ s \wedge s\ {-c\rightarrow}\ t\})\hat{}* \wedge body\ q{\downarrow}pre \wedge$
$(\exists\, cs.\ ([body\ q],\ pre) \rightarrow^{*} ((WHILE\ b\ DO\ c)\ \#\ cs,\ s))$ **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*:
$(\bigcup p.\ \{(\lambda z\ s.\ (z{=}s \wedge body\ p{\downarrow}s) \wedge ((p,s),(q,pre)) \in termi\text{-}call\text{-}steps,\ CALL\ p,$
$\lambda z\ s.\ z\ {-body\ p{\rightarrow}}\ s)\}) \vdash_t$
$\{\lambda z\ s.\ (z{=}s \wedge body\ q{\downarrow}s) \wedge s = pre\}\ body\ q\ \{\lambda z\ s.\ z\ {-body\ q{\rightarrow}}\ s\}$
apply(*rule strengthen-pre*[*OF - CALL-lemma*])
apply *blast*
done

lemma *MGT-CALL*: $\{\} \mid\!\vdash_t (\bigcup p.\ \{MGT_t(CALL\ p)\})$
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*: $\forall p.\ \{\} \mid\!\vdash_t \{MGT_t(CALL\ p)\}$
by(*fastforce 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 nondeterminism. 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. Steinbrüggen, 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.