Computer Supported Modeling and Reasoning

David Basin, Achim D. Brucker, Jan-Georg Smaus, and Burkhart Wolff

April 2005

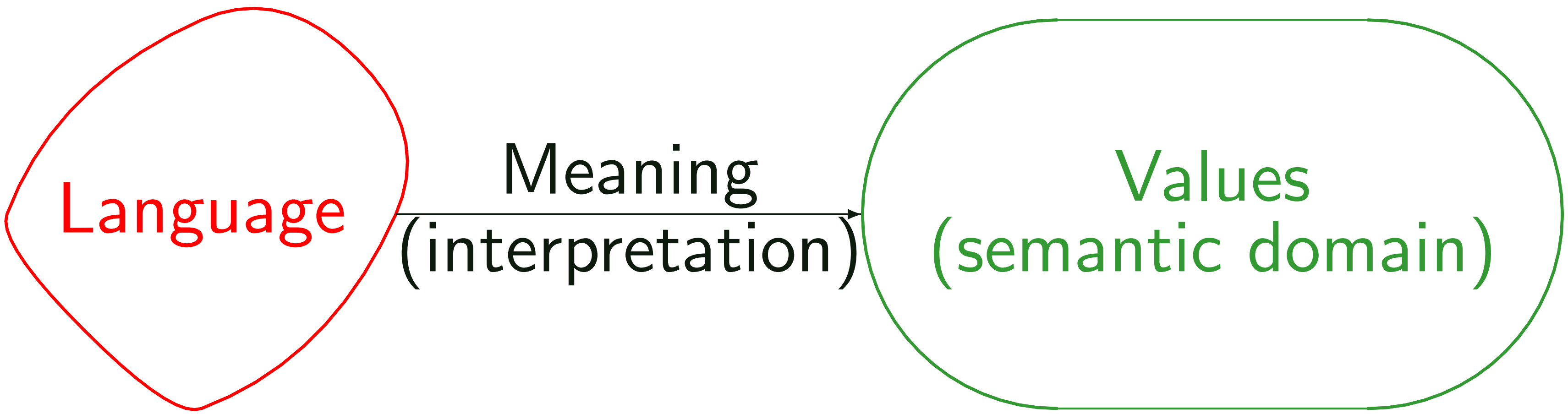
http://www.infsec.ethz.ch/education/permanent/csmr/

Higher-Order Logic Applications: IMP

Jan Smaus and Burkhart Wolff

Language Semantics: An Introduction

Question: What is the meaning of a (programming or specification) language?



- Syntax: language = set of symbols
- Semantics: set of denotations, the semantic domain
- Meaning is a function (interpretation Sem) relating them.

Terminology

- embedding: a theory representing syntax and semantics of a language in a theorem prover.
- object-language is the language to be represented,meta-language the language used for this (e.g. Pure for HOL)
- a deep embedding declares syntax as a data type and defines an explicit interpretation Sem
- a shallow embedding just provides the semantic functions for the operations of the language (no own syntax; e.g. variables are represented by meta-language variables).

Imperative Languages in the Isabelle/HOL Library

There are several embeddings of imperative languages in Isabelle/HOL [Nip02]:

- Hoare: shallowish, good examples
- IMP: deepish, good theory
- IMPP: extends IMP with procedures
- MicroJava: deep, complex, powerful, state-of-the-art

We choose IMP to learn a bit about "good ole imperative languages".

IMP offers:

- operational semantics;
  - natural semantics[Plo81, CDTK86];
  - transition semantics[Plo81];

- denotational semantics;
- axiomatic semantics (Hoare logic);
- equivalence proofs;
- weakest preconditions and verification condition generator.

It closely follows the standard textbook [Win96].

The Command Language (Syntax)

The (abstract) syntax is defined in Com.thy.

Com = Main +	**datatype** com =
typedecl loc	SKIP
types	| ":==" loc aexp (**infixl** 60)
val = nat $(*arb.*)$	| Semi com com ("_ ; _" [60, 60]10)
state = loc⇒val	| Cond bexp com com
aexp = state⇒val	("IF _ THEN _ ELSE _"60)
bexp = state⇒bool	| While bexp com ("WHILE _ DO _"60)

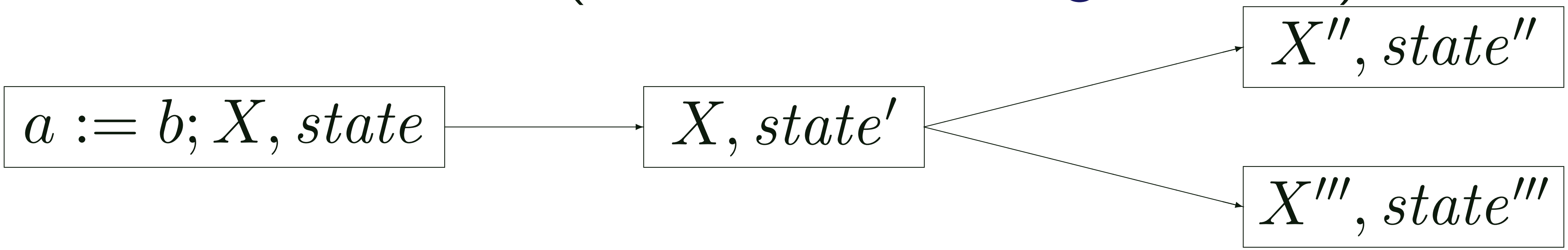
The type loc stands for locations. Note expressions are represented using the shallow technique. The datatype com stands for commands (command sequences).

Operational Semantics: Two Kinds

Natural semantics (idea: a program relates states):

evalc :: (com × state × state) set

Transition semantics (idea: relates "configurations"):



evalc1 :: ((com × state) × (com × state)) set

Natural Semantics

The transition relation of natural semantics is inductively defined.

This means intuitively: The given steps are the only steps.

consts evalc :: (com $\times$state $\times$ state) set
translations "<cm,s> −c−> s' " $\equiv$"(cm,s,s') $\in$ evalc"

We now start giving the actual inductive definition of the natural semantics transition relation . . .

Natural Semantics: Skip and Assignment

inductive evalc

intrs

Skip: ⟨SKIP, s⟩ −c−> s

Assign ⟨x :== a,s⟩ −c−> s[x::=(a s)]

Note that s[x::=(a s)] is an abbreviation for update s x (a s), where

update s x v ≡ λ y. if y=x then v else s y

Note that a is of type aexp or bexp.

Natural Semantics: Sequential Composition

Semi ⟦ <c_0,s> −c−> s_1; <c_1,s_1> −c−> s_2 ⟧
⟹ <c_0;c_1, s> −c−> s_2

Rationale of natural semantics:

- "jump" via c_0 from s to s_1, . . .
- . . . then 'jump" via c_1 from s_1 to s_2

Natural Semantics: Control Statements

IfTrue $$\llbracket\ b\ s;\ <c\_0,s>\ -c->\ s\_1\ \rrbracket$$
$$\Longrightarrow\ <\texttt{IF}\ \ b\ \texttt{THEN}\ c\_0\ \texttt{ELSE}\ c\_1,\ s>\ -c->\ s\_1$$

IfFalse $$\llbracket\ \neg b\ s;\ <c\_1,s>\ -c->\ s\_1\ \rrbracket$$
$$\Longrightarrow\ <\texttt{IF}\ \ b\ \texttt{THEN}\ c\_0\ \texttt{ELSE}\ c\_1,\ s>\ -c->\ s\_1$$

WhileFalse $$\llbracket \neg b\ s \rrbracket$$
$$\Longrightarrow\ <\texttt{WHILE}\ b\ \texttt{DO}\ c,\ s>\ -c->\ s$$

WhileTrue $$\llbracket\ b\ s;\ <c,s>\ -c->\ s\_1;$$
$$<\texttt{WHILE}\ b\ \texttt{DO}\ c, s\_1>\ -c->\ s\_2\ \rrbracket$$
$$\Longrightarrow\ <\texttt{WHILE}\ b\ \texttt{DO}\ c,\ s>\ -c->\ s\_2$$

Note that for non-terminating programs no final state can be derived !

Transition Semantics

Transition semantics relates (inductively defined) "configurations".

consts evalc1 :: $((\text{com} \times \text{state}) \times (\text{com} \times \text{state}))$ set
translations "cs_0 -1-> cs_1" $\equiv$ "(cs_0, cs_1) $\in$ evalc1"

We now start giving the actual inductive definition . . .

Transition Semantics: Assignment, Sequential Composition

inductive evalc1

```
  intro
  Assign    (x:==a,s)   -1-> (SKIP, s[x::=a s])
  Semi1     (SKIP;c,s)  -1-> (c,s)
  Semi2     (c_0,s)     -1-> (c_2,s_1)
            ==>  (c_0;c_1,s) -1-> (c_2;c_1,s_1)
```

Rationale of Transition Semantics:

- the first component in a configuration represents a program counter . . .
- transition semantics is close to an abstract machine.

Transition Semantics: Control Statements

IfTrue

$b\ s \implies$ (`IF b THEN c_1 ELSE c_2,s`) –1–> (c_1,s)

IfFalse

$\neg b\ s \implies$ (`IF b THEN c_1 ELSE c_2,s`) –1–> (c_2,s)

WhileFalse

$\neg b\ s \implies$ (`WHILE b DO c,s`) –1–> (`SKIP`,s)

WhileTrue

$b\ s \implies$ (`WHILE b DO c,s`) –1–> (c;`WHILE` b `DO c,s`)

A non-terminating loop always leads to successor configurations . . .

Generalized Step Relations

- n-step semantics:

  "cs_0 $-$n$->$ cs_1" $==$ "(cs_0,cs_1) $\in$evalc1^n"

- multistep-semantics:

  "cs_0 $-*->$ (c_1,s_1)" $\equiv$"(cs_0,c_1,s_1) $\in$ evalc1^$*$"

Equivalence

Natural semantics vs. transition semantics.

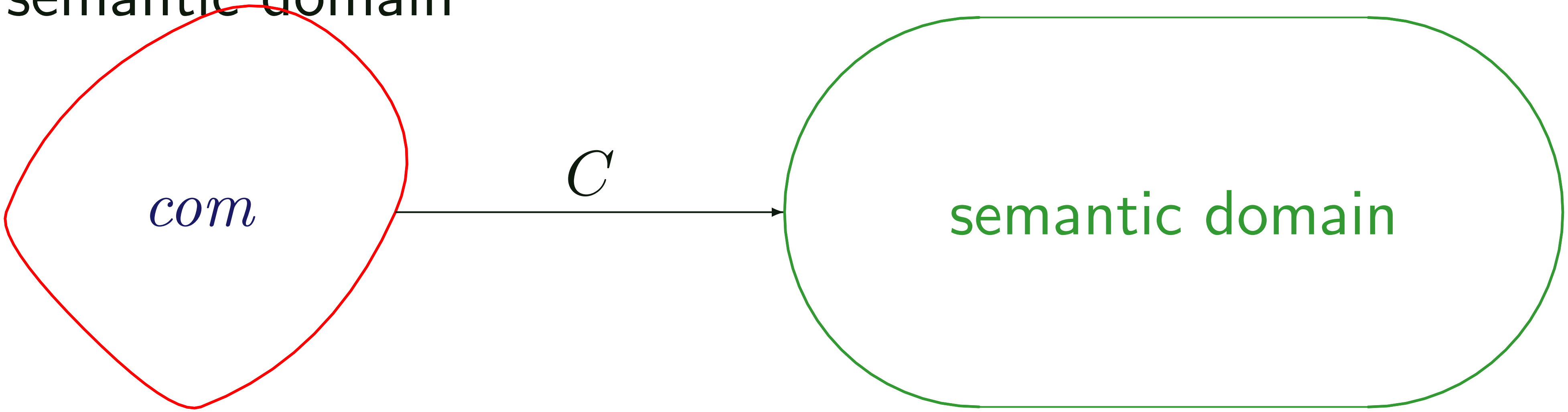
Theorem 1 (`evalc1_eq_evalc`):

(c, s) −∗−> (SKIP, t) = (<c,s> −c−> t)

Proof: by induction over com.

Denotational Semantics

Idea: Explain recursion as fixpoint construction on the semantic domain



An (imperative) semantic domain is a state relation:

$$\text{com\_den} = (\text{state} \times \text{state})\ \text{set}$$

Semantic function:

consts C :: com $\Rightarrow$ com_den

The Inductive Definition

The semantics C is defined inductively:

primrec

C_skip $C(\texttt{SKIP}) = \text{Id}$

C_assign $C(x :== a) = \{(s,t).\ t = s[x::=a(s)]\}$

C_comp $C(c\_0\ ;\ c\_1) = C(c\_1)\ O\ C(c\_0)$

C_if $C(\ \texttt{IF}\ \ b\ \texttt{THEN}\ c\_1\ \texttt{ELSE}\ c\_2) =$
$\{(s,t).\ (s,t) \in C(c\_1) \wedge b(s)\} \cup$
$\{(s,t).\ (s,t) \in C(c\_2) \wedge \neg b(s)\}$"

C_while $C(\ \texttt{WHILE}\ b\ \texttt{DO}\ c) = \text{lfp}\ (\text{Gamma}\ b\ (C\ c))$"

where

$\text{Gamma}\ b\ cd \equiv (\lambda phi.\{(s,t).\ (s,t) \in (phi\ \ O\ cd) \wedge b(s)\} \cup$
$\{(s,t).\ s{=}t \wedge \neg b(s)\})$

Equivalence of Programs

The following is an equivalence relating program fragments.

Theorem 2 (`C_While_If`):

C (`WHILE b DO c`) =
C (`IF b THEN c; WHILE b DO c ELSE SKIP`)

Such results justify program transformations.

Equivalence of Semantics

It turns out that denotational and natural semantics are equivalent in the following sense:

Theorem 3 (denotational is natural):

((s, t) $\in$ C c) = (<c,s> −c−> t)

Still, if we want to prove properties of states to hold, we need different proof techniques. An answer to this need is axiomatic semantics or Hoare Logics.

Axiomatic Semantics

Idea:we relate “legal states” before and after a program execution. A set of legal states is an “assertion”:

types assn $=$ state $\Rightarrow$ bool

Hoare Logics

The key concept of a Hoare Logics is a Hoare Triple

consts hoare :: (assn $\times$com $\times$assn) set
translations "|− {P}c{Q}" $\equiv$"(P,c,Q) $\in$hoare"

A triple has the intuitive meaning: if P holds for some state s and c terminates and reaches some state s' then Q must hold for s'.

The "logic" itself is an inductive definition: . . .

Hoare Logics: SKIP

inductive hoare

intro

skip "|– {P} `SKIP` {P}"

. . .

Hoare Logics: Assignment

ass "|− {λs. P(s[x::=(a s)])} x:==a {P}"

This may be counter-intuitive but but consider the example $a \equiv \lambda s.1$ and $P \equiv \lambda s.\ s\,x = 1$

$$\{\lambda s.(\lambda s.s\,x = 1)(s[x ::= 1])\}x :== \lambda s.1\{\lambda s.s\,x = 1\} \longrightarrow_\beta$$
$$\{\lambda s.(s[x ::= 1])x = 1\}x :== \lambda s.1\{\lambda s.s\,x = 1\} \longrightarrow_\beta$$
$$\{\lambda s.\mathit{True}\}x :== \lambda s.1\{\lambda s.s\,x = 1\}$$

Hoare Logics: Sequence,IF

semi $\llbracket\ \vdash \{P\}c\{Q\};\ \vdash \{Q\}d\{R\}\ \rrbracket$
$\Longrightarrow\ \vdash \{P\}\ c;d\ \{R\}$

If $\llbracket\ \vdash \{\lambda s.\ P\ s \wedge b\ s\}c\{Q\};$
$\vdash \{\lambda s.\ P\ s \wedge \neg b\ s\}d\{Q\}\rrbracket$
$\Longrightarrow\ \vdash \{P\}$ `IF b THEN c ELSE d` $\{Q\}$"

The rule for `IF` represents, as expected, the case split.

Hoare Logics: `WHILE`

While

$$\llbracket \ \vdash \{\lambda s.\ P\ s \wedge b\ s\}\ c\ \{P\}\ \rrbracket$$
$$\Longrightarrow \vdash \{P\}\ \texttt{WHILE}\ b\ \texttt{DO}\ c\ \{\lambda s.\ P\ s \wedge \neg b\ s\}\text{''}$$

If `WHILE` terminates, then the contrary of the condition must hold. . .

Hoare Logics: Consequence Rule

conseq $\llbracket\ \forall s.\ P'\ s \longrightarrow P\ s;$
$\quad\ |\!-\ \{P\}c\{Q\};$
$\quad\ \forall s.\ Q\ s \longrightarrow Q'\ s\ \rrbracket$
$\Longrightarrow |\!-\ \{P'\}c\{Q'\}$

One can always strengthen the pre-condition or weaken the post-condition.

Hoare Logics at a Glance

inductive hoare

intro

skip $\vdash \{P\}\texttt{SKIP}\{P\}$

ass $\vdash \{\lambda s.\ P(s[x::=a\ s])\}\ x:==a\ \{P\}$

semi $[\![\vdash \{P\}c\{Q\}; \vdash \{Q\}d\{R\}]\!] \Longrightarrow \vdash \{P\}\ c;d\ \{R\}$

If $[\![\vdash \{\lambda s.\ P\ s \wedge b\ s\}c\{Q\}; \vdash \{\lambda s.\ P\ s \wedge \neg b\ s\}d\{Q\}]\!]$
$\Longrightarrow \vdash \{P\}\ \texttt{IF}\ b\ \texttt{THEN}\ c\ \texttt{ELSE}\ d\ \{Q\}$

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

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

$\forall$s. Q s $\longrightarrow$Q' s$\rrbracket \Longrightarrow$ |– {P'}c{Q'

Validity Relation

We define a validity relation:

$$\models \{P\}c\{Q\} \equiv \forall s.\ \forall t.\ (s,t) \in C(c) \longrightarrow P\ s \longrightarrow Q\ t"$$

Validity represents our intuition of what Hoare triples mean: whenever the program c can make a transition from s to t (wrt. to the underlying operational/denotational semantics), and whenever P holds for s, Q must hold for t.

Relating Hoare and Denotational Semantics

Theorem 4 (Hoare soundness):

$$\vdash \{P\}\, c\, \{Q\} \Longrightarrow \models \{P\}\, c\, \{Q\}$$

Theorem 5 (Hoare relative completeness):

$$\models \{P\}\, c\, \{Q\} \Longrightarrow \vdash \{P\}\, c\, \{Q\}$$

Why relative?

So the Hoare relation is in fact compatible with the denotational semantics of IMP.

Example Program

```
tm   :== λs. 1;
sum  :== λs. 1;
i    :== λs. 0;
 WHILE λs. (s sum) <= (s a) DO
  (i   :== λs. (s i) + 1;
   tm  :== λs. (s tm) + 2;
   sum :== λs. (s tm) + (s sum))
```

What does this program do?

Try $a = 1$, $a = 2$, . . . , and look at i!

Square Root

Answer: The program computes the square root. Informally:

$$\begin{aligned} Pre &\equiv \text{"}True\text{"} \\ Post &\equiv \text{"}i^2 \leq a < (i+1)^2\text{"} \end{aligned}$$

Formally

$$\begin{aligned} Pre &\equiv \lambda s.\ \ True \\ Post &\equiv \lambda s.\ \ (s\ i) * (s\ i) \leq (s\ a) \wedge \\ &\qquad\quad s\ a < (s\ i + 1) * (s\ i + 1) \end{aligned}$$

Proving $\{Pre\} \ldots \{Post\}$

When using the Hoare-calculus directly:

- we apply the rules following the syntax
- we can apply the conseq-rule initially (interfacing Pre and $Post$
- we can apply the conseq-rule when entering a `WHILE` (interfacing the invariant)

Abbreviation: $ExC \equiv \lambda s.Inv\ s \wedge \neg s\ sum \leq s\ a$ ("exit condition"). We will develop the proof and from its structure "guess" the Invariant.

Proof

$$
\dfrac{\boxed{\mathcal{I}_1} \qquad \dfrac{\boxed{\mathcal{A}_1} \qquad \dfrac{\boxed{\mathcal{A}_2} \qquad \dfrac{\boxed{\mathcal{A}_3} \qquad \dfrac{\boxed{\mathcal{I}_3} \quad \{Inv\}\boxed{WH\ldots}\{ExC\} \quad \boxed{\mathcal{I}_4}}{\{PW\}\boxed{WH\ldots}\{ExC\}}\,conseq}{\{\lambda s.PW(s[''i''])\}\boxed{i\ldots}\{ExC\}}\,semi}{\{\lambda s.PW(s[''i,sum''])\}\boxed{sum\ldots}\{ExC\}}\,semi}{\{\lambda s.PW(s[''i,sum,tm''])\}\boxed{tm\ldots}\{ExC\}}\,semi \qquad \boxed{\mathcal{I}_2}}{\{Pre\}\boxed{tm\ldots}\{Post\}}\,conseq
$$

Completing the Proof

$\boxed{\mathcal{A}_1}$, $\boxed{\mathcal{A}_2}$ and $\boxed{\mathcal{A}_3}$ are complete, and $\boxed{\mathcal{I}_4}$ is trivial.

$\boxed{\mathcal{I}_1}$, $\boxed{\mathcal{I}_2}$, $\boxed{\mathcal{I}_3}$, and $\{Inv\}\boxed{\;WH\ldots\;}\{ExC\}$ remain to be shown.

This also involves the question how the metavariables must be instantiated.

What is PW?

The metavariable PW ("precondition of $WHILE$") must fulfill (to show $\boxed{\mathcal{I}_1}$)

$$\forall s. Pre\ s \rightarrow PW(s[i ::= 0][sum ::= 1][tm ::= 1])$$

Solution (recall that $Pre \equiv \lambda s. True$):

$$PW = \lambda s. s\ i = 0 \wedge s\ sum = 1 \wedge s\ tm = 1$$

What is Inv?

Continuing our proof tree construction:

$$\dfrac{\begin{array}{c}\{\lambda s. Inv\ s \wedge s\ sum \leq s\ a\} i :== \lambda s. s\ i + 1 \{P'\} \\ \{P'\} tm :== \lambda s. s\ tm + 2 \{P''\} \\ \{P''\} sum :== \lambda s. s\ tm + s\ sum \{Inv\}\end{array}}{\dfrac{\{\lambda s. Inv\ s \wedge s\ sum \leq s\ a\} \boxed{\text{"body"}} \{Inv\}}{\{Inv\} \boxed{WH \ldots} \{ExC\}}\ While}\ semi^2$$

Just blindly applying $semi$ twice gives three formulas to be proven using ass, one for each assignment in the loop.

Now what are P' and P''? Have a look at rule ass first!

Calculating P' and P'' (by Rule ass)

$$P'' = \lambda s.Inv(s[sum ::= s\ tm + s\ sum])$$

$$\begin{aligned}
P' &= \lambda s'.P''(s'[tm ::= s'\ tm + 2]) \qquad \text{(rule } ass\text{)}\\
&= \lambda s'.(\lambda s.Inv(s[sum ::= s\ tm + s\ sum]))\\
&\qquad (s'[tm ::= s'\ tm + 2])\\
&= \lambda s'.Inv((s'[tm ::= s'\ tm + 2])\\
&\qquad [sum ::= (s'[tm ::= s'\ tm + 2])\ tm +\\
&\qquad (s'[tm ::= s'\ tm + 2])\ sum])\\
&= \lambda s'.Inv(s'[tm ::= s'\ tm + 2]\\
&\qquad [sum ::= s'\ tm + 2 + s'\ sum]).
\end{aligned}$$

Applying ass to $i :== \lambda s. s\ i + 1$

Now treat $i :== \lambda s. s\ i + 1$ in the same way. Temporarily, let's write P for $\lambda s. Inv\ s \wedge s\ sum \leq s\ a$. Recall $P' =$

$$\lambda s. Inv(s[tm ::= s\ tm + 2][sum ::= s\ tm + 2 + s\ sum]).$$

$$\begin{aligned}
P &= \lambda s'. P'(s'[i ::= s'\ i + 1]) \qquad \text{(by rule } ass\text{)}\\
&= \lambda s'. (\lambda s. Inv(s[tm ::= s\ tm + 2][sum ::= s\ tm + 2 + s\ sum]))\\
&\qquad\qquad (s'[i ::= s'\ i + 1])\\
&= \lambda s'. Inv((s'[i ::= s'\ i + 1])\\
&\qquad [tm ::= (s'[i ::= s'\ i + 1])\ tm + 2]\\
&\qquad [sum ::= (s'[i ::= s'\ i + 1])\ tm + 2 + (s'[i ::= s'\ i + 1])\ sum]))\\
&= \lambda s. Inv(s[i ::= s\ i + 1][tm ::= s\ tm + 2][sum ::= s\ tm + 2 + s\ sum]).
\end{aligned}$$

So Inv must solve this equation.

Inv Must Fulfill the Equation

Inv must fulfill the equation

$$
\begin{aligned}
&\forall s. Inv\ s \wedge s\ sum \leq s\ a = \\
&\forall s. Inv (s[i ::= s\ i + 1][tm ::= s\ tm + 2] \\
&\qquad\qquad\qquad [sum ::= s\ tm + 2 + s\ sum])
\end{aligned}
$$

We can replace λ by $\forall$ due to extensionality.

Guessing the right Inv is obviously difficult! Informally

$$Inv \ \equiv \ "(i+1)^2 = sum \ \wedge \ tm = (2 * i) + 1 \ \wedge \ i^2 \leq a"$$

Checking that Inv Fulfills equation

$$s\,sum \leq s\,a \quad \wedge \tag{1}$$

$$(s\,i + 1)^2 = (s\,sum) \quad \wedge \tag{2}$$

$$s\,tm = (2 * (s\,i)) + 1 \quad \wedge \tag{3}$$

$$(s\,i)^2 \leq (s\,a) \quad \wedge \tag{4}$$

$$\text{(recall: } = \text{ means } \leftrightarrow) \quad = \tag{5}$$

$$((s\,i + 1) + 1)^2 = (s\,sum) + (s\,tm) + 2 \quad \wedge \tag{6}$$

$$(s\,tm + 2) = (2 * (s\,i + 1)) + 1 \quad \wedge \tag{7}$$

$$(s\,i + 1)^2 \leq (s\,a) \tag{8}$$

Proof Sketch

First show the "$\longrightarrow$"-direction:

(3) $\longrightarrow$ (7) and (1) $\wedge$ (2) $\longrightarrow$ (8) by simple arithmetic. (6) is shown as follows:

$$\begin{aligned}
((s\,i+1)+1)^2 &= (s\,i+1)^2+2*(s\,i+1)+1\\
&\stackrel{(2)}{=} (s\,sum)+2(s\,i)+1+2\\
&\stackrel{(3)}{=} (s\,sum)+(s\,tm)+2
\end{aligned}$$

Proof Sketch (Cont.)

Now show the "$\longleftarrow$"-direction:

(7) $\longrightarrow$ (3) and (8) $\longrightarrow$ (4) by simple arithmetic. (2) is shown as follows:

$$\begin{array}{rcl}
(s\,i+1)^2 & = & ((s\,i+1)+1)^2 - 2*(s\,i+1)-1 \\
& \stackrel{(6)}{=} & (s\,sum)+(s\,tm)+2-2*(s\,i+1)-1 \\
& \stackrel{(7)}{=} & (s\,sum)+2*(s\,i+1)+1 \\
& & \qquad -2*(s\,i+1)-1 \\
& = & s\,sum
\end{array}$$

Finally, (2) $\wedge$ (8) $\longrightarrow$ (1). So Inv is indeed an invariant!

The WHILE Loop: Remarks

We have shown

("enter condition" $\wedge$ "invar. at entry")$\leftrightarrow$"invar. at exit"

One would definitely expect $\longrightarrow$, but $\longleftarrow$ is remarkable!

We can show this because our invariant is so strong: for showing $\longrightarrow$, the weaker invariant (2) $\wedge$ (3), i.e.

$$"(i+1)^2 = sum \ \wedge \ tm = (2 * i) + 1$$

would do (check it!).

But the extra condition $i^2 \leq a$ is needed for showing $Post$, which states what the program actually computes.

Taking Care of $Post$

We have shown $\boxed{\mathcal{I}_1}$ and $\{Inv\}\boxed{WH\ldots}\{ExC\}$. Now continue with $\boxed{\mathcal{I}_2}$.

Does $Post\ s$ follow from $Inv\ s \wedge \neg s\ sum \leq s\ a$?

Yes!

$(s\ i)^2 \leq (s\ a)$	follows from (4)
$(s\ a) < (s\ i + 1)^2$	follows from $\neg s\ sum \leq (s\ a)$ and (2).

The Final Missing Part

$\boxed{\mathcal{I}_3}$ remains to be shown, i.e.

$$\forall s. PW\ s \longrightarrow Inv\ s$$

or, expanding the solutions for PW and Inv

$$\begin{aligned}\forall s.\quad & s\, i = 0 \wedge s\, sum = 1 \wedge s\, tm = 1 \longrightarrow \\ & (s\, i + 1)^2 = s\, sum \wedge \\ & s\, tm = (2 * (s\, i)) + 1 \wedge \\ & (s\, i)^2 \leq (s\, a)\end{aligned}$$

This is easy to check.

An Alternative for Tackling the Loop Part

Recall that our loop invariant was "too strong". An alternative:

$$
\dfrac{\begin{array}{l}\forall s.(Inv\ s \wedge \\ s\ sum \leq s\ a) \longrightarrow \\ Inv'\ s\end{array} \qquad \dfrac{\begin{array}{c}\{Inv'\} i :== \lambda s. s\ i + 1 \{P'\} \\ \{P'\} tm :== \lambda s. s\ tm + 2 \{P''\} \\ \{P''\} sum :== \lambda s. s\ tm + s\ sum \{Inv\}\end{array}}{\{Inv'\} \boxed{\text{"body"}} \{Inv\}}\ semi^2}{\dfrac{\{\lambda s. Inv\ s \wedge s\ sum \leq s\ a\} \boxed{\text{"body"}} \{Inv\}}{\{Inv\} \boxed{WH \ldots} \{ExC\}}\ While}\ conseq
$$

Alternative (Cont.)

Applying ass as before gives

$$
\begin{aligned}
Inv' = \lambda s. Inv(s[i ::= s\, i + 1][tm ::= s\, tm + 2]\\
[sum ::= s\, tm + 2 + s\, sum])
\end{aligned}
$$

We are left with the proof obligation

$$
\begin{aligned}
&\forall s. (Inv\ s \wedge s\ sum \leq s\ a) \rightarrow\\
&Inv(s[i ::= s\, i + 1][tm ::= s\, tm + 2]\\
&\qquad [sum ::= s\, tm + 2 + s\, sum])
\end{aligned}
$$

Automating Hoare Proofs

In the example, we have verified a program computing the square root.

But this was tedious, and parts of the task can be automated.

Weakest Preconditions

Observation: the Hoare relation is deterministic to a certain extent.

Idea: we use this fact for the generation of weakest preconditions.

Weakest preconditions are:

constdefs wp :: com $\Rightarrow$ assn $\Rightarrow$ assn
"wp c Q $\equiv$(λs. $\forall$t. (s,t) $\in$ C(c) $\longrightarrow$ Q t)"

So $wp\ c\ Q$ returns the set of states containing all states s such that if t is reached from s via c, then the post-condition Q holds for t. Computable? Not obvious.

Equivalence Proofs

Main results of the wp-generator are:

wp_SKIP: $\quad$ wp SKIP Q = Q

wp_Ass: $\quad$ wp(x :== a) Q = (λs. Q (s[x::=a s]))

wp_Semi: $\quad$ wp(c; d) Q = wp c (wp d Q)

wp_If: $\quad$ wp (IF b THEN c ELSE d) Q =
$\quad$ (λs. (b s $\longrightarrow$ wp c Q s) $\wedge$ ($\neg$b s $\longrightarrow$ wp d Q s))

wp_While_True: $\quad$ b s $\Longrightarrow$ wp (WHILE b DO c) Q s =
$\quad$ wp (c; WHILE b DO c) Q s

wp_While_False: $\quad$ $\neg$b s $\Longrightarrow$ wp (WHILE b DO c) Q s = Q s

wp_While_if: $\quad$ wp (WHILE b DO c) Q s =
$\quad$ (if b s then wp(c; WHILE b DO c) Q s else Q s)

Computing Weakest Preconditions

Except for termination problem due to While, weakest precondition wp can be computed.

This fact can be used for further proof support by verification condition generation.

Idea: for all statements, the exact wp is computed, except for the creative step at While, where the assertion INV provided by the user for the invariant is taken. An additional function vc ("verification condition") establishes necessary conditions that INV is indeed an invariant.

Annotated IMP Programs

We enrich the syntax by loop-invariants:

```
datatype acom =
    Askip
  | Aass loc aexp
  | Asemi acom acom
  | Aif bexp acom acom
  | Awhile bexp assn acom
```

Computing a "Approximative" Weakest Precondition

We define a function that computes an "approximative" wp:

primrec

awp Askip Q = Q

awp (Aass x a) Q = (λs. Q(s[x::=a s]))

awp (Asemi c d) Q = awp c (awp d Q)

awp (Aif b c d) Q = (λs. (b s $\longrightarrow$awp c Q s) $\wedge$($\neg$b s$\longrightarrow$awp d Q s))

awp (Awhile b Inv c) Q = Inv

Note that awp is not necessarily a wp; this depends if Inv is indeed an invariant.

Verification Condition Generation

Inv is an Invariant if verification conditionvc holds:

primrec

$$
\begin{aligned}
&\mathsf{vc}\ \mathsf{Askip}\ Q = (\lambda s.\ \mathsf{True})\\
&\mathsf{vc}\ (\mathsf{Aass}\ x\ a)\ Q = (\lambda s.\ \mathsf{True})\\
&\mathsf{vc}\ (\mathsf{Asemi}\ c\ d)\ Q = (\lambda s.\ \mathsf{vc}\ c\ (\mathsf{awp}\ d\ Q)\ s \wedge \mathsf{vc}\ d\ Q\ s)\\
&\mathsf{vc}\ (\mathsf{Aif}\ \ b\ c\ d)\ Q = (\lambda s.\ \mathsf{vc}\ c\ Q\ s \wedge \mathsf{vc}\ d\ Q\ s)\\
&\mathsf{vc}\ (\mathsf{Awhile}\ b\ \mathsf{Inv}\ \ c)\ Q = (\lambda s.\ (\mathsf{Inv}\ \ s \wedge \neg b\ s \longrightarrow Q\ s) \wedge\\
&\qquad\qquad (\mathsf{Inv}\ \ s \wedge b\ s \longrightarrow \mathsf{awp}\ c\ \mathsf{Inv}\ \ s) \wedge\\
&\qquad\qquad \mathsf{vc}\ c\ \ \mathsf{Inv}\ \ s)
\end{aligned}
$$

Results on vc (1)

The following facts on vc and awp makes this concept powerful:

Theorem 6 (Soundness (`vc_sound`)):

$(\forall s.\ \text{vc ac Q s}) \Longrightarrow\ \vdash \{\text{awp ac Q}\}\ \text{astrip ac}\ \{Q\}$

vc generated from the annotated program holds, then there exists a Hoare-proof for the program without annotations showing that the postcondition follows from its weakest (annotated) precondition.

Results on vc (2)

Moreover, we have:

Theorem 7 (Completeness (`vc_complete`)):

$$
\begin{aligned}
\vert\!- \{P\}\ c\ \{Q\} \Longrightarrow \exists\, ac.\ \ & \text{astrip}\ \ ac = c\ \wedge \\
& (\forall s.\ \text{vc}\ \ ac\ \ Q\ \ s)\ \wedge \\
& (\forall s.\ P\ \ s \longrightarrow \text{awp}\ ac\ Q\ s)
\end{aligned}
$$

If a Hoare-proof exists, there must exist an annotated program, for which vc generates a true formula and whose precondition P implies the its weakest precondition.

Quintessence: vc abstracts Hoare-proofs away from (imperative) program verification. . .

Summary

- IMP closely follows the standard textbook [Win96].
- Isabelle/HOL is a powerful framework for embedding imperative languages.
- Isabelle/HOL is also a framework for state-of-the-art languages like JAVA.
- Even verification condition generators can be proven sound and complete within HOL.

More Detailed Explanations

Equivalence Proofs

Summarizing, we have the following equivalence results:

- natural vs. transition semantics
- denotational vs. natural semantics

Locations

We realize program variables via pointers (locations). The type of pointers is an abstract datatype.

Defining of values by nat is just a simplification.

A state is a function taking a location to a value, i.e. intuitively, each program variable corresponds to a location, each access to a program variable is an application of state to the location of this variable.

The Intuition of Natural Semantics

The idea of the natural semantics is that a program relates two states, the "input state" and the "output state", provided that it terminates.

This is similar to denotational at first sight, but the treatment of "recursive" constructs such as the `WHILE` is different: denotational semantics reduces these to fixpoint operators, natural semantics to the (meta)-question, if a derivation for the transition is possible or not.

The Intuition of Transition Semantics

Unlike the natural semantics, the transition semantics records the single steps of the computation. A configuration is a pair consisting of a program and a state, and one step reaches a new program and a new state.

The intuition behind the program-component is "the program to be executed"; it is manipulated in a stack-like manner during the evaluation of the rules.

An in-depth investigation of the rules reveals, that there are only finitely many different programs-components in all configuration traces; these correspond to "positions" in a program. (Note that assignments have a "position before" and a "position after" execution). Thus, this component can be seen as a program counter (PC).

Understanding Gamma

We discuss the approximation relation Gamma in more detail:

"Gamma b cd $\equiv$(λphi.{(s,t). (s,t) : (phi O cd) $\wedge$ b(s)} $\cup$
{(s,t). s=t $\wedge \neg$b(s)})"

Note that in the definition of `WHILE`, the second argument cd to Gamma is used for the meaning of the body of the loop.

Thus, the underlying principle is similar to the one used when defining the transitive closure by lfp.

Let

$$S_0 = \{(s, t) \mid s = t \wedge \neg b(s)\}$$

be the initial apprimation (the subset of the identity relation, for which the condition b is false.

Then we can iterate from S_0 via composition cd^n arbitrarily many transitions through the body of the loop. The lfp represents the limit of this approximation process.

Note that for b $= \lambda$x. true S_0 is empty, therefore Gamma b cd is empty and, consequently, the denotational semantics C will yield the empty relation in such cases.

A Table of Values

a is not modified anywhere. Therefore a can be seen as input of the program.

i counts the number of times the loop is entered, i.e. the final value of i is the number of times the loop was entered. This number depends on a. The following table shows that final values of i, tm and sum depending on the value of a:

	i	tm	sum
$0 \leq a < 1$	0	1	1
$1 \leq a < 4$	1	3	4
$4 \leq a < 9$	2	5	9
$9 \leq a < 16$	3	7	16
$16 \leq a < 25$	4	9	25
$25 \leq a < 36$	5	11	36
$36 \leq a < 49$	6	13	49

sum takes the values of all squares successively, computed by the famous binomial formula:

$$(i + 1)^2 = i^2 + 2i + 1$$

Since tm takes the value $2i + 1$ for all i successively, it follows that

$sum + tm$ always gives the next value of sum.

$(s\,i)$, $(s\,a)$ etc.

Informally we talk about variables i, x etc. and say "x has value 5", for example. But formally, program variables are realized via locations, and when accessing a program variable, we get expressions of the form $s\,x$. That is, $s\,x$ is the value of variable x.

Nondeterminacy in the Hoare Calculus

The $conseq$ rule can always be applied. For all other commands, the choice for a hoare-rule is uniquely determined.

Relative Completeness

After Gödel, logicians tend to be nervous whenever a logic is claimed to be complete, in particular if arithmetic is involved as is the case for IMP.

Relative completeness means that for any valid Hoare triple in validity relation there will be a proof in hoare logic.

However, will we be able to derive in HOL that any semantically valid hoare triple is valid in the sense of validity relation ? What if we write in a precondition something like "if Goldbach's conjecture holds" or, worse, "if we can solve an arbitrary diophantine equation"?

The answer is no since HOL itself is incomplete wrt. standard models.

Program "Fragment"

This is the entire program, namely:

$$
\begin{array}{ll}
\mathsf{tm} & :== \lambda \mathsf{s}.\ 1; \\
\mathsf{sum} & :== \lambda \mathsf{s}.\ 1; \\
\mathsf{i} & :== \lambda \mathsf{s}.\ 0; \\
\multicolumn{2}{l}{\ \mathtt{WHILE}\ \ \lambda \mathsf{s}.\ \ \mathsf{s\ sum} <= \mathsf{s\ a}\ \mathtt{DO}} \\
\multicolumn{2}{l}{\quad (\mathsf{i} \qquad :== \lambda \mathsf{s}.\ \mathsf{s\ i} + 1;} \\
\multicolumn{2}{l}{\quad\ \ \mathsf{tm} \quad\ :== \lambda \mathsf{s}.\ \mathsf{s\ tm} + 2;} \\
\multicolumn{2}{l}{\quad\ \ \mathsf{sum} \ \ :== \lambda \mathsf{s}.\ \mathsf{s\ tm} + \mathsf{s\ sum})}
\end{array}
$$

(return to main proof tree)

Program Fragment

This is the program fragment starting from sum :==, namely:

```
sum  :== λs. 1;
i    :== λs. 0;
 WHILE  λs.  s sum <= s a DO
  (i       :== λs. s i + 1;
   tm      :== λs. s tm + 2;
   sum     :== λs. s tm + s sum)
```

(return to main proof tree)

Program Fragment

This is the program fragment starting from i :==, namely:

i :== λx. 0;
WHILE λs. s sum <= s a DO
(i :== λs. s i + 1;
tm :== λs. s tm + 2;
sum :== λs. s tm + s sum)

(return to main proof tree)

Invariant

$$Inv \;\equiv\; ''\lambda s.(s\,i + 1)^2 = s\,sum \;\wedge\; s\,tm = (2 * s\,i) + 1 \;\wedge\; s\,i^2 \leq a''$$

Program Fragment

This is the program fragment starting from $WHILE$, namely:

$$
\begin{array}{l}
\texttt{WHILE}\ \ \lambda \mathsf{s.}\ \ \mathsf{s\ sum} <= \mathsf{s\ a}\ \texttt{DO} \\
\quad (\mathsf{i} \qquad :== \lambda \mathsf{s.\ s\ i} + 1; \\
\quad\ \ \mathsf{tm} \quad :== \lambda \mathsf{s.\ s\ tm} + 2; \\
\quad\ \ \mathsf{sum}\ \ :== \lambda \mathsf{s.\ s\ tm} + \mathsf{s\ sum})
\end{array}
$$

(return to main proof tree)

Program Fragment

This is the program fragment consisting of the loop body, namely:

$$
\begin{array}{ll}
\mathsf{i} & :== \lambda \mathsf{s}.\ \mathsf{s}\ \mathsf{i}\ + 1; \\
\mathsf{tm} & :== \lambda \mathsf{s}.\ \mathsf{s}\ \mathsf{tm} + 2; \\
\mathsf{sum} & :== \lambda \mathsf{s}.\ \mathsf{s}\ \mathsf{tm} + \mathsf{s}\ \mathsf{sum}
\end{array}
$$

(return to main proof tree)

A Missing Part

$\boxed{\mathcal{I}_1}$ is the formula

$$\forall s. Pre\ s \rightarrow PW(s[''i, sum, tm''])$$

where Pre is defined above and PW is a metavariable ("precondition of $WHILE$").

(return to main proof tree)

A Missing Part

$\boxed{\mathcal{I}_2}$ is the formula

$$\forall s. ExC\ s \rightarrow Post\ s,$$

i.e.

$$\forall s. Inv\ s \wedge \neg sum\ s \leq s\ a \rightarrow Post\ s,$$

where $Post$ is defined above and Inv is a metavariable ("loop invariant").

(return to main proof tree)

A Missing Part

$\boxed{\mathcal{A}_1}$ is the proof tree

$$\frac{}{\{\lambda s.PW(s[''i, sum, tm''])\}tm :== \lambda x.1\{\lambda s.PW(s[''i, sum''])\}}\ ass$$

where PW is a metavariable ("precondition of $WHILE$").

(return to main proof tree)

A Missing Part

$\boxed{\mathcal{A}_2}$ is the proof tree

$$\frac{}{\{\lambda s.PW(s[i ::= 0][sum ::= 1])\}sum :== \lambda x.1\{\lambda s.PW(s[i ::= 0])\}}\ ass$$

where PW is a metavariable ("precondition of $WHILE$").

(return to main proof tree)

A Missing Part

$\boxed{\mathcal{A}_3}$ is the proof tree

$$\frac{}{\{\lambda s.PW(s[i ::= 0])\}i :== \lambda x.0\{PW\}}\,ass$$

where PW is a metavariable ("precondition of $WHILE$").

(return to main proof tree)

A Missing Part

$\boxed{\mathcal{I}_3}$ is the formula

$$\forall s. PW\ s \rightarrow Inv\ s$$

where PW is a metavariable ("precondition of $WHILE$") and Inv is a metavariable ("loop invariant").

(return to main proof tree)

A Missing Part

$\boxed{\mathcal{I}_4}$ is the formula

$$\forall s. PW\ s \rightarrow PW\ s$$

which is of course trivial to prove.

(return to main proof tree)

An Abbreviation for an Updated State

We use $s[\text{"}i, sum, tm\text{"}]$ as abbreviation for

$$s[i ::= 0][sum ::= 1][tm ::= 1]$$

(return to main proof tree)

An Abbreviation for an Updated State

We use $s[\text{"}i, sum\text{"}]$ as abbreviation for

$$s[i ::= 0][sum ::= 1]$$

(return to main proof tree)

An Abbreviation for an Updated State

We use $s[''i'']$ as abbreviation for

$$s[i ::= 0]$$

(return to main proof tree)

What must Inv Be?

Recall that we had to prove the three formulas

$$\{\lambda s. Inv\ s \wedge s\ sum \leq s\ a\} i :== \lambda s. s\ i + 1 \{P'\}$$
$$\{P'\} tm :== \lambda s. s\ tm + 2 \{P''\}$$
$$\{P''\} sum :== \lambda s. s\ tm + s\ sum \{Inv\}$$

all by ass. Dealing with the second and third formula using ass, we found that

$$P' = \lambda s'. Inv(s'[tm ::= s'\ tm + 2][sum ::= s'\ tm + 2 + s'\ sum]).$$

Therefore, to show

$$\{\lambda s. Inv\ s \wedge s\ sum \leq s\ a\} i :== \lambda s. s\ i + 1 \{P'\}$$

as well, Inv must have such a form that the formula becomes an instance of ass.

References

[CDTK86] J. Cl ´ement, J. Despeyroux, T.Despeyroux, and G. Kahn. A simple applicative language: mini-ml. In *Proc. of the 1986 ACM Conference on Lisp and Functional Programming*. ACM, 1986.

[Nip02] Tobias Nipkow. Hoare logics in Isabelle/HOL. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability*, pages 341–367. Kluwer, 2002.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Denmark, 1981.

[Win96] Glynn Winskel. *The Formal Semantics of Programming Languages – An Introduction*. MIT Press, 1996. 3rd ed.