Abstract Hoare Logics

Tobias Nipkow

August 16, 2018

Abstract

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

Contents

1 Introduction 1

2 Hoare Logics for While 2
  2.1 The language 2
  2.2 Hoare logic for partial correctness 4
  2.3 Termination 7
  2.4 Hoare logic for total correctness 9

3 Hoare Logics for 1 Procedure 12
  3.1 The language 12
  3.2 Hoare logic for partial correctness 16
  3.3 Termination 20
  3.4 Hoare logic for total correctness 22

4 Hoare Logics for Mutually Recursive Procedure 37
  4.1 The language 38
  4.2 Hoare logic for partial correctness 41
  4.3 Termination 45
  4.4 Hoare logic for total correctness 47

1 Introduction

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

theory Lang imports Main begin

2.1 The language

We start by declaring a type of states:

typedec 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 ⇒ 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 ⇒ state set) | Semi com com (\_; \_; [60, 60] 10) | Cond bexp com com (IF - THEN - ELSE - 60) | While bexp com (WHILE - DO - 60) | Local (state ⇒ state) com (state ⇒ state ⇒ state) (LOCAL - ; ; - [0,0,60] 60)

Statements in this language are called commands. They are modelled as terms of type com. Do \( f \) represents an atomic nondeterministic command that changes the state from \( s \) to some element of \( f s \). Thus the command that does nothing, often called skip, can be represented by \( \text{Do} \ (\lambda s. \{s\}) \).

Again we have chosen to model the semantics rather than the syntax, which simplifies matters enormously. Of course it means that we can no longer talk about certain syntactic matters, but that is just fine.

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

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

instead of Semi \( c_1 \ c_2 \), Cond \( b \ c_1 \ c_2 \) and While \( b \ c \).

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

inductive

exec :: state ⇒ com ⇒ state ⇒ bool (/ / -> / - [50,0,50] 50)

where

\[
t \in f s \Rightarrow s \rightarrow f \rightarrow t
\]

\[
\left[ s0 \rightarrow c1 \rightarrow s1; s1 \rightarrow c2 \rightarrow s2 \right] \Rightarrow s0 \rightarrow c1; c2 \rightarrow s2
\]
\[
\begin{align*}
&\frac{\begin{array}{l}
  b \land s \rightarrow c_1 \\
  \neg b \land s \rightarrow c_2
\end{array}}{
  s \rightarrow \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 
} \implies \\
&\frac{\begin{array}{l}
  b \land s \rightarrow t \\
  \neg b \land s \rightarrow t
\end{array}}{
  s \rightarrow \text{WHILE } b \text{ DO } c \rightarrow u 
} \implies \\
&f \land s \rightarrow t \implies s \rightarrow \text{LOCAL } f; c; g \rightarrow g \land s \land t
\end{align*}
\]

Assuming that the state is a function from variables to values, the declaration of a new local variable \( x \) with initial value \( a \) can be modelled as

\[
\text{LOCAL} (\lambda s. s(x := a) ; c ; (\lambda s. t(x := s x))).
\]

**Lemma** exec-Do-iff \([\text{iff}]\):

\[
(s \rightarrow \text{Do } f \rightarrow t) = (t \in f s)
\]

**Proof** (auto elim: exec.cases intro:exec.intros)

**Lemma** [iff]:

\[
(s \rightarrow c ; d \rightarrow u) = (\exists t. s \rightarrow c \land t \rightarrow d \land u)
\]

**Proof** (best elim: exec.cases intro:exec.intros)

**Lemma** [iff]:

\[
(s \rightarrow \text{IF } b \text{ THEN } c \text{ ELSE } d \rightarrow t) =
(s \rightarrow \text{if } b \land s \rightarrow c \land \neg b \land s \rightarrow d \rightarrow t)
\]

**Proof** auto

**Lemma** unfold-while:

\[
(s \rightarrow \text{WHILE } b \text{ DO } c \rightarrow u) =
(s \rightarrow \text{IF } b \text{ THEN } c \text{ ELSE } \text{DO}(\lambda s. \{s\}) \rightarrow u)
\]

**Proof** (auto elim: exec.cases intro:exec.intros split:if-split-asm)

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

\[
s \rightarrow w \rightarrow t \implies \forall b \land c. w = \text{WHILE } b \text{ DO } c \land P \land \forall s \land (\forall s' . P \land b \land s \land s \rightarrow c \rightarrow s' \rightarrow P \land \neg b \land t \rightarrow P \land t \land \neg b \land t)
\]

**Proof** (erule exec.induct)

**Lemma** while-rule:

\[
[s \rightarrow \text{WHILE } b \text{ DO } c \rightarrow t; P \land s; \forall s' . P \land s \land b \land s \land s \rightarrow c \rightarrow s' \rightarrow P \land s'] \implies P \land t \land \neg b \land t
\]

**Proof** (drule while-lemma)

**Preference** 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 ⇒ 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-valid :: assn ⇒ com ⇒ assn ⇒ bool (\(\models\) \(\{P\}\) c \(\{Q\}\)) where
\(\models\) \(\{P\}\) c \(\{Q\}\) \iff (\(\forall s t. s \rightarrow c \rightarrow t \rightarrow P s \rightarrow Q t\))

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 ⇒ com ⇒ assn ⇒ bool (\(\vdash\) \(\{P\}\) c \(\{Q\}\)) where
\(\vdash\) \(\{\lambda s. \forall t \in f s. P t\}\) Do f \(\{P\}\)

\(\vdash\) \(\{P\}\) c1 \(\{Q\}\); \(\vdash\) \(\{Q\}\) c2 \(\{R\}\) \implies \(\vdash\) \(\{P\}\) c1; c2 \(\{R\}\)

\(\vdash\) \(\{\lambda s. P s \land b s\}\) c1 \(\{Q\}\); \(\vdash\) \(\{\lambda s. P s \land \neg b s\}\) c2 \(\{Q\}\)

\(\Rightarrow\) \(\vdash\) \(\{P\}\) IF b THEN c1 ELSE c2 \(\{Q\}\)

\(\vdash\) \(\{\lambda s. P s \land b s\}\) c \(\{P\}\) \implies \(\vdash\) \(\{P\}\) WHILE b DO c \(\{\lambda s. P s \land \neg b s\}\)

\(\vdash\) \(\{\forall s. P' s \rightarrow P s; \vdash\} c\{Q\}; \forall s. Q s \rightarrow Q' s\} \implies \vdash\) \(\{P\}\) c \(\{Q'\}\)

\(\vdash\) \(\{\lambda s. P s \rightarrow P' s (f s); \forall s. \vdash\} c\{Q \circ (g s)\}\} \implies \vdash\) \(\{\lambda P f; c; g\} c\{Q\}\)

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

theorem hoare-sound; \(\vdash\) \(\{P\}\) c \(\{Q\}\) \implies \(\models\) \(\{P\}\) c \(\{Q\}\)
apply (unfold hoare-valid-def)
apply (erule hoare.induct)
apply blast
apply blast
Completeness is not quite as straightforward, but still easy. The proof is best explained in terms of the weakest precondition:

**definition**

\[
wp :: \text{com} \Rightarrow \text{assn} \Rightarrow \text{assn} \quad \text{where} \quad wp\ c\ Q = (\lambda s. \forall t. s \rightarrow c \rightarrow t \rightarrow Q t)
\]

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]:** \( wp (\text{Do } f)\ Q = (\lambda s. \forall t \in f s. Q(t)) \)

**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\text{-while}:**
\[ wp \ (WHILE \ b \ DO \ c) \ Q = \ (\lambda s. \ if \ b \ s \ then \ wp \ (c; \ WHILE \ b \ DO \ c) \ Q \ s \ else \ Q \ s) \]

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

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

lemma weaken-post: [ \[ \vdash \{P\}c\{Q\}; \forall s. \ Q \ s \rightarrow Q' \ s \] ] \implies \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: \{ P \} c \{ Q \} \implies \{ P \} c \{ Q \}
apply (rule strengthen-pre[OF - wp-is-prev])
apply(unfold hoare-valid-def wp-def)
apply blast
done

end

theory Termi imports Lang begin

2.3 Termination

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

inductive
\begin{align*}
termi ::& \quad \text{com} \Rightarrow \text{state} \Rightarrow \text{bool} \quad \text{(infixl $\downarrow$ 50)} \\
\text{where} & \\
\quad f s \neq {} \quad \Rightarrow \quad \text{Do } f \downarrow s \\
\mid [& \quad c_1 \downarrow s_0; \forall s_1. s_0 \vdash c_1 \Rightarrow s_1 \vdash c_2 \downarrow s_1 ] \quad \Rightarrow \quad (c_1;c_2) \downarrow s_0 \\
\mid [& \quad b s; c_1 \downarrow s | \quad \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 \downarrow s \\
\mid [& \quad \neg b s; c_2 \downarrow s | \quad \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 \downarrow s \\
\mid [& \quad \neg b s \quad \Rightarrow \quad \text{WHILE } b \text{ DO } c \downarrow s \\
\mid [& \quad b s; c \downarrow s; \forall t. s \vdash t \vdash \text{WHILE } b \text{ DO } c \downarrow t ] \quad \Rightarrow \quad \text{WHILE } b \text{ DO } c \downarrow s \\
\mid [& \quad c \downarrow f s \quad \Rightarrow \quad \text{LOCAL } f;c;g \downarrow s \\
\end{align*}

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

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

lemma [iff]: \((IF b THEN c_1 ELSE c_2 \downarrow s) = \)
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]: \((LOCAL f;g \downarrow s) = (c \downarrow f s) \)
by(fast elim: termi.cases intro:termi.intros)

lemma termi-while-lemma[rule-format]:
\[w \downarrow f k \Rightarrow \]
\((\forall k b c. f k = f k \land w = WHILE b DO c \land (∀i. f i \rightarrow c \rightarrow f(Suc i)))
\rightarrow (∃i. \neg b(f i)))\]
apply(erule termi.induct)
apply simp-all
apply blast
apply blast
done

lemma termi-while:
\[ (WHILE b DO c) \downarrow f k; (∀i. f i \rightarrow c \rightarrow f(Suc i)) \Rightarrow (∃i. \neg b(f i)) \]
by(blast intro:termi-while-lemma)

lemma wf-termi: \(w f \) t.s. \(WHILE b DO c \downarrow s \land b s \land s \rightarrow c \rightarrow t\)
apply(subst wf-iff-no-infinite-down-chain)
apply(rule notI)
apply clarsimp
apply(insert termi-while)
apply blast
done

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

\[
\text{hoare-tvalid} :: \text{assn} \Rightarrow \text{com} \Rightarrow \text{assn} \Rightarrow \text{bool}
\]

\[
\models_t \{ P \} c \{ Q \} \text{ where } (\forall s. P s \rightarrow c \downarrow s)
\]

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

inductive

\[
\text{thoare} :: \text{assn} \Rightarrow \text{com} \Rightarrow \text{assn} \Rightarrow \text{bool}
\]

\[
\vdash \{ (1-) \} / (\cdot) / (1-)\ 50
\]

where

Do:

\[
\vdash \{ \lambda s. (\forall t \in f s. P t) \land f s \neq \{\} \} \text{ Do } f \{ P \}
\]

Semi:

\[
\vdash \{ P \} c \{ Q \} ; \vdash \{ Q \} d \{ R \} \quad \Rightarrow \quad \vdash \{ P \} c ; d \{ R \}
\]

If:

\[
\vdash \{ \lambda s. P s \land b s \} c \{ Q \} \land \vdash \{ \lambda s. P s \land \sim b s \} d \{ Q \} \quad \Rightarrow
\]

\[
\vdash \{ P \} \text{ IF } b \text{ THEN } c \text{ ELSE } d \{ Q \}
\]

While:

\[
[\text{wf } r; \forall s'. \vdash \{ \lambda s. P s \land b s \land s' = s \} \ c \{ \lambda s. P s \land (s, s') \in r \}] \quad \Rightarrow \quad \vdash \{ P \} \text{ WHILE } b \text{ DO } c \{ \lambda s. P s \land \sim b s \}
\]

Conseq:

\[
[ \forall s. P' s \rightarrow P s; \vdash \{ P \} c \{ Q \}; \forall s. Q s \rightarrow Q' s ] \quad \Rightarrow
\]

\[
\vdash \{ P' \} c \{ Q' \}
\]

Local:

\[
\{ P \} \Rightarrow \{ P' \} (f s) \Rightarrow \forall p. \vdash \{ P' \} \ c \{ Q o (g p) \} \Rightarrow
\]

\[
\vdash \{ P \} \text{ LOCAL } f; c g \{ Q \}
\]

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 (\( \text{wf } r \)) on the state space decreases.

The soundness theorem

\[
\vdash \{ P \} c \{ Q \} \Rightarrow \models_t \{ P \} c \{ Q \}
\]

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 ⇒ assn ⇒ assn (wp_t)
where
wp_t c Q = (λs. wp c Q s ∨ c↓)

lemmas wp-defs = wp-def wpt-def

lemma [simp]: wp_t (Do f) Q = (λs. (∀t ∈ f s. Q t) ∧ f s ≠ {})
by (simp add: wp-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 = (λs. 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]: \( \wp_t (\text{LOCAL } f;c;g) Q = (\lambda s. \wp_t c (Q \circ g s) (f s)) \)
apply(unfold wp-defs)
apply(rule ext)
apply auto
done

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

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

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

lemma wp-is-pre[rule-format]: \( \vdash_t \{ \wp_t c Q \} c \{ Q \} \)
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 \( \vdash_t \{ P \} c\{Q\} \implies \vdash_t \{ P \} c\{Q\} \)
apply (rule strengthen-pre[OF - wp-is-pre])
apply(unfold hoare-tvalid-def hoare-valid-def wp-defs)
apply blast

11
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 $s \rightarrow c \rightarrow t$. The only new rule is the one for CALL — it requires no comment:

inductive

\[\text{exec} : \text{state} \Rightarrow \text{com} \Rightarrow \text{state} \Rightarrow \text{bool} \quad (-/ -/-/ - [50,0,50] 50)\]

where

\[\text{Do:} \quad t \in f s \quad \Rightarrow \quad s \rightarrow \text{Do } f \rightarrow t\]

\[\text{Semi:} \quad \left[\begin{array}{l}
  s0 \rightarrow c1 \rightarrow s1; s1 \rightarrow c2 \rightarrow s2
\end{array}\right] \quad \Rightarrow \quad s0 \rightarrow c1; c2 \rightarrow s2\]

\[\text{IfTrue:} \quad \left[\begin{array}{l}
  b; s \rightarrow c1 \rightarrow t
\end{array}\right] \quad \Rightarrow \quad s \rightarrow \text{IF } b \text{ THEN } c1 \text{ ELSE } c2 \rightarrow t\]

\[\text{IfFalse:} \quad \left[\begin{array}{l}
  \neg b; s \rightarrow c2 \rightarrow t
\end{array}\right] \quad \Rightarrow \quad s \rightarrow \text{IF } b \text{ THEN } c1 \text{ ELSE } c2 \rightarrow t\]

\[\text{WhileFalse:} \quad \neg b \quad \Rightarrow \quad s \rightarrow \text{WHILE } b \text{ DO } c \rightarrow s\]

\[\text{WhileTrue:} \quad \left[\begin{array}{l}
  b; s \rightarrow c \rightarrow t; t \rightarrow \text{WHILE } b \text{ DO } c \rightarrow u
\end{array}\right] \quad \Rightarrow \quad s \rightarrow \text{WHILE } b \text{ DO } c \rightarrow u\]

\[s \rightarrow \text{body} \rightarrow t \quad \Rightarrow \quad s \rightarrow \text{CALL} \rightarrow t\]

\[\text{Local:} \quad f s \rightarrow c \rightarrow t \quad \Rightarrow \quad s \rightarrow \text{LOCAL } f; c; g \rightarrow g s t\]
lemma [iff]: \((s - \text{Do } f \rightarrow t)\) \(= (t \in f s)\)
by(auto elim: exec.cases intro:exec.intros)

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

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

lemma unfold-while:
\((s - \text{WHILE } b \text{ DO } c \rightarrow u) = \)
\((s - \text{IF } b s \text{ then } c \text{ ELSE } c \text{ DO } \lambda s. \{s\} \rightarrow u)\)
by(auto elim: exec.cases intro:exec.intros)

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

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

lemma simp: \(-b s = \Rightarrow s - \text{WHILE } b \text{ DO } s\)
by(fast intro:exec.intros)

lemma WhileI: \([b s; s - c \rightarrow t; t - \text{WHILE } b \text{ DO } c \rightarrow u] = \Rightarrow s - \text{WHILE } b \text{ DO } c \rightarrow u\)
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
\(\text{execn} :: \text{state} \Rightarrow \text{com} \Rightarrow \text{nat} \Rightarrow \text{state} \Rightarrow \text{bool} \ (-/ -/ -/ -/ -50,0,0,50 \ 50)\)
where
\(t \in f s \Rightarrow s - \text{Do } f - n \rightarrow t\)
\(\mid [ s0 - c1 - n \rightarrow s1; s1 - c2 - n \rightarrow s2 ] = \Rightarrow s0 - c1; c2 - n \rightarrow s2\)
\(\mid [ b \; s; s - c1 - n \rightarrow t ] = \Rightarrow s - \text{IF } b \text{ THEN } c1 \text{ ELSE } c2 - n \rightarrow t\)
\(\mid [ \neg b \; s; s - c2 - n \rightarrow t ] = \Rightarrow s - \text{IF } b \text{ THEN } c1 \text{ ELSE } c2 - n \rightarrow t\)
\[ \neg b \quad s \implies s - \text{WHILE} b \ DO \ c - n \rightarrow s \]
\[ [b \; s; \; s - c - n \rightarrow t; \; t - \text{WHILE} b \ DO \ c - n \rightarrow u] \implies s - \text{WHILE} b \ DO \ c - n \rightarrow u \]
\[ s - \text{body} - n \rightarrow t \implies s - \text{CALL} - \text{Suc} \ n \rightarrow t \]
\[ f \; s - c - n \rightarrow t \implies s - \text{LOCAL} \ f; \ c; \ g - n \rightarrow g \ s \ t \]

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

\textbf{lemma}\ [\text{iff}]:\ (s - c \; t; \ c 2 - n \rightarrow u) = (\exists t. \ s - c 1 - n \rightarrow t \land t - c 2 - n \rightarrow u)
\text{by}(\text{best elim: execn.cases intro:execn.intros})

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

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

\textbf{lemma}\ [\text{iff}]:\ (s - \text{LOCAL} \ f; \ c; \ g - n \rightarrow u) = (\exists t. \ f \ s - c - n \rightarrow t \land u = g \ s \ t)
\text{by}(\text{auto elim: execn.cases intro:execn.intros})

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

\textbf{lemma}\ \text{exec-mono}\ [\text{rule-format}]:\ s - c - m \rightarrow t \implies \forall n. \ m \leq n \implies s - c - n \rightarrow t
\text{apply}(\text{erule execn.induct})
\quad \text{apply}(\text{blast})
\quad \text{apply}(\text{blast})
\quad \text{apply}(\text{simp})
\quad \text{apply}(\text{simp})
\quad \text{apply}(\text{simp add:execn.intros})
\quad \text{apply}(\text{blast intro:execn.intros})
\quad \text{apply}(\text{clarify})
\quad \text{apply}(\text{rename-tac m})
\quad \text{apply}(\text{case-tac m})
\quad \text{apply simp}
\quad \text{apply simp}
\quad \text{apply blast}
\text{done}

With the help of this lemma we prove the expected relationship between the two semantics:
lemma \( \text{exec-iff-execn} \): (\( s - c \rightarrow t \)) = (\exists n. s - c-n \rightarrow t) 
apply (rule \text{iffI})
apply (erule \text{exec.induct})
apply blast
apply clarify
apply (rename-tac m n)
apply (rule-tac \( x = \max m n \) in \text{exI})
apply (fastforce intro: \text{exec.intros} \text{exec-mono} simp add: max-def)
apply fastforce
apply fastforce
apply (blast intro: \text{execn.indros})
apply clarify
apply (rename-tac m n)
apply (rule-tac \( x = \max m n \) in \text{exI})
apply (fastforce elim: \text{execn.WhileTrue} \text{exec-mono} simp add: max-def)
apply blast
apply blast
apply (erule \text{exE}, erule \text{execn.induct})
apply blast
apply blast
apply fastforce
apply fastforce
apply (erule \text{exec.WhileFalse})
apply (blast intro: \text{exec.intros})
apply blast
apply blast
done

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

lemma \text{while-rule}:
\[ [s \dashrightarrow \text{WHILE} b \text{ DO } c-n \rightarrow t; P s; \exists s'. [P s; b s; s - c-n \rightarrow s'] \implies P s'] \implies P t \land \neg b t \]
apply (drule \text{while-lemma})
prefer 2 apply assumption
apply blast
done

15
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 ⇒ state ⇒ bool

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

**type-synonym** 'a cntxt = ('a assn × com × '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 ⇒ com ⇒ 'a assn ⇒ bool (|= {(1-)}/ (-)/ {(1-)} 50) where

|= {P} c{Q} ←→ (∀s t. s −c→ t −→ (∀z. P z s −→ 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 ⇒ bool (= - 50) where

[simp]: |= C ≡ (∀(P,c,Q) ∈ C. |= {P} c{Q})

**definition**

cvalid :: 'a cntxt ⇒ 'a assn ⇒ com ⇒ 'a assn ⇒ bool (=/ {(1-)}/ (-)/ {(1-)} 50) where

C |= {P} c{Q} ←→ |= C −→ |= {P} c{Q}

Note that {} |= {P} c {Q} is equivalent to |= {P} c {Q}.

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

**definition**

nvalid :: nat ⇒ 'a assn ⇒ com ⇒ 'a assn ⇒ bool (=− {(1-)}/ (-)/ {(1-)} 50) where
\[ \vdash n \{P\} c\{Q\} \equiv (\forall s \ t. \ s - c \ n \rightarrow t \rightarrow (\forall z. \ P z s \rightarrow Q z t)) \]

definition
\[ \text{cnvalid} :: 'a \text{cntxt} \Rightarrow \text{nat} \Rightarrow 'a \text{assn} \Rightarrow \text{com} \Rightarrow 'a \text{assn} \Rightarrow \text{bool} (\vdash \vdash/ \vdash \|(1-))/(-)/\|(1-))/50) \text{ where} \]
\[ C \vdash n \{P\} c\{Q\} \iff \vdash \vdash C \vdash \vdash n \{P\} c\{Q\} \]

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

inductive
\[ \text{hoare} :: 'a \text{cntxt} \Rightarrow 'a \text{assn} \Rightarrow \text{com} \Rightarrow 'a \text{assn} \Rightarrow \text{bool} (\vdash \vdash/ \vdash \|(1-))/(-)/\|(1-))/50) \text{ where} \]
\[ C \vdash \{\lambda z s. \forall t \in f s. \ P z t\} \text{ Do } \{P\} \]
\[ | \square \vdash \{P\} c1\{Q\}; \ C \vdash \{Q\} c2\{R\} \implies C \vdash \{P\} c1; c2 \{R\} \]
\[ | \square \vdash \{\lambda z s. P z s \land b s\} c1\{Q\}; \ C \vdash \{\lambda z s. P z s \land \neg b s\} c2\{Q\}; \]
\[ \implies C \vdash \{P\} \text{ IF } b \text{ THEN } c1 \text{ ELSE } c2 \{Q\} \]
\[ | C \vdash \{\lambda z s. P z s \land b s\} c \{P\} \]
\[ \implies C \vdash \{P\} \text{ WHILE } b \text{ DO } c \{\lambda z s. P z s \land \neg b s\} \]
\[ | \square \vdash \{P\} c\{Q'\}; \ \forall s t. (\forall z. \ P' z s \rightarrow Q' z t) \rightarrow (\forall z. \ P z s \rightarrow Q z t) \]
\[ \implies C \vdash \{P\} c\{Q\} \]
\[ | \{(P, CALL, Q)\} \vdash \{P\} \text{body}\{Q\} \implies \{\} \vdash \{P\} \text{CALL}\{Q\} \]
\[ | \{(P, CALL, Q)\} \vdash \{P\} \text{CALL}\{Q\} \]
\[ | \forall s'. C \vdash \{\lambda z s. P z s' \land s = f s'\} c \{\lambda z t. Q z (g s' t)\} \]
\[ \implies C \vdash \{P\} \text{LOCAL } f; c; g\{Q\} \]

abbreviation \[ \text{hoare1} :: 'a \text{cntxt} \Rightarrow 'a \text{assn} \times \text{com} \times 'a \text{assn} \Rightarrow \text{bool} (\vdash \vdash) \text{ where} \]
\[ C \vdash x \equiv C \vdash \{\text{fst } x\} \text{fst } (\text{snd } x)\{\text{snd } (\text{snd } x)\} \]

The first four rules are familiar, except for their adaptation to auxiliary variables. The \text{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 \[ \text{strengthen-pre}: \]
\[ [ \forall s. P' z s \rightarrow P z s; \ C \vdash \{P\} c\{Q\} ] \implies C \vdash \{P'\} c\{Q\} \]
by (rule hoare.Conseq, assumption, blast)

lemmas \[ \text{valid-defs} = \text{valid-def valids-def cvalid-def} \]
nvalid-def nvalids-def cnvalid-def

**Theorem** hoare-sound: \( C \vdash \{P\} c \{Q\} \Rightarrow C \models \{P\} c \{Q\} \)

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 \textsection ???. 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. \( n \).

```ml
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**  
\( MGT :: \text{com} \Rightarrow \text{state assn} \times \text{com} \times \text{state assn} \) where  
\( MGT \ c = (\lambda z \ s. z = s, c, \lambda z \ t. z \rightarrow t) \)

**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, \( \{\} \models MGT \ c \) implies completeness:

**Lemma** `MGT-implies-complete`:
\{\} \vdash \textit{MGT} \ c \Rightarrow \{\} \models \{P\} \ c \{Q\} \Rightarrow \{\} \vdash \{P\} \ c \{ \text{state assn} \}

apply (simp add: \textit{MGT-def})
apply (erule \textit{hoare}.Conseq)
apply (simp add: valid-defs)
done

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

\textbf{lemma} \textit{MGT-lemma}: \ C \vdash \textit{MGT} \ \textit{CALL} \Rightarrow \ C \vdash \textit{MGT} \ c

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

The proof is by induction on \ c. In the \textit{While}-case it is easy to show that
\lambda z t. (z, t) \in \{ (s, t). \ b \ s \land 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. \textit{exec} z (WHILE \ b \ DO \ c). The remaining cases are trivial.

Using the \textit{MGT-lemma} (together with the \textit{CALL} and the assumption rule) one can easily derive

\textbf{lemma} \textit{MGT-CALL}: \{\} \vdash \textit{MGT} \ \textit{CALL}

apply (simp add: \textit{MGT-def})
apply (rule \textit{hoare}.\textit{Call})
apply (rule \textit{hoare}.\textit{Conseq}[OF \textit{MGT-lemma}[simplified], \textit{OF} \textit{hoare}.\textit{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** $\{\} \models \{P\} \ c \{Q\} \Rightarrow \{\} \models \{P\} \ c \{Q::\text{state assn}\}$

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

**done**

end

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

3.3 Termination

**inductive**

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

**where**

$\text{Do [iff]}$: $f \ s \neq \{\} \Rightarrow \text{Do } f \downarrow s$

$\mid \text{Semi [intro]}$: $[\ c1 \downarrow s0; \land s1. \ s0 - c1 \rightarrow s1 \Rightarrow c2 \downarrow s1 \ ] \Rightarrow (c1;c2) \downarrow s0$

$\mid \text{IfTrue [intro, simp]}$: $[\ b \ s \downarrow s ] \Rightarrow \text{IF } b \ \text{THEN } c1 \ \text{ELSE } c2 \downarrow s$

$\mid \text{IfFalse [intro, simp]}$: $[\ \neg b \ s \downarrow s ] \Rightarrow \text{IF } b \ \text{THEN } c1 \ \text{ELSE } c2 \downarrow s$

$\mid \text{WhileFalse} \ : \neg b \ s \Rightarrow \text{WHILE } b \ \text{DO } c \downarrow s$

$\mid \text{WhileTrue} \ : [\ b \ s; \ c \downarrow s; \land t. \ s - c \rightarrow t \Rightarrow \text{WHILE } b \ \text{DO } c \downarrow t \ ] \Rightarrow \text{WHILE } b \ \text{DO } c \downarrow s$

$\mid \text{body} \downarrow s \Rightarrow \text{CALL} \downarrow s$

$\mid \text{Local} \ : c \downarrow f \ s \Rightarrow \text{LOCAL } f;c:g \downarrow s$

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

**apply (rule iffI)**

**prefer 2**

**apply (best intro:termi.intros)**

**apply (erule termi.cases)**

**apply blast+**

**done**

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

**apply (rule iffI)**

**prefer 2**

**apply (best intro:termi.intros)**

**apply (erule termi.cases)**

**apply blast+**

**done**

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

**apply simp**

20
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↓ 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 =⇒ 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 =⇒ 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 =⇒ 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**

\[
	ext{tvalid} :: \{a\ \text{assn} \Rightarrow \text{com} \Rightarrow \{a\ \text{assn} \Rightarrow \text{bool} (|_{-}((1-)}/ (-)/((1-)) \ 50) \}
\]

**definition**

\[
	ext{ctvalid} :: \{a\ \text{cntxt} \Rightarrow \{a\ \text{assn} \Rightarrow \text{com} \Rightarrow \{a\ \text{assn} \Rightarrow \text{bool} ((|_{-}((1-)}/ (-)/((1-)) \ 50) \}
\]

**inductive**

\[
\text{thoare} :: \{a\ \text{cntxt} \Rightarrow \{a\ \text{assn} \Rightarrow \text{com} \Rightarrow \{a\ \text{assn} \Rightarrow \text{bool} ((|_{-}((1-)}/ (-)/((1-)) \ 50) \}
\]

**abbreviation**

\[
\text{hoare1} :: \{a\ \text{cntxt} \Rightarrow \{a\ \text{assn} \times \text{com} \times \{a\ \text{assn} \Rightarrow \text{bool } (-|_{-})\ \text{where} \}
\]

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

**lemma**

\[
((|_{-}((1-)}/ (-)/((1-)) \ 50) \}
\]

\[
\text{\texttt{\textcolor{red}{abbreviation}} \ hoare1 :: \{a\ \text{cntxt} \Rightarrow \{a\ \text{assn} \times \text{com} \times \{a\ \text{assn} \Rightarrow \text{bool } (-|_{-})\ \text{where} \}
\]

\[
\text{\texttt{\textcolor{red}{abbreviation}} \ hoare1 :: \{a\ \text{cntxt} \Rightarrow \{a\ \text{assn} \times \text{com} \times \{a\ \text{assn} \Rightarrow \text{bool } (-|_{-})\ \text{where} \}
\]

\[
\text{\texttt{\textcolor{red}{abbreviation}} \ hoare1 :: \{a\ \text{cntxt} \Rightarrow \{a\ \text{assn} \times \text{com} \times \{a\ \text{assn} \Rightarrow \text{bool } (-|_{-})\ \text{where} \}
\]

\[
\text{\texttt{\textcolor{red}{abbreviation}} \ hoare1 :: \{a\ \text{cntxt} \Rightarrow \{a\ \text{assn} \times \text{com} \times \{a\ \text{assn} \Rightarrow \text{bool } (-|_{-})\ \text{where} \}
\]
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:**
\[
\forall z s. \ P' z s \rightarrow P z s; \ C \vdash t \{P\} c\{Q\} \implies C \vdash t \{P'\} c\{Q\}
\]
by \text{rule thoare.Conseq, assumption, blast}

**Lemma weaken-post:**
\[
C \vdash t \{P\} c\{Q\}; \forall z s. \ Q z s \rightarrow Q' z s \implies C \vdash t \{P\} c\{Q'\}
\]
by \text{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]:**
\[
(\models t \{\lambda z s. \ P z s \land P'\} c\{Q\}) = (P' \rightarrow \models t \{P\} c\{Q\})
\]
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 (erule 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 \(t, x\))
apply fast
apply blast
apply (simp (no-asм-use) add: ctvalid-def)
apply (simp (no-asм-use) add: tvalid-defs)
apply blast
apply (rule allI)
apply (erule wf-induct)
apply (unfold tvalid-defs)
apply fast
apply fast
done

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

lemma MGT-implies-complete:
\(\{\} \mid t \ MGT_t c \Longrightarrow \{\} \mid t \ \{P\} c \{Q\} \Longrightarrow \{\} \mid t \ \{P\} c \{Q::\text{state assn}\}\)
apply (simp add: MGT_t-def)
apply (erule thoare. Conseq)
apply (simp add: tvalid-defs)
apply blast
done

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

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

lemma MGT-lemma: \(C \mid t \ MGT_t \ CALL \Longrightarrow C \mid t \ MGT_t c\)
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 (erule 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 \& s - c \rightarrow t\}^* \&
            WHILE b DO c \rightarrow 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 \ (\Rightarrow [81, 81]
100)
where
cs0 \rightarrow cs1 \equiv (cs0, cs1) : exec1

| Do[iff]: t \in f s \Rightarrow ((Do f) \# cs, s) \rightarrow (cs, t)
| Semi[iff]: ((c1 ; c2) \# cs, s) \rightarrow (c1 \# c2 \# cs, s)
| IfTrue: b s \Rightarrow ((IF b THEN c1 ELSE c2) \# cs, s) \rightarrow (c1 \# cs, s)
| IfFalse: \neg b s \Rightarrow ((IF b THEN c1 ELSE c2) \# cs, s) \rightarrow (c2 \# cs, s)
WhileFalse: \( \neg b \Rightarrow ((\text{WHILE } b \text{ DO } c) \# cs, s) \rightarrow (cs, s) \)

WhileTrue: \( b \Rightarrow ((\text{WHILE } b \text{ DO } c) \# cs, s) \rightarrow (c\#(\text{WHILE } b \text{ DO } c) \# cs, s) \)

Call[iff]: \( (\text{CALL} \# cs, s) \rightarrow (\text{body} \# cs, s) \)

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

abbreviation
execr :: \((\text{com list} \times \text{state}) \Rightarrow (\text{com list} \times \text{state}) \Rightarrow \text{bool} \)

where \( cs0 \rightarrow^* cs1 \equiv (cs0, cs1) : \text{exec1}\)

inductive-cases execIE[elim!]:
  \(([]), s) \rightarrow (cs', s')\)
  \((\text{Do } f \# cs, s) \rightarrow (cs', s')\)
  \(((c1;c2) \# cs, s) \rightarrow (cs', s')\)
  \(((\text{IF } b \text{ THEN } c1 \text{ ELSE } c2) \# cs, s) \rightarrow (cs', s')\)
  \(((\text{WHILE } b \text{ DO } c) \# cs, s) \rightarrow (cs', s')\)
  \((\text{CALL} \# cs, s) \rightarrow (cs', s')\)
  \(((\text{LOCAL } f;c;g) \# cs, s) \rightarrow (cs', s')\)

lemma [iff]: \( \neg ([], s) \rightarrow u \)
  by \( \text{(induct } u \text{) blast} \)

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

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

lemma exec-impl-execs[rule-format]:
  \( s -c \rightarrow s' \Rightarrow \forall cs. (c\# cs, s) \rightarrow^* (cs, s') \)
apply(erule exec.induct)
  apply blast
    apply(blast intro:rtrancl-trans)
    apply(blast intro:exec1.IfTrue rtrancl-trans)
    apply(blast intro:exec1.IfFalse rtrancl-trans)
    apply(blast intro:exec1.WhileFalse rtrancl-trans)
    apply(blast intro:exec1.WhileTrue rtrancl-trans)
  apply(blast intro: rtrancl-trans)
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-exec: (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(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-execs: (cs,s) ⇒* ([],t) ⇒= s =cs⇒ t
by(blast dest: exec1s-impl-execs)

primrec termis :: com list ⇒ state ⇒ bool (infixl \-\-\-\- 60) where
  []\-\-\-\- s = True
  | c#cs \-\-\-\- s = (c\-\-\-\- s ∧ (\forall 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
definition

termi-call-steps :: (state × state)set where

\[ \text{termi-call-steps} = \{ (t,s). \text{body}↓s \land (\exists cs. ([\text{body}], s) \rightarrow^\ast (\text{CALL} \# cs, t)) \} \]

lemma lem:

\[ \forall y. (a,y)\in r^+ \rightarrow P a \rightarrow P y \implies ((b,a) \in \{(y,x). P x \land (x,y)\in r^\}\) = ((b,a) \in \{(y,x). P x \land (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:

\[ \forall i. (a,f i) : r^* \land (f i,f(Suc i)) : r; (a,b) : r^* \] \implies b = f 0 \implies (\exists f. f 0 = a \land (\forall i. (f i, f(Suc i)) : r))

apply(erule converse-rtrancl-induct)

apply blast

apply(clar)

apply(rule_tac x=\lambda i. case i of 0 \Rightarrow y | Suc i \Rightarrow fa i in exI)

apply simp

apply clarify

apply(case_tac i)

apply simp-all

done

lemma renumber:

\[ \forall i. (a,f i) : r^* \land (f i,f(Suc i)) : r \implies (\exists f. f 0 = a \land (\forall i. (f i, f(Suc i)) : r)) \]

by(blast dest:renumber-aux)

definition inf :: com list ⇒ state ⇒ bool where

\[ \text{inf} cs s \longleftrightarrow (\exists f. f 0 = (cs,s) \land (\forall i. f i \rightarrow f(Suc i))) \]

lemma [iff]: ¬ inf [] s

apply(unfold inf-def)

apply clarify

apply(erule_tac x=0 in allE)

apply simp

done
lemma [iff]: \( \neg \inf (\text{Do } f) s \)
applyunfold inf-def
applyclarify
applyrule-tac \( x = 0 \) in spec
applyerule-tac \( x = 1 \) in allE
applycase-tac fa (Suc 0)
done

lemma [iff]: \( \inf ((c1;c2)\#cs) s = \inf (c1\#c2\#cs) s \)
applyunfold inf-def
applyrule iffI
applyclarsimp
applyrule-tac \( x = \lambda i. f(Suc i) \in exI \)
applyrule-tac \( x = 0 \) in spec
applycase-tac f (Suc 0)
done

lemma [iff]: \( \inf ((\text{IF } b \text{ THEN } c1 \text{ ELSE } c2)\#cs) s = \inf ((\text{IF } b \text{ THEN } c1 \text{ ELSE } c2)\#cs) s \)
applyunfold inf-def
applyrule iffI
applyclarsimp
applyrule-tac \( x = 0 \) in spec
applycase-tac f (Suc 0)
done

lemma [simp]: \( \inf ((\text{WHILE } b \text{ DO } c)\#cs) s = (\text{if } b \text{ s then } \inf (c\#(\text{WHILE } b \text{ DO } c)\#cs) \text{ s else } \inf cs s) \)
applyunfold inf-def
applyrule iffI
applyclarsimp
applyrule-tac \( x = 0 \) in spec
apply (case-tac f (Suc 0))
apply (rule conjI)
apply clarsimp
apply (rule-tac x = λi. f(Suc i) in exI)
apply clarsimp
apply (rule-tac x = λi. Suc f(Suc i) in exI)
apply clarsimp
apply (clarsimp split: if-splits)
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# cs) s = inf (body# cs) s
apply (unfold inf-def)
apply (rule iffI)
apply clarsimp
apply (rule-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 split: if-splits)
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 ((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 (rule-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) | Suc i ⇒ F i in exI)
apply (simp add: exec1.intros split:nat.split)
done
lemma exec1-only1-aux: \((cs, s) \rightarrow (cs', t) \Rightarrow \forall c. cs = c\#cs \Rightarrow (\exists cs1. cs' = cs1 \circ cs)\)
apply (erule exec1.induct)
apply blast
apply force+
done

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

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

lemma exec1-drop-suffix:
\((cs1\#cs2, s) \rightarrow (cs1'\#cs2, s') \Rightarrow cs1 \neq [] \Rightarrow (cs1, s) \rightarrow (cs1', s')\)
by (blast dest: exec1-drop-suffix-aux)

lemma execs-drop-suffix [rule-format (no-asn)]:
\[( f \theta = (c\#cs, s); \forall i. f(i) \rightarrow f(Suc i) ) \Rightarrow
\forall i < k. p i \neq [] \& \text{fst}(f i) = p i \# cs \Rightarrow \text{fst}(f k) = p k \# cs
\rightarrow ([i], s) \rightarrow (p k, \text{snd}(f k))\)
apply (induct-tac k)
apply simp
apply (clarsimp)
apply (clar simp)
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:
\[( f \theta \pi = (c\#cs, s); \forall i. f(i) \rightarrow f(Suc i); \forall i < k. p i \neq [] \& \text{fst}(f i) = p i \# cs;
\text{fst}(f k) = cs; p k = [] \Rightarrow ([i], s) \rightarrow \text{snd}(f k))\]
apply (drule execs-drop-suffix, assumption, assumption)
apply simp
apply simp
apply simp
apply simp
apply simp
apply simp
apply simp
apply simp
apply simp
done

lemma skolemize1: \(\forall x. P x \rightarrow (\exists y. Q x y) \Rightarrow \exists f. \forall x. P x \rightarrow Q x \; (f x)\)
apply (rule-tac x = \lambda x. SOME y. Q x y in exI)
apply (fast intro: some1)
done
lemma least-aux: \[
[f \ 0 = (c \# cs, s); \ \forall \ i. \ f \ i \to f \ (Suc \ i); \\
\quad fst(f \ k) = cs; \ \forall \ i<k. \ fst(f \ i) \neq cs \]
\implies \ \forall \ i \leq k. \ (\exists \ p. \ (p \neq []) = (i < k) \ \& \ \fst(f \ i) = p @ cs)
\] apply (rule allI)
apply (induct-tac i)
apply simp
apply (rule ccontr)
apply simp
apply clarsimp
apply (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 clarsimp
apply clarsimp
apply clarsimp
apply simp
apply simp
done

lemma least-lem: \[
[f \ 0 = (c\#cs,s); \ \forall \ i. \ f \ i \to f \ (Suc \ i); \ \exists \ i. \ fst(f \ i) = cs \]
\implies \ \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))) \ \& \\
\quad \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

32
apply (erule least-aux, assumption)
apply (fast intro: LeastI)
apply clarify
apply (drule not-less-Least)
apply blast
done

lemma skolemize2: \( \forall x. \exists y. P x y \implies \exists f. \forall x. P x (f x) \)
apply (rule-tac \( x = \lambda x. \) SOME \( y \). \( P x y \) in exI)
done

lemma inf-cases: \( \inf (c#cs) s \implies \inf [c] s \lor (\exists t. s - c \rightarrow t \land \inf cs t) \)
apply (unfold inf-def)
apply (clarsimp del: disjCI)
apply (case-tac \( \exists i. \) fst(f i) = cs)
apply (drule disjI2)
apply (drule least-lemt, assumption, assumption)
apply clarify
apply (drule exec1s-impl-exec)
apply (case-tac f k)
simp
apply (rule exI, rule conjI, assumption)
apply (rule-tac \( x = \lambda i. f(i+k) \) in exI)
apply (clarsimp)
apply (erule disjI1)
simp
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 (erule-tac \( x = \text{Suc} i \) in spec)
apply (erule-tac f i)
apply (case-tac \( f(Suc i) \))
simp
apply clarify
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = \text{Suc} i \) in spec)
apply (case-tac f i)
apply (case-tac \( f(Suc i) \))
simp
apply clarify
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = \text{Suc} i \) in spec)
apply (case-tac f i)
apply (case-tac \( f(Suc i) \))
simp
apply clarify
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = \text{Suc} i \) in spec)
apply (case-tac f i)
apply (case-tac \( f(Suc i) \))
simp
apply clarify
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = \text{Suc} i \) in spec)
apply (case-tac f i)
apply (case-tac \( f(Suc i) \))
simp
apply clarify
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = \text{Suc} i \) in spec)
apply (case-tac f i)
apply (case-tac \( f(Suc i) \))
simp
apply clarify
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = \text{Suc} i \) in spec)
apply (case-tac f i)
apply (case-tac \( f(Suc i) \))
simp
apply clarify
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = \text{Suc} i \) in spec)
apply (case-tac f i)
apply (case-tac \( f(Suc i) \))
simp
apply clarify
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = \text{Suc} i \) in spec)
apply (case-tac f i)
apply (case-tac \( f(Suc i) \))
simp
apply clarify
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = \text{Suc} i \) in spec)
apply (case-tac f i)
apply (case-tac \( f(Suc i) \))
simp
apply clarify
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = \text{Suc} i \) in spec)
apply (case-tac f i)
apply (case-tac \( f(Suc i) \))
simp
apply clarify
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = \text{Suc} i \) in spec)
apply (case-tac f i)
apply (case-tac \( f(Suc i) \))
simp
apply clarify
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = \text{Suc} i \) in spec)
apply (case-tac f i)
apply (case-tac \( f(Suc i) \))
simp
apply clarify
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = \text{Suc} i \) in spec)
apply (case-tac f i)
apply (case-tac \( f(Suc i) \))
simp
apply clarify
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = \text{Suc} i \) in spec)
apply (case-tac f i)
apply (case-tac \( f(Suc i) \))
simp
apply clarify
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = \text{Suc} i \) in spec)
apply (case-tac f i)
apply (case-tac \( f(Suc i) \))
simp
apply clarify
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = \text{Suc} i \) in spec)
apply (case-tac f i)
apply (case-tac \( f(Suc i) \))
simp
apply clarify
apply (erule-tac \( x = i \) in allE)
apply (erule-tac \( x = i \) in allE)
apply blast
apply (erule-tac \(x = n\) in allE)
apply (erule-tac \(x = \text{Suc } n\) in allE)
apply (case-tac \(f \cdot n\))
apply (case-tac \(f(\text{Suc } n)\))
apply clarsimp
apply (drule exec1-only1)
apply clarsimp
done

lemma termi-impl-not-inf: \(\text{c} \downarrow \text{s} \Rightarrow \neg \text{inf } [\text{c}] \text{s}\)
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:
\(\text{c} \downarrow \text{s} \Rightarrow \neg (\exists \text{f} \cdot \text{f} \ 0 = ([\text{c}], \text{s}) \land (\forall \text{i}::\text{nat}. \ (\text{f} \text{i}, \text{f} (\text{i}+1)) : \text{exec1}^+)\))
apply (subgoal-tac wf {\((\text{y}, \text{x}). ([\text{c}], \text{s}) \to^* \text{x} & \text{x} \to \text{y}\)^+})
apply (simp only:wf-iff-no-infinite-down-chain)
apply (erule contrapos-nn)
apply clarify
apply (subgoal-tac \(\forall \text{i}. ([\text{c}], \text{s}) \to^* \text{f} \text{i}\))
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(rewrite 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:
{ (∀ z. (z=s ∧ body↓s) ∧ (s,t) ∈ termi-call-steps, CALL, λz s. z − body→ s)} ⊢_1
{ (∀ z. (z=s ∧ body↓t) ∧ (∃ cs. ([body],t) →* (c#cs,s)))} c { (∀ z. 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 $s \neg c1 \rightarrow ta$)
prefer 2
apply (blast intro: exec1.Semi exec-impl-execs rtrancl-trans)
apply (subgoal-tac $[[\text{body}]], t \rightarrow^* (c2 \# cs, ta))$
prefer 2
apply (blast intro: exec-impl-execs rtrancl-trans)
apply (blast intro: exec-impl-execs rtrancl-trans execs-pres-termi)
apply (fast intro: exec1.Semi rtrancl-trans)
apply simp only: termi-call-steps-def
apply (rule thoare.Conseq)
apply blast
defer
apply simp
apply (rule thoare.Local)
apply (rule allI)
apply (erule thoare.Conseq)
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 $P' = \lambda z . (z,s) \in ((s,t). b s \land s \rightarrow t}) \ast \land \text{body} \downarrow t \land$
\(\exists cs. ([\text{body}], t) \rightarrow^* ((\text{WHILE } b \text{ DO } c) \# cs, s) \in \text{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-exec rtrancl-trans)
apply (rule conjI)
  apply (rule conjI)
  apply (blast intro: execs-pres-terms)
  apply (blast intro: exec1. WhileTrue exec-impl-exec rtrancl-trans)
apply (rule conjI)
  apply (blast intro: exec1. WhileTrue exec-impl-exec 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 \land \text{body} \downarrow s) \land (s, t) \in \text{termi-call-steps}, \text{CALL}, \lambda z s. z \rightarrow \text{body} \rightarrow s) \} \vdash t
\{ (\lambda z s. (z = s \land \text{body} \downarrow s) \land s = t) \} \text{body} \{ (\lambda z s. z \rightarrow \text{body} \rightarrow s) \}
apply (rule strengthen-pre [OF - CALL-lemma])
apply blast
done

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

theorem \{ \} \vdash_t \{ P \} c\{ Q \} \implies \{ \} \vdash_t \{ P \} c\{ Q:: \text{state assn} \}
apply (erule \text{MGT-implies-complete}[OF \text{MGT-lemma}[OF \text{MGT-CALL}]]
done

end

4 Hoare Logics for Mutually Recursive Procedure

theory PsLang imports Main begin
4.1 The language

typedec state
typedec pname

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 pname
  ∣ Local (state ⇒ state) com (state ⇒ state ⇒ state)
      (∑ state ⇒ state ⇒ state ⇒ state ⇒ state)
  ∣ WhileFalse ¬b s ⇒ s - WHILE b DO c ⇒ s
  ∣ WhileTrue [ b s; s - c ⇒ t; t - WHILE b DO c ⇒ u ]
      ⇒ s - WHILE b DO c ⇒ u

consts body :: pname ⇒ 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 ⇒ com.
- The CALL command now has an argument of type pname, the name of the procedure that is to be called.

inductive
exec :: state ⇒ com ⇒ state ⇒ bool (⇒; - - -; - - - [50, 0, 50] 50)

where
  Do: t ∈ f s ⇒ s - Do f ⇒ t
  ∣ Semi: [ s0 - c1 ⇒ s1; s1 - c2 ⇒ s2 ] ⇒ s0 - c1; c2 ⇒ s2
  ∣ IfTrue: [ b s; s - c1 ⇒ t ] ⇒ s - IF b THEN c1 ELSE c2 ⇒ t
  ∣ IfFalse: [ ¬b s; s - c2 ⇒ t ] ⇒ s - IF b THEN c1 ELSE c2 ⇒ t
  ∣ WhileFalse: ¬b s ⇒ s - WHILE b DO c ⇒ s
  ∣ WhileTrue: [ b s; s - c ⇒ t; t - WHILE b DO c ⇒ u ]
      ⇒ s - WHILE b DO c ⇒ u
  ∣ Call: s - body p ⇒ t ⇒ s - CALL p ⇒ t
  ∣ Local: f s - c ⇒ t ⇒ s - LOCAL f; c; g ⇒ g s t

lemma [iff]: (s - Do f ⇒ t) = (t ∈ f s)
by(auto elim: exec.cases intro:exec.intros)
lemma [iff]: \((s \mathbin{\cdot} c; d \mathbin{\cdot} u) = (\exists t. s \mathbin{\cdot} c \mathbin{\cdot} t \land t \mathbin{\cdot} d \mathbin{\cdot} u)\)
by(auto elim: exec.cases intro:exec.intros)

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

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

lemma [iff]: \((s \mathbin{\cdot} \text{LOCAL } f; c; g \mathbin{\cdot} u) = (\exists t. f s \mathbin{\cdot} c \mathbin{\cdot} t \land u = g s t)\)
by(fastforce elim: exec.cases intro:exec.intros)

inductive
\(\text{execn} :: \text{state} \Rightarrow \text{com} \Rightarrow \text{nat} \Rightarrow \text{state} \Rightarrow \text{bool} \quad (-/-/-/-/-)\)
where
\(\text{Do}: \quad t \in f s \Rightarrow s \mathbin{\cdot} \text{Do } f \mathbin{\cdot} n \mathbin{\cdot} t\)
| \(\text{Semi:} \quad \ll [s0 -c0 -n - s1; s1 -c1 -n - s2 \rr] \Rightarrow s0 -c0; c1 -n - s2\)
| \(\text{IfTrue:} \quad \ll b s; s -c0 -n - t \rr \Rightarrow s -\text{IF } b \text{ THEN } c0 \text{ ELSE } c1 -n - t\)
| \(\text{IfFalse:} \quad \ll \neg b s; s -c1 -n - t \rr \Rightarrow s -\text{IF } b \text{ THEN } c0 \text{ ELSE } c1 -n - t\)
| \(\text{WhileTrue:} \quad \ll b s; s -c -n - t; t -\text{WHILE } b \text{ DO } c -n - u \rr \Rightarrow s -\text{WHILE } b \text{ DO } c -n - u\)
| \(\text{Call:} \quad s -\text{body } p -n - t \Rightarrow s -\text{CALL } p -\text{Suc } n - t\)
| \(\text{Local:} \quad f s -c -n - t \Rightarrow s -\text{LOCAL } f; c; g -n - g s t\)

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

lemma [iff]: \((s \mathbin{\cdot} c1; c2 -n - u) = (\exists t. s -c1 -n - t \land t -c2 -n - u)\)
by(best elim: execn.cases intro:execn.intros)

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

lemma [iff]: \((s \mathbin{\cdot} \text{CALL } p - 0 - t) = \text{False}\)
by (blast elim: execn.cases intro: execn.intros)

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

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

lemma exec-mono [rule-formal]: s − c − m → t =⇒ ∀ n. m ≤ n → s − c − n → t
apply (erule execn.induct)
  apply (blast)
  apply (blast)
  apply (simp)
  apply (simp)
  apply (simp add: execn.intros)
  apply (blast intro: execn.intros)
  apply (clarify)
  apply (rename_tac m)
  apply (case_tac m)
  apply simp
  apply simp
  apply blast
done

lemma exec-iff-execn: (s − c → t) = (∃ n. s − c − n → t)
apply (rule iffI)
apply (erule execn.induct)
  apply blast
  apply clarify
  apply (rename_tac m n)
  apply (rule-tac x = max m n in exI)
  apply (fastforce intro: execn.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: execn.intros)
apply blast
apply blast
done

lemma while-lemma[rule-format]:
s \neg w \rightarrow n \rightarrow t \Longrightarrow \forall b. c. w = \text{WHILE} b \text{ DO } c \land P s \land
(\forall s s'. P s \land b s \land s' \neg c \neg n \rightarrow s' \Longrightarrow P s') \Longrightarrow P t \land \neg b t
apply(erule execn.induct)
apply clarify+
def er
apply clarify+
apply(subgoal-tac P t)
apply blast
apply blast
done

lemma while-rule:
[s \neg \text{WHILE} b \text{ DO } c \neg n \rightarrow t; P s; \land s s'. [P s; b s; s' \neg c \neg n \rightarrow s'] \Longrightarrow P s']
\Longrightarrow P t \land \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 \text{state} \Rightarrow \text{bool}
type-synonym 'a cntxt = ('a assn \times \text{com} \times 'a assn)\text{set}

definition valid :: 'a assn \Rightarrow \text{com} \Rightarrow 'a assn \Rightarrow \text{bool} (|= \{(1-)/\} 50) \text{ where}
|=(P)c{Q} \equiv (\forall z s t. s \rightarrow c \rightarrow t \Longrightarrow P z s \rightarrow Q z t)

definition valids :: 'a cntxt \Rightarrow \text{bool} (|= - 50) \text{ where}
|= D \equiv (\forall (P,c,Q) \in D. |=(P)c{Q})

definition nvalid :: nat \Rightarrow 'a assn \Rightarrow \text{com} \Rightarrow 'a assn \Rightarrow \text{bool} (|=\ (-)/\ \{(1-)/\} 50) \text{ where}
|=n \{P\}c{Q} \equiv (\forall s t z. s \rightarrow c \rightarrow n \rightarrow t \Longrightarrow P z s \rightarrow Q z t)

definition nvalids :: nat \Rightarrow 'a cntxt \Rightarrow \text{bool} (|=\ -/\ - 50) \text{ 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**

\[ \text{cvalids} :: \text{'a cntxt} \Rightarrow \text{'a cntxt} \Rightarrow \text{bool} \; (- \models / \cdot 50) \; \text{where} \]

\[ C \models D \iff \models C \rightarrow \models D \]

**definition**

\[ \text{cnvalids} :: \text{'a cntxt} \Rightarrow \text{nat} \Rightarrow \text{'a cntxt} \Rightarrow \text{bool} \; (- \models / \cdot 50) \; \text{where} \]

\[ C \models -n \; D \iff \models -n \; C \rightarrow \models -n \; D \]

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**

\[ \text{hoare} :: \text{'a cntxt} \Rightarrow \text{'a cntxt} \Rightarrow \text{bool} \; (- \vdash / \cdot 50) \]

and \( \text{hoare}3 :: \text{'a cntxt} \Rightarrow \text{'a assn} \Rightarrow \text{com} \Rightarrow \text{'a assn} \Rightarrow \text{bool} \; (- \vdash / \cdot (\{(I-)\}/ (-)/ \{(I-)\}) \cdot 50) \)

\[ \text{where} \]

\[ C \vdash \{P\}c\{Q\} \iff C \vdash \{(P,c,Q)\} \]

| Do: \( C \vdash \{\lambda z \; s. \; \forall t \in f \; s. \; P \; z \; t\} \; \text{Do} \; \{P\} \)
| Semi: [ \( C \vdash \{P\}c\{Q\}; \; C \vdash \{Q\}d\{R\} \] \( \implies C \vdash \{P\}c; d\{R\} \]
| If: [ \( C \vdash \{\lambda z \; s. \; P \; z \; s \; \land \; b \; s\}c\{Q\}; \; C \vdash \{\lambda z \; s. \; P \; z \; s \; \land \; \neg \; b \; s\}d\{Q\} \] \( \implies C \vdash \{P\} \; \text{IF} \; b \; \text{THEN} \; c \; \text{ELSE} \; d\{Q\} \]
| While: [ \( C \vdash \{\lambda z \; s. \; P \; z \; s \; \land \; b \; s\} \; c \; \{P\} \) \( \implies C \vdash \{P\} \; \text{WHILE} \; b \; \text{DO} \; c \; \{\lambda z \; s. \; P \; z \; s \; \land \; \neg \; b \; s\} \]
| Conseq: [ \( C \vdash \{P\}c\{Q\}; \)
\[ \forall s \; t. \; (\forall z. \; P' \; z \; s \; \rightarrow \; Q' \; z \; t) \rightarrow (\forall z. \; P \; z \; s \; \rightarrow \; Q \; z \; t) \] \( \implies C \vdash \{P\}c\{Q\} \]
| Call: [ \( \forall (P,c,Q) \in C. \; \exists \; p. \; c = \text{CALL} \; p; \)
\[ C \vdash \{(P,b,Q). \; \exists \; p. \; (P,\text{CALL} \; p, \; Q) \in C \; \land \; b = \text{body} \; p\} \] \( \implies \{\} \vdash \) \( C \]
| Asm: [ \( (P,\text{CALL} \; p, \; Q) \in C \) \( \implies C \vdash \{P\} \; \text{CALL} \; p \; \{Q\} \]
| Conj: [ \( \forall (P,c,Q) \in D. \; C \vdash \{P\}c\{Q\} \) \( \implies C \vdash D \]
| ConjE: [ \( C \vdash D; \; (P,c,Q) \in D \) \( \implies C \vdash \{P\}c\{Q\} \]
| Local: [ \( \forall s'. \; C \vdash \{\lambda z \; s. \; P \; z \; s' \; \land \; s = f \; s'\} \; c \; \{\lambda z \; t. \; Q \; z \; (g \; s' \; t)\} \) \( \implies C \vdash \{P\} \; \text{LOCAL} \; f; c; g \; \{Q\} \]

**monos** split-beta
lemmas valid-defs = valid-def valids-def cnvalids-def

nvalid-def nvalids-def cnvalids-def

theorem C ⊢ D ⇒ C ||= D

As before, we prove a generalization of C ||= D, namely ∀ n. C ||= -n D, by induction on C ⊢ D, with an induction on n in the CALL case.

apply (subgoal-tac ∀ n. C ||= -n D)
apply (unfold valid-defs exec-iff-execn THEN eq-reflection)
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 fast

apply fastforce

apply fastforce

done

definition MGT :: com ⇒ state assn × com × state assn where
[simp]: MGT c = (λz s. z = s, c, λz t. z − c → t)

lemma strengthen-pre:
[ ∀ z s. P' z s → P z s; C ⊢ {P} c{Q} ] ⇒ C ⊢ {P'} c{Q}
by (rule hoare.Conseq, assumption, blast)

lemma MGT-implies-complete:
{ } ⊢ {MGT c} ⇒ {P} c{Q} ⇒ { } ⊢ {P} c{Q::state assn}
apply (unfold MGT-def)
apply (erule hoare.Conseq)  
apply (simp add: valid-defs)  
done

lemma MGT-lemma: ∀ p. C ⊢ {MGT(CALL p)} ⟹ C ⊢ {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.Conseq)  
  apply blast  
  apply simp  
  prefer 2  
  apply simp  
  apply (rename-tac b c)  
  apply (rule hoare.Conseq)  
  apply (rule-tac P = λz s. (z,s) ∈ {{s,t}. b s ∧ s −c→ t}^∗ in hoare.While)  
  apply (erule hoare.Conseq)  
  apply (blast intro:rtrancl-into-rtrancl)  
  apply clarsimp  
  apply (rename-tac s t)  
  apply (erule-tac x = s in allE)  
  apply clarsimp  
  apply (erule converse-rtrancl-induct)  
  apply (blast intro:exec.intros)  
  apply (fast elim:exec.WhileTrue)  
apply (fastforce intro: hoare.Local elim!: hoare.Conseq)  
done

lemma MGT-body: (P, CALL p, Q) = MGT (CALL pa) ⟹ C ⊢ {MGT (body p)} ⟹ C ⊢ {P} body p {Q}  
apply clarsimp  
done

declare MGT-def[simp del]
lemma MGT-CALL: \{\} ⊢ \{ \text{mgt} \exists \, \text{p, mgt} = \text{MGT}(\text{CALL} \, \text{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: \{P\} \text{c\{}Q\text{\} ⇒ \{\} ⊢ \{P\} \text{c\{}Q::\text{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]: \{\} ≠ \{\} ⇒ Do \, f ↓ s
| Semi [intro]: \[ c1 ↓ s0; \, \land \, s1. \, s0 \longrightarrow s1 ⇒ c2 ↓ s1 \] \⇒ (c1;c2) ↓ s0
| IfTrue [intro, simp]: \[ b \, s; \, c1 ↓ s \] \⇒ IF \, b \, THEN \, c1 \, ELSE \, c2 ↓ s
| IfFalse [intro, simp]: \[ \neg b \, s; \, c2 ↓ s \] \⇒ IF \, b \, THEN \, c1 \, ELSE \, c2 ↓ s
| WhileFalse: \neg b \, s \⇒ WHILE \, b \, DO \, c \downarrow s
| WhileTrue: \[ b \, s; \, c ↓ s; \, \land \, t. \, s \longrightarrow t \Rightarrow WHILE \, b \, DO \, c ↓ t \] \⇒ WHILE \, b \, DO \, c ↓ s
| body p ↓ s ⇒ CALL p ↓ s
| Local: \, c ↓ f \, s \⇒ LOCAL \, f;\, c;\, g \, ↓ \, s

lemma [iff]: (Do \, f ↓ s) = (f \, s \, \neq \, \{\})
apply (rule iffI)

45
prefer 2
apply (best intro: termi.intros)
apply (erule termi.cases)
apply blast+
done

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

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

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

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

lemma \textit{termi-while-lemma}[rule-format]:
\(w \downarrow fk \implies (\forall k b c. fk = f k \land w = \text{WHILE } b \text{ DO } c \land (\forall i. f i -\rightarrow f(Suc i)) \rightarrow (\exists i. b(f i)))\)
apply (erule termi.induct)
apply simp-all
apply blast
apply blast
done

lemma \textit{termi-while}:
lemma wf-termi: wf \((t,s)\). WHILE \(b\) DO \(c\) \(\downarrow\ s \land b \land s - \rightarrow t\)
apply \(\text{sub} \, \text{wf-iff-no-infinite-down-chain}\)
apply \(\text{rule notI}\)
apply \(\text{clarsimp}\)
apply \(\text{insert} \, \text{termi-while}\)
done
end

theory PsHoareTotal imports PsHoare PsTermi begin

4.4 Hoare logic for total correctness

declaration \(\text{a assn} \Rightarrow \text{com} \Rightarrow \text{a assn} \Rightarrow \text{bool} ((\models_t \{(I-)\})/ (\cdot)/ \{(I-)\}) 50) \) where
\(\models_t \{P\}c(Q) \iff \{P\}c(Q) \land (\forall z. P z \rightarrow c\downarrow s)\)

declaration \(\text{a cntxt} \Rightarrow \text{bool} ((\models_t - 50) \) where
\(\models_t D \iff (\forall (P,c,Q) \in D. \models_t \{P\}c(Q))\)

declaration \(\text{a cntxt} \Rightarrow \text{a assn} \Rightarrow \text{com} \Rightarrow \text{a assn} \Rightarrow \text{bool} ((\models_t/ \{(I-)\})/ (\cdot)/ \{(I-)\}) 50) \) where
\(C \models_t \{P\}c(Q) \iff \models_t C \rightarrow \models_t \{P\}c(Q)\)

declaration \(\text{a cntxt} \Rightarrow \text{a cntxt} \Rightarrow \text{bool} (- ((\models_t - 50) \) where
\(\models_t D \iff \models_t C \rightarrow \models_t D\)

inductive
\(\text{thoare} : \text{a cntxt} \Rightarrow \text{a cntxt} \Rightarrow \text{bool} (\rightarrow ((\models_t -)) 50)\)
and \(\text{thoare} : \text{a cntxt} \Rightarrow \text{a cntxt} \Rightarrow \text{a assn} \Rightarrow \text{com} \Rightarrow \text{a assn} \Rightarrow \text{bool} ((\models_t/ ((\cdot)/ ((I-)\})) [50,0,0,0] 50)\) where
\(\models_t \{P\}c(Q) \equiv C \models_t \{(P,c,Q)\}\)
| Do: \(\models_t \{\lambda z. s. (\forall t \in f s. P z t) \land f s \neq \{\}\\} \) Do \(\{P\}\)
| Semi: \(\models_t \{P\}c1(Q); \models_t \{Q\}c2(R) \Rightarrow \models_t \{P\}c1;c2 \{R\}\)
| If: \(\models_t \{\lambda z s. P z s \land b s\}c(Q); \models_t \{\lambda z s. P z s \land \sim b s\}d(Q) \Rightarrow \models_t \{P\} \) IF \(b\) THEN \(c\) ELSE \(d\) \{Q\}\)
| While: \(\models_t; \forall s'. C \models_t \{\lambda z s. P z s \land b s \land s' = s\} c \{\lambda z s. P z s \land (s,s') \in R\}\) \Rightarrow \(\models_t \{P\} \) WHILE \(b\) DO \(c\) \{\lambda z s. P z s \land \sim b s\}\)

\[
\begin{align*}
\text{Call:} & \quad \forall q \preceq. (\{ (\lambda z, s. P p z s \land \lambda q, \preceq, q) \in r, \text{CALL p, Q p} \}) \\
& \quad \Rightarrow \{ } \quad \forall p. (\{ (P p, \text{CALL p, Q p}) \}) \\
\text{Asm:} & \quad (P, \text{CALL p, Q}) \in C \Rightarrow C \vdash \{ P \} \text{ CALL p } \{ Q \} \\
\text{Conseq:} & \quad C \vdash \{ P \} c\{ Q \}; \\
& \quad (\forall s t, (\forall z. P' z s \rightarrow Q' z t) \rightarrow (\forall z. P z s \rightarrow Q z t)) \land \\
& \quad (\forall s. (\exists z. P z s) \rightarrow (\exists z. P' z s)) \] \\
& \quad \Rightarrow C \vdash \{ P \} c\{ Q \} \\
\text{ConjI:} & \quad \forall (P, c, Q) \in D. C \vdash \{ P \} c\{ Q \} \Rightarrow C \vdash \{ P \} c\{ Q \} \\
\text{ConjE:} & \quad [ C \vdash \{ P \} c\{ Q \}; (P, c, Q) \in D ] \Rightarrow C \vdash \{ P \} c\{ Q \} \\
\text{Local:} & \quad [ \forall s', C \vdash \{ \lambda z, s. P z s' \land s = f s' \} \ c \{ \lambda z. Q z (g s' t) \} ] \Rightarrow \\
& \quad C \vdash \{ P \} \text{ LOCAL f;c;g } \{ Q \} \\
\text{lemma strengthen-pre:} & \quad [ \forall z s. P' z s \rightarrow P z s; C \vdash \{ P \} c\{ Q \} ] \Rightarrow C \vdash \{ P' \} c\{ Q \} \\
& \text{by(rule thoare.Conseq, assumption, blast)} \\
\text{lemma weaken-post:} & \quad [ C \vdash \{ P \} c\{ Q \}; (\forall z. Q z s \rightarrow Q' z s) ] \Rightarrow C \vdash \{ P \} c\{ Q' \} \\
& \text{by(rule thoare.Conseq, blast)} \\
\text{lemmas} & \quad \text{tvalid-defs = tvalid-def ctvalid-def valids-def cvalids-def valid-defs} \\
\text{lemma [iff]:} & \quad (\vdash \{ \lambda z, s. \exists n. P n z s \} c\{ Q \}) = (\forall n. \vdash \{ P n \} c\{ Q \}) \\
& \text{apply(unfold tvalid-defs)} \\
& \text{apply fast} \\
& \text{done} \\
\text{lemma [iff]:} & \quad (\vdash \{ \lambda z, s. P z s \land P' z s \} c\{ Q \}) = (P' \rightarrow \vdash \{ P \} c\{ Q \}) \\
& \text{apply(unfold tvalid-defs)} \\
& \text{apply fast} \\
& \text{done} \\
\text{lemma [iff]:} & \quad (\vdash \{ P \} \text{ CALL p } \{ Q \}) = (\vdash \{ P \} \text{ body p } \{ Q \}) \\
& \text{apply(unfold tvalid-defs)} \\
& \text{apply fast} \\
& \text{done}
\end{align*}
\]
lemma unfold-while:  
\((s \rightarrow \text{WHILE } b \text{ DO } c \rightarrow u) =\)  
\((s \rightarrow \text{IF } b \text{ THEN } c; \text{WHILE } b \text{ DO } c \text{ ELSE } \text{Do}(\lambda s. \{s\}) \rightarrow u)\)  
by (auto elim: exec.cases intro:exec.intro split:if-split-asm)  

theorem C \(\vdash_t D \implies C \vdash_t D\)  
apply (erule thoaer.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 implI, 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-asn-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-asn-use) add:tevalid-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 :: com \Rightarrow state assn \times com \times state assn where
[simp]: MGT_t c = (\lambda z s. z = s \& c \downarrow s, c, \lambda z t. z \rightarrow t)

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

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

lemma while-termiE2: \[ WHILE b DO c \downarrow s; b s; s \rightarrow t \] \Rightarrow WHILE b DO c \downarrow t
by(erule termi.cases, auto)

lemma MGT-lemma: \forall p. \{ \} \models_t \{ MGT_t (CALL p) \} \Rightarrow \{ \} \models_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 \rightarrow com1 \rightarrow s \& com2 \downarrow s in thoare.Semi)
  apply(erule thoare.Conseq)
  apply fast
  apply(erule thoare.Conseq)
apply fast
apply (rule thoa.re.If)
apply (erule thoa.re.Conseq)
apply simp
apply (erule thoa.re.Conseq)
apply simp
defer
apply simp
apply (fast intro:thoa.re.Local elim!: thoa.re.Conseq)
apply (rename-tac b c)
apply (rule-tac P' = λz s. (z,s) ∈ ((s,t), b s ∧ s -c→ t}) ^s ∧ WHEN b DO c ↓ s in thoa.re.Conseq)
apply (rule-tac thoa.re.While[OF wf-termi])
apply (rule allI)
apply (erule thoa.re.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
eexec1' :: (com list × state) ⇒ (com list × state) ⇒ bool (- → - [S1,S1] 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
\texttt{execr} :: \((\text{com list} \times \text{state}) \Rightarrow (\text{com list} \times \text{state}) \Rightarrow \text{bool} \ (- \to^* - [81,81] \ 100)\
\textbf{where}\ cs0 \to^* cs1 \equiv (cs0,cs1) : \text{execr}^* \\

\textbf{inductive-cases} \ \texttt{execrE[elim!]}:\
\begin{align*}
(\text{[],s) } \to (cs',s') \\
(\text{Do f#cs,s) } \to (cs',s') \\
((\text{c1; c2})#cs,s) \to (cs',s') \\
((\text{IF b THEN c1 ELSE c2})#cs,s) \to (cs',s') \\
((\text{WHILE b DO c})#cs,s) \to (cs',s') \\
(\text{CALL p#cs,s) } \to (cs',s') \\
((\text{LOCAL f;c;g})#cs,s) \to (cs',s') \\
\end{align*}

\textbf{lemma} [iff]: \ ~ ([],s) \to u \\
\textbf{by} (\text{induct u) blast} \\

\textbf{lemma} \ \texttt{app-exec} : (cs,s) \to (cs',s') \Rightarrow (cs@cs2,s) \to (cs'@cs2,s') \\
\textbf{apply}(\text{erule \texttt{execr}.induct}) \\
\quad \textbf{apply}(\text{simp-all del;fun-upd-apply}) \\
\quad \textbf{apply}(\text{blast intro:execr1.intros}) \\
\textbf{done} \\

\textbf{lemma} \ \texttt{app-execs} : (cs,s) \to^* (cs',s') \Rightarrow (cs@cs2,s) \to^* (cs'@cs2,s') \\
\textbf{apply}(\text{erule \texttt{rtrancl-induct2})} \\
\textbf{apply} \ \texttt{blast} \\
\textbf{apply}(\text{blast intro:app-exec \texttt{rtrancl-trans})} \\
\textbf{done} \\

\textbf{lemma} \ \texttt{exec-impl-execs[rule-format]}:\ \\
\texttt{s \to c \to s' \Rightarrow \forall cs. (c\#cs,s) \to^* (cs,s')} \\
\textbf{apply}(\text{erule \texttt{execr}.induct}) \\
\quad \textbf{apply} \ \texttt{blast} \\
\quad \textbf{apply}(\text{blast intro:rtrancl-trans}) \\
\quad \textbf{apply}(\text{blast intro:exec1.IFTrue rtrancl-trans}) \\
\quad \textbf{apply}(\text{blast intro:exec1.IFFalse rtrancl-trans}) \\
\quad \textbf{apply}(\text{blast intro:exec1.WhileTrue rtrancl-trans}) \\
\quad \textbf{apply}(\text{blast intro: \texttt{rtrancl-trans})} \\
\textbf{apply}(\text{blast intro: \texttt{rtrancl-trans})} \\
\textbf{done} \\

\textbf{inductive} \ \\
\texttt{execs} :: \text{state} \Rightarrow \text{com list} \Rightarrow \text{state} \Rightarrow \text{bool} \ (- / \to - / [50,0,50] \ 50) \\
\textbf{where}\ s = [] \Rightarrow s \\
| s \to c \to t \Rightarrow t = cs \Rightarrow u \Rightarrow s = c\#cs \Rightarrow u \\

\textbf{inductive-cases} [elim!]: \\
\begin{align*}
\text{\texttt{s} = [\]\Rightarrow t} \\
\text{\texttt{s} \# c\#cs} \Rightarrow t \\
\end{align*}
theorem exec1s-impl-execs: \((c,s) \Rightarrow (\text{exec} t)\) \Rightarrow s \Leftarrow c \Rightarrow \text{exec} 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(blast intro:execs.intros)
apply(blast intro:execs.intros exec.intros)+
done

theorem exec1s-impl-exec: \((c,s) \Rightarrow \text{exec} t\) \Rightarrow s \Leftarrow c \Rightarrow \text{exec} t
by(blast dest: exec1s-impl-execs)

primrec termis :: com list ⇒ state ⇒ bool (infixl \(\Rightarrow\) 60) where
\[\begin{align*}
\text{[]} \Rightarrow s & = \text{True} \\
c \# cs \Rightarrow s & = (c \downarrow s \land \forall t. s \Leftarrow t \Rightarrow cs \uparrow t)
\end{align*}\]

lemma exec1-pres-termis: \((c,s) \Rightarrow \text{exec} t\) \Rightarrow cs \Rightarrow s \Rightarrow cs \Rightarrow 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: \((c,s) \Rightarrow \text{exec} t\) \Rightarrow cs \Rightarrow s \Rightarrow cs \Rightarrow s'
apply(erule rtrancl-induct2)
apply blast
apply(blast dest: exec1-pres-termis)
done

lemma execs-pres-termi: \[(c,s) \Rightarrow (c',s')\] \Rightarrow cs \Rightarrow s \Rightarrow cs \Rightarrow s'
apply(insert execs-pres-termis[of \[c\] - c\# cs', simplified])
apply blast
done

definition termi-call-steps :: ((cname × state) × (cname × state))set where
termi-call-steps = \[\{((q,t),(p,s)). body p \downarrow s \land (\exists cs. (body p), s) \Rightarrow \text{exec} t\}\]

lemma lem:
\forall y. (a,y)\in r^+ \Rightarrow P a \Rightarrow P y \Rightarrow ((b,a) \in \{(y,x). P x \land (x,y)\in r\}^+) = ((b,a) \in \{(y,x). P x \land (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:
\[
\forall i. (a, f i) \in r^* \land (f i, f(Suc i)) : r \Rightarrow b = f 0 \rightarrow (\exists f. f 0 = a \land \forall i. (f i, f(Suc i)) : r)
\]
apply (erule converse-rtrancl-induct)
apply blast
apply (clarsimp)
apply (rule-tac x = \lambda i. case i of 0 \Rightarrow y | 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^* \land (f i, f(Suc i)) : r \Rightarrow \exists f. f 0 = a \land \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 \leftarrow (\exists f. f 0 = (cs, s) \land \forall i. f i \rightarrow f(Suc i))

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

lemma [iff]: \sim 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

54
apply (unfold inf-def)
apply (rule iffI)
apply clarify
apply (rule-tac x = λi. 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 = λi. 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 = λi. case i of 0 ⇒ ((c1;c2)#cs, s) | Succ i ⇒ f i in exI)
apply (simp add: exec1.intros 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 = λi. f (Suc i) in exI)
apply clarsimp
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 ⇒ ((IF b THEN c1 ELSE c2)#cs, s) | Succ 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 = λi. f(Suc i) in exI)
apply clarsimp
apply clarsimp
apply (rule-tac x = λi. f (Suc i) in exI)
apply clarsimp
apply (clarsimp split:if-splits)
apply (rule-tac x = λi. case i of 0 ⇒ ((WHILE b DO c)#cs, s) | Succ i ⇒ f i in exI)

55
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(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(case-tac F (Suc 0))
apply(clarsimp)
apply(clarsimp)
apply(rename-tac F)
apply(case-tac F)
apply(rename-tac F)
apply(case-tac)
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 [iff]: inf ((LOCAL f;c;g)#cs) s =
inf (c#Do(λt. {g * t})#es) (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(case-tac F)
apply(rename-tac F)
apply(case-tac)
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: (cs,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:
apply \text{erule exec1.induct}
  apply \text{force intro:exec1.intros simp add: neq-Nil-conv}+
done

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

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

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

\textbf{lemma skolemize1:} \(\forall x.\ P\ x \rightarrow (\exists y.\ Q\ x\ y) \implies \exists f.\ \forall x.\ P\ x \rightarrow Q\ x\ (f\ x)\)
apply\text{(rule-tac x = \lambda x.\ \text{SOME y.\ Q\ x\ y in cxI})}
apply\text{(fast intro:someI2)}
done

\textbf{lemma least-aux:} \[ f\ 0\ (c \neq cs, s); \forall i. f\ i \rightarrow f\ (Suc\ i)\; \\
\implies \forall i < k.\ (\exists p.\ (p\neq[])\ = (i < k) \&\ \text{fst}(f\ i) = p\ \text{fs}\ cs)\]
apply\text{(rule allI)}
apply\text{(induct-tac i)}
apply\text{simp}
apply\text{(rule econtr)}
apply\text{simp}
apply\text{clarsimp}
apply\text{clarsimp}
apply\text{(drule order-le-imp-less-or-eq)}

57
apply(erule disjE)
pref 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 cs1 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:
\[
\begin{align*}
&\{ f 0 = (c\#cs, s) ; \forall i. f i \rightarrow f(Suc i) ; \exists i. fst(f i) = cs \} \\
\rightarrow \exists k. fst(f k) = cs & (\exists \bar{c}. s \rightarrow^{\star} ([], \text{snd}(f k))
\end{align*}
\]
apply(rule_tac x=LEAST i. fst(f i) = cs in exI)
apply(rule conjI)
apply(fast intro: LeastI)
apply(clarsimp)
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(clarsimp)
apply(rename-tac not-less-Least)
apply blast
done

lemma skolemize2:
\[
\forall x. \exists y. P x y \Rightarrow \exists f. \forall x. P x (f x)
\]
apply(rule_tac x = \lambda x. SOME y. P x y in exI)
apply(fast intro:someI2)
done

58
lemma inf-cases: \( \inf (c \# cs) s \implies \inf [c] s \lor (\exists t. s \rightarrow c \rightarrow t \land \inf cs t) \)
apply (unfold inf-def)
apply (clarsimp del: disjCI)
apply (case-tac \( \exists i. \text{fst}(f i) = cs \))
apply (rule disjI2)
apply (drule least-lem, assumption, assumption)
apply clarify
apply (drule exec1s-impl-exec)
apply (case-tac \( \exists i. f i \))
apply simp
apply (rule exI, rule conjI, assumption)
apply (rule-tac \( \lambda i. f(i+k) \) in exI)
apply (clarsimp)
apply (rule disjI1)
apply simp
apply (erule-tac \( \forall i. \exists p. p \neq [] \land \text{fst}(f i) = p \# cs \) in allE)
apply (drule skolemize2)
apply clarify
apply (rename-tac p)
apply (erule-tac \( \exists i. (p i, \text{snd}(f i)) \) in exI)
apply (erule conjI)
apply (erule-tac \( \lambda x. f(i+k) \) in allE, erule conjE)
apply simp
apply clarify
apply (erule-tac \( \lambda x. f(i+k) \) in allE)
apply (erule-tac \( \lambda x. f(i+k) \) in allE)
apply (frule-tac \( \lambda x. f(i+k) \) in spec)
apply (erule-tac \( \lambda x. f(Suc i) \) in allE)
apply (case-tac f i)
apply (case-tac f (Suc i))
apply (erule-tac \( \lambda x. f(Suc i) \) in allE)
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 \( \lambda x. f(Suc n) \) in allE)
apply (erule-tac \( \lambda x. f(Suc n) \) in allE)
apply (case-tac f n)
apply (case-tac f (Suc n))
apply clarsimp
apply (drule exec1-only1)
apply clarsimp
done

lemma termi-impl-not-inf: \( c \downarrow s \implies \neg \inf [c] s \)
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:
  $$c↓s = \neg\exists f. f\ 0 = ([c],s) \land (\forall i::nat. (f\ i, f(i+1)) : exec1^-+)$$

apply (subgoal-tac wf $\{(y,x). ([c],s) \rightarrow^* x \land x \rightarrow y\}^+\) $)

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 (rule-tac x = f in exI)
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 \(\text{wf-termi-call-steps} \colon \text{wf termi-call-steps}\)
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
\[\exists C_s. C_s 0 = [] \& (\forall i. (\text{body}(\text{fst}(S i)) \# C_s i, \text{snd}(S i)) \rightarrow^* (\text{CALL}(\text{fst}(S(i+1)))\# C_s(i+1), \text{snd}(S(i+1))))\])
prefer 2
apply (rule-tac \(x = cseq S\) in exI)
apply clarsimp
apply (erule-tac \(x = i\) in allE)
apply clarsimp
apply (rename-tac q t p s cs)
apply (erule-tac P = \(\lambda cs. ([\text{body} p], s) \rightarrow^* (\text{CALL} p\# cs, t)\) in someI2)
apply (fastforce dest: execs-pres-termi)
apply (drule termi-impl-no-inf-chain)
apply simp
apply blast
apply blast
apply blast
apply blast
apply blast
done

lemma \(\text{CALL-lemma}\):
\[\bigcup p. \{(\lambda z s. (z=s \& \text{body} p \downarrow s) \land ((p,s),(q,\text{pre})) \in \text{termi-call-steps}, \text{CALL} p, \lambda z s. (z - \text{body} p \rightarrow s)\}) \vdash_t\]
\[\{\lambda z s. (z=s \& \text{body} q \downarrow \text{pre}) \land (\exists cs. ([\text{body} q], \text{pre}) \rightarrow^* (c\#cs, s))\}\]
\[c \& (\lambda z s. z - c \rightarrow s)\]
apply (induct-tac c)
apply (rule strengthen-pre[OF - thoare.\(\text{Do}\)])
apply (blast dest: execs-pres-termi)
apply (rename-tac c1 c2)
apply (rule-tac \(Q = \lambda z s. \text{body} q \downarrow \text{pre} \& (\exists cs. ([\text{body} q], \text{pre}) \rightarrow^* (c2\#cs, s))\) &
\( z \rightarrow c_1 \rightarrow s \land c_2 \rightarrow s \ \text{in thoare}_{\text{Semi}} \)

apply (erule thoare.Conseq)
apply (rule conjI)
apply (clarsimp)
apply (subgoal-tac \( s \rightarrow c_1 \rightarrow t \))
prefer 2
apply (blast intro: exec1.Semi exec-impl-execs rtrancl-trans)
apply (subgoal-tac ([body q], pre) \( \rightarrow^* (c_2 \land cs, t) \))
prefer 2
apply (blast intro: exec1.Semi[THEN r-into-rtrancl] exec-impl-execs rtrancl-trans)
apply (subgoal-tac ([body q], pre) \( \rightarrow^* (c_2 \land cs, t) \))
prefer 2
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)

deferr
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 \( P' = \lambda z s. \{(s,t). b s \land s \rightarrow c \rightarrow t\}\) \( \rightarrow^* \) body q \( \rightarrow^* (\exists cs. (\{body q, pre\} \rightarrow^* (((WHILE b DO c) \land cs, s))) \) in thoare.Conseq)
apply (rule-tac thoare. While [OF wf-termi])
apply (rule allI)
apply (erule thoare. Conseq)
apply clarsimp
apply (rule conjI)
apply (erule 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 (rule conjI)
apply (blast intro: exec1. WhileTrue exec-impl-execs rtrancl-trans)
apply (rule conjI)
apply (blast intro: exec1. WhileTrue exec-impl-execs rtrancl-trans)
apply (rule conjI)
apply (fast elim: exec. WhileFalse)
apply (fast intro: rtrancl-refl)
done

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

lemma MGT-CALL: \{p\} \vdash_{s} (\bigcup p. \{\text{MGT}\text{'}( \text{CALL } p)\})
apply (simp add: MGT\text{'}-def)
apply (rule thoare. Call)
apply (rule wf-termi-call-steps)
apply clarify
apply (rule CALL-cor)
done

lemma MGT-CALL1: \forall p. \{p\} \vdash_{s} \{\text{MGT}\text{'}( \text{CALL } p)\}
by (fastforce intro: MGT-CALL [THEN ConjE])

theorem \{p\} \vdash_{s} \{P\} \text{c}\{Q\} \Rightarrow \{p\} \text{c}\{Q::\text{state assn}\}
apply (erule MGT-implies-complete [OF MGT-lemma [OF MGT-CALL1]])

63
References

[1] T. Nipkow. Hoare logics for recursive procedures and unbounded nonde-
terminism. In J. Bradfield, editor, Computer Science Logic (CSL 2002),

[2] T. Nipkow. Hoare logics in Isabelle/HOL. In H. Schwichtenberg and

In C. P. Rangan, V. Raman, and R. Ramanujam, editors, Foundations
of Software Technology and Theoretical Computer Science (FST&TCS),