Winskel is (almost) Right
Towards a Mechanized Semantics Textbook

Tobias Nipkow *

Abstract. We present a formalization of the first 100 pages of Winskel's *The Formal Semantics of Programming Languages* in the theorem prover Isabelle/HOL: 2 operational, 2 denotational, 1 axiomatic semantics, a verification condition generator, and the necessary soundness, completeness and equivalence proofs, all for a simple imperative language.

Are theorem provers capable of formalizing significant portions of mathematics or computer science? If we talk about leading edge research, the answer is at best "with difficulty". However, if we talk about foundations and textbooks, the answer must be a guarded "yes". The first and best known example is the translation of Landau's "Grundlagen" into Automath [12]. Our paper summarizes the formalization of the first 100 pages of a textbook on programming language semantics [13]. It deals with various semantics for a simple imperative language and proves their equivalence. The main purpose of the whole development is

– To lay the foundation for a unified treatment of the many facets of a programming language ranging from its denotational semantics to the soundness and completeness of a verification condition generator. This formalization allows performing proofs both about the language, e.g. compiler verification, and about programs in the language in the same system.
– To demonstrate not just the mere possibility of such an undertaking, but to show that the result is both readable and fairly close to the original text.

It is a tribute to Winskel's thoroughness that we only found one serious mistake in his proofs, which is the source of the "almost" in the title. The mistake occurs in the completeness proof for Hoare logic (Section 5.2) and is easily fixed.

The idea of embedding the semantics of a programming language in a theorem prover goes back at least to Gordon [3]. His paper has spawned many further language embeddings, ours included. However, we are not aware of any previous unified treatment of all the different semantic formalisms covered by our paper.

After a short introduction to HOL, the rest of the paper is structured like Winskel's book: operational, denotational and axiomatic semantics are presented together with their equivalence proofs. In addition we prove the soundness and completeness of a verification condition generator (Section 5.3) and treat the thorny issue of partial functions in a logic of total functions (Section 6).

The complete formalization (including proofs) is available on the web via `http://www4.informatik.tu-muenchen.de/~nipkow/isabelle/HOL/IMP/`.

* Institut für Informatik, TU München, 80290 München, Germany.
`http://www4.informatik.tu-muenchen.de/~nipkow/`
Research supported by DFG Schwerpunktprogramm *Deduktion*.

1 Isabelle/HOL

Isabelle/HOL is the instantiation of the generic interactive theorem prover Isabelle [8] with Church's formulation of Higher Order Logic and is very close to Gordon's HOL system [4]. In this paper HOL is short for Isabelle/HOL.

Below you find a short introduction to HOL's surface syntax (as rendered by Regensburger's LaTeX-converter):

Formulae The syntax is standard, except that there are two implications ($\longrightarrow$ and $\Longrightarrow$) and two equalities ($=$ and $\equiv$) which stem from the object and meta-logic, respectively. The distinction can be ignored while reading this paper. The notation $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$ is short for the nested implication $A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow A$.

Types follow the syntax for ML-types, except that the function arrow is $\Rightarrow$.

Theories introduce constants with the keyword **consts**, non-recursive definitions with **defs**, and primitive recursive definitions with **primrec**. Further constructs are explained as we encounter them.

Although we do not present any of the proofs, we usually indicate their complexity. If we state that some proof is automatic, it means that it was either solved by rewriting or by the "classical reasoner", `fast_tac` in Isabelle parlance [9]. The latter provides a reasonable degree of automation for predicate calculus proofs. Note, however, that its success depends on the right selection of lemmas supplied as parameters.

2 IMP

IMP is a simple imperative programming language with WHILE-loops. The syntax for **commands** (aka statements) is

$$c \ ::= \ \mathsf{SKIP} \ \mid \ X := a \ \mid \ c; c \ \mid \ \mathsf{IF}\ b\ \mathsf{THEN}\ c\ \mathsf{ELSE}\ c \ \mid \ \mathsf{WHILE}\ b\ \mathsf{DO}\ c$$

where X is a **location** (aka variable), a an **arithmetic expression** and b a **boolean expression**.

Datatypes in HOL resemble those in functional programming languages and allow a direct representation of the abstract syntax of commands:

```
datatype com = SKIP
             | ":=" loc aexp         (infixl 90)
             | ";"  com com          (infixl 90)
             | Cond bexp com com     ("IF _ THEN _ ELSE _" 100)
             | While bexp com        ("WHILE _ DO _" 100)
```

The annotations define the concrete syntax.

Winskel also treats syntax and semantics of arithmetic and boolean expressions, which we followed in an earlier formalization of IMP [6]. Because expressions add nothing new, we have taken a semantic view, i.e. we have identified expressions with their semantics. The central semantic concept is that of a **state**, i.e. a mapping from locations to **values**. We formalize both locations loc and values val as unspecified types and define state, aexp and bexp as function spaces:

```
types state = loc ⇒ val
      aexp  = state ⇒ val
      bexp  = state ⇒ bool
```

Alternatively, we could have made loc and val explicit parameters of all types, which would have cluttered up the types considerably.

Bypassing the syntax of expressions in favour of semantics means that concrete expressions look a bit unusual. For example, $X := X + 1$ becomes $X := (\lambda s.s(X) + 1)$. It is routine to modify the parser and pretty printer to translate between the two forms automatically [3]. We ignore these syntactic issues and focus on the semantic side of things.

3 Operational Semantics

There are two standard forms of operational semantics which are often called "natural" and "transition" semantics. Winskel concentrates on natural semantics but connects it to transition semantics in an exercise.

3.1 Natural Semantics

Natural semantics expresses the evaluation of commands as a relation between a command, an initial state and a final state. In HOL we declare a constant evalc as a set of such triples

```
consts evalc :: (com ∗ state ∗ state)set
```

and add some syntactic sugar for better readability:

```
translations <c,s> -c-> t ≡ (c,s,t) ∈ evalc
```

This means we read and write $<c,s> \stackrel{c}{\to} t$ instead of (c,s,t) ∈ evalc. The relation evalc is defined inductively by a set of inference rules, i.e. implications:

```
inductive evalc
  <SKIP,s> -c-> s

  <x := a,s> -c-> s[a(s)/x]

  ⟦ <c1,s> -c-> s1; <c2,s1> -c-> s2 ⟧ ⟹ <c1;c2, s> -c-> s2

  ⟦   b s; <c1,s> -c-> t ⟧ ⟹ <IF b THEN c1 ELSE c2, s> -c-> t
  ⟦ ¬ b s; <c2,s> -c-> t ⟧ ⟹ <IF b THEN c1 ELSE c2, s> -c-> t

  ¬ b s ⟹ <WHILE b DO c, s> -c-> s
  ⟦ b s; <c,s> -c-> s1; <WHILE b DO c, s1> -c-> s2 ⟧ ⟹ <WHILE b DO c, s> -c-> s2
```

The assignment command is defined in terms of an auxiliary function on states:

```
consts assign :: state ⇒ val ⇒ loc ⇒ state   ("_[_/_]")
defs   s[m/x] ≡ (λy. if y=x then m else s y)
```

The keyword **inductive** means that evalc is defined as the least relation closed under the given rules. HOL automatically derives a corresponding induction principle, called **rule induction** in [13], which will be our major weapon in the proofs to come.

3.2 Transition Semantics

An alternative semantics is the **transition semantics** which is a relation between **configurations**, i.e. pairs of commands and states. A configuration (c, s) represents a computation in state s which is about to execute c. Each transition is regarded as one step in the computation. This semantics is particularly appropriate for concurrent languages where different executions have to be interleaved.

Winskel outlines the transition semantics for IMP and leaves the details, in particular the equivalence proof of natural and transition semantics, as exercises. This section provides the details, uncovers a minor slip in one of Winskel's hints, and presents an alternative equivalence proof.

Winskel distinguishes two kinds of transitions: $(c, s) \rightarrow_1 (c', s')$ and $(c, s) \rightarrow_1 s'$, where the latter indicates the termination of the computation. To simplify matters we have abolished the second kind of transition and consider $(\mathsf{SKIP}, \mathsf{s})$ a terminal configuration. Hence we need only a single relation:

consts evalc1 :: ((com∗state) ∗ (com∗state))set

Syntactic sugar is introduced in the customary manner:

translations $\mathsf{cs} \xrightarrow{1} \mathsf{cs'} \;\equiv\; (\mathsf{cs},\mathsf{cs'}) \in \mathsf{evalc1}$

The inductive definition of evalc1 is straightforward:

inductive evalc1

$$(\mathsf{x} := \mathsf{a},\mathsf{s}) \xrightarrow{1} (\mathsf{SKIP},\mathsf{s}[\mathsf{a}(\mathsf{s})/\mathsf{x}])$$

$$(\mathsf{SKIP};\mathsf{c},\mathsf{s}) \xrightarrow{1} (\mathsf{c},\mathsf{s})$$
$$(\mathsf{c0},\mathsf{s}) \xrightarrow{1} (\mathsf{c1},\mathsf{t}) \implies (\mathsf{c0};\mathsf{c2},\mathsf{s}) \xrightarrow{1} (\mathsf{c1};\mathsf{c2},\mathsf{t})$$

$$\mathsf{b\ s} \implies (\mathsf{IF\ b\ THEN\ c1\ ELSE\ c2},\mathsf{s}) \xrightarrow{1} (\mathsf{c1},\mathsf{s})$$
$$\neg\ \mathsf{b\ s} \implies (\mathsf{IF\ b\ THEN\ c1\ ELSE\ c2},\mathsf{s}) \xrightarrow{1} (\mathsf{c2},\mathsf{s})$$

$$\mathsf{b\ s} \implies (\mathsf{WHILE\ b\ DO\ c},\mathsf{s}) \xrightarrow{1} (\mathsf{c};\mathsf{WHILE\ b\ DO\ c},\mathsf{s})$$
$$\neg\ \mathsf{b\ s} \implies (\mathsf{WHILE\ b\ DO\ c},\mathsf{s}) \xrightarrow{1} (\mathsf{SKIP},\mathsf{s})$$

The desired equivalence theorem is

$$(<\mathsf{c},\mathsf{s}> \xrightarrow{c} \mathsf{t}) \;=\; ((\mathsf{c},\mathsf{s}) \xrightarrow{*} (\mathsf{SKIP},\mathsf{t})) \tag{1}$$

where $\xrightarrow{*}$ is the transitive and reflexive closure of $\xrightarrow{1}$. The proof also employs $\xrightarrow{\mathsf{n}}$, the n-fold iteration of $\xrightarrow{1}$. Both arrows are syntactic sugar for the postfix operators $^{\mathsf{n}}$ and * which are part of HOL's theory of relations:

translations $\mathsf{cs} \xrightarrow{\mathsf{n}} \mathsf{cs'} \equiv (\mathsf{cs},\mathsf{cs'}) \in \mathsf{evalc1}^{\mathsf{n}}$
$\mathsf{cs} \xrightarrow{*} \mathsf{cs'} \equiv (\mathsf{cs},\mathsf{cs'}) \in \mathsf{evalc1}^{*}$

The $\Longrightarrow$-direction of (1) is proved by rule induction on $<\mathsf{c},\mathsf{s}> \xrightarrow{c} \mathsf{t}$ and uses the following lemma

$$[\![(\mathsf{c1},\mathsf{s1}) \xrightarrow{\mathsf{n}} (\mathsf{SKIP},\mathsf{s2}); (\mathsf{c2},\mathsf{s2}) \xrightarrow{*} (\mathsf{SKIP},\mathsf{s3})]\!] \Longrightarrow (\mathsf{c1};\mathsf{c2},\mathsf{s1}) \xrightarrow{*} (\mathsf{SKIP},\mathsf{s3})$$

which is proved by induction on n.

The $\Longleftarrow$-direction of (1) is proved by induction on c and a nested induction on the length of $(\mathsf{c},\mathsf{s}) \xrightarrow{*} (\mathsf{SKIP},\mathsf{t})$ in the WHILE-case. The nested induction requires the following lemma:

$$(\mathsf{c1};\mathsf{c2},\mathsf{s1}) \xrightarrow{\mathsf{n}} (\mathsf{SKIP},\mathsf{s3}) \Longrightarrow \exists \mathsf{s2}\ \mathsf{m}.\ (\mathsf{c1},\mathsf{s1}) \xrightarrow{*} (\mathsf{SKIP},\mathsf{s2}) \wedge (\mathsf{c2},\mathsf{s2}) \xrightarrow{\mathsf{m}} (\mathsf{SKIP},\mathsf{s3}) \wedge \mathsf{m} \leq \mathsf{n}$$

This lemma is stronger than the one suggested by Winskel, where $\xrightarrow{\mathsf{n}}$ and $\xrightarrow{\mathsf{m}}$ are replaced by $\xrightarrow{*}$ and $\mathsf{m} \leq \mathsf{n}$ disappears. The reason for the stronger lemma is the induction on the length in the WHILE-case: unless we have $\mathsf{m} \leq \mathsf{n}$, the induction hypothesis is not applicable.

There is an alternative proof of (1) which does not drag in natural numbers via $\xrightarrow{\mathsf{n}}$ at all. For a start, we can prove the generalized lemma

$$[\![(\mathsf{c1},\mathsf{s1}) \xrightarrow{*} (\mathsf{SKIP},\mathsf{s2}); (\mathsf{c2},\mathsf{s2}) \xrightarrow{*} \mathsf{cs3}]\!] \Longrightarrow (\mathsf{c1};\mathsf{c2},\mathsf{s1}) \xrightarrow{*} \mathsf{cs3}$$

directly by induction on the structure (as opposed to the length) of $(\mathsf{c1},\mathsf{s1}) \xrightarrow{*} (\mathsf{SKIP},\mathsf{s2})$. As above, this yields the $\Longrightarrow$-direction of (1). Note that the generalization of $(\mathsf{SKIP},\mathsf{s3})$ to $\mathsf{cs3}$ is not really essential in this context but improves our understanding of what is going on. The proof of the opposite direction was suggested by Ranan Fraer (personal communication) and is based on similar proofs of his in the Coq system (see [1] for related material). The key lemma

$$[\![(\mathsf{c},\mathsf{s}) \xrightarrow{1} (\mathsf{c'},\mathsf{s'}); <\mathsf{c'},\mathsf{s'}> \xrightarrow{c} \mathsf{t}]\!] \Longrightarrow <\mathsf{c},\mathsf{s}> \xrightarrow{c} \mathsf{t}$$

is proved by rule induction on $(\mathsf{c},\mathsf{s}) \xrightarrow{1} (\mathsf{c'},\mathsf{s'})$. An induction on the structure of $(\mathsf{c},\mathsf{s}) \xrightarrow{*} (\mathsf{c'},\mathsf{s'})$ now yields

$$[\![(\mathsf{c},\ \mathsf{s}) \xrightarrow{*} (\mathsf{c'},\ \mathsf{s'}); <\mathsf{c'},\mathsf{s'}> \xrightarrow{c} \mathsf{t}]\!] \Longrightarrow <\mathsf{c},\mathsf{s}> \xrightarrow{c} \mathsf{t}$$

which directly implies the $\Longleftarrow$-direction of (1).

4 Denotational Semantics

Winskel starts with a low cost version of denotational semantics which is entirely based on sets. It is sometimes called "relational semantics" because the denotation of a command is a relation between initial and final states:

types $\mathsf{com\_den} = (\mathsf{state} * \mathsf{state})\mathsf{set}$

This approach suits us fine because it avoids partial functions, a sticky issue in HOL. In Section 6 we come back to this point. For the time being, we work with relations. The semantic function C is defined by primitive recursion on commands:

consts $\mathsf{C} :: \mathsf{com} \Rightarrow \mathsf{com\_den}$
primrec

$$
\begin{array}{l}
\mathsf{C(SKIP)} \;=\; \mathsf{id} \\
\mathsf{C(x := a)} \;=\; \{(\mathsf{s,t}).\ \mathsf{t = s[a(s)/x]}\} \\
\mathsf{C(c1;c2)} \;=\; \mathsf{C(c2)\ O\ C(c1)} \\
\mathsf{C(IF\ b\ THEN\ c1\ ELSE\ c2)} \;=\; \{(\mathsf{s,t}).\ (\mathsf{s,t}) \in \mathsf{C(c1)} \wedge \mathsf{b(s)}\} \cup \\
\qquad\qquad\qquad\qquad\qquad\qquad\; \{(\mathsf{s,t}).\ (\mathsf{s,t}) \in \mathsf{C(c2)} \wedge \neg\, \mathsf{b(s)}\} \\
\mathsf{C(WHILE\ b\ DO\ c)} \;=\; \mathsf{lfp}(\Gamma\ \mathsf{b}\ (\mathsf{C\ c}))
\end{array}
$$

where Γ is an auxiliary function:

consts $\Gamma :: \mathsf{bexp} \Rightarrow \mathsf{com\_den} \Rightarrow (\mathsf{com\_den} \Rightarrow \mathsf{com\_den})$
defs $\Gamma\ \mathsf{b\ cd} \;\equiv\; (\lambda \mathsf{R}.\{(\mathsf{s,t}).\ (\mathsf{s,t}) \in (\mathsf{R\ O\ cd}) \wedge \mathsf{b(s)}\} \cup \{(\mathsf{s,t}).\ \mathsf{s{=}t} \wedge \neg\, \mathsf{b(s)}\})$

This definition relies heavily on HOL's theory of sets and relations: O and id are identity and composition of relations, and $\mathsf{lfp} :: (\alpha\ \mathsf{set} \Rightarrow \alpha\ \mathsf{set}) \Rightarrow \alpha\ \mathsf{set}$ computes the least fixpoint of a monotone function on sets. The two key theorems about lfp express that the result of lfp is indeed a fixpoint and satisfies an induction principle ($\bigwedge$ is the universal quantifier of Isabelle's meta-logic):

$$\mathsf{mono(f)} \Longrightarrow \mathsf{lfp(f)} = \mathsf{f(lfp(f))} \tag{2}$$

$$[\![\ \mathsf{a} \in \mathsf{lfp(f)};\ \mathsf{mono(f)};\ \bigwedge \mathsf{x}.\ \mathsf{x} \in \mathsf{f(lfp(f)} \cap \{\mathsf{x}.\ \mathsf{P(x)}\}) \Longrightarrow \mathsf{P(x)}\ ]\!] \Longrightarrow \mathsf{P(a)} \tag{3}$$

Monotonicity of Γ, i.e. $\mathsf{mono}(\Gamma\ \mathsf{b\ cd})$, is proved automatically. A simple consequence is the following recursion equation:

$$\mathsf{C(WHILE\ b\ DO\ c)} = \mathsf{C(IF\ b\ THEN\ c; WHILE\ b\ DO\ c\ ELSE\ SKIP)} \tag{4}$$

4.1 Denotational is Natural

The equivalence proof between the natural and the denotational semantics follows Winskel closely and is pretty much automatic:

$\mathsf{<c,s>} \stackrel{c}{\rightarrow} \mathsf{t} \Longrightarrow (\mathsf{s,t}) \in \mathsf{C(c)}$

is proved by rule induction. The opposite direction

$(\mathsf{s,t}) \in \mathsf{C(c)} \Longrightarrow \mathsf{<c,s>} \stackrel{c}{\rightarrow} \mathsf{t}$

is proved by induction on c and an application of the lfp induction principle (3) in the WHILE-case. This is slightly different from Winskel's proof, which uses induction on $\mathbb{N}$ in the WHILE-case because Γ is continuous.

Apart from the application of the induction rules and a few explicit unfoldings of lfp — equation (2) is not a terminating rewrite rule and has to be applied "by hand" — the proofs are automatic. The final equivalence is now trivial:

$$(\mathsf{s,t}) \in \mathsf{C(c)} \;=\; (\mathsf{<c,s>} \stackrel{c}{\rightarrow} \mathsf{t}) \tag{5}$$

5 Axiomatic Semantics

In this section we diverge most significantly from Winskel's treatment: on the one hand we take a short cut by not formalizing the syntax of the assertion language, on the other hand we provide a more satisfactory treatment of verification conditions.

A complete formalization of syntax and semantics of assertions, i.e. first order arithmetic, is a project in its own right. Therefore we have taken the semantic way out:

types assn = state ⇒ bool

This is the same trick we used for arithmetic and boolean expressions.

Validity of Hoare triples is now straightforward:

consts hoare_valid :: assn ⇒ com ⇒ assn ⇒ bool ("⊨ {_} _ {_}")
defs ⊨ {P} c {Q} ≡ ∀s t. (s,t) ∈ C(c) ⟶ P(s) ⟶ Q(t)

Hoare logic is formalized just like operational semantics, i.e. as a relation:

consts hoare :: (assn ∗ com ∗ assn) set
translations ⊢ {P} c {Q} ≡ (P,c,Q) ∈ hoare

The inference rules of the logic constitute an inductive definition:

inductive hoare
⊢ {P} SKIP {P}

⊢ {λs.P(s[a(s)/x])} x := a {P}

⟦ ⊢ {P} c1 {Q}; ⊢ {Q} c2 {R} ⟧ ⟹ ⊢ {P} c1;c2 {R}

⟦ ⊢ {λs. P(s) ∧ b(s)} c1 {Q}; ⊢ {λs. P(s) ∧ ¬ b(s)} c2 {Q} ⟧
⟹ ⊢ {P} IF b THEN c1 ELSE c2 {Q}

⊢ {λs. P(s) ∧ b(s)} c {P} ⟹ ⊢ {P} WHILE b DO c {λs. P(s) ∧ ¬ b(s)}

⟦ ∀s. P'(s) ⟶ P(s); ⊢ {P} c {Q}; ∀s. Q(s) ⟶ Q'(s) ⟧ ⟹ ⊢ {P'} c {Q'}

The last rule is called the **rule of consequence**.

Having identified assertions with their semantics we have to write λs. P(s) ∧ b(s) instead of the usual P ∧ b, which is not well typed. This effect was already discussed in the context of expressions (see the end of Section 2).

5.1 Soundness

The proof of

⊢ {P} c {Q} ⟹ ⊨ {P} c {Q}

is by rule induction on the derivation of ⊢ {P} c {Q}. All cases are automatic except for the WHILE-rule which requires an explicit lfp induction step (3).

5.2 Relative Completeness

The adjective "relative" refers to the fact that it is relative to the completeness of the proof system for the assertion language. In our setting this is already built in, because we do not have a proof system but merely the semantics of the assertion language. Hence we drop "relative" in the sequel.

Although we can prove completeness of our formalization of Hoare logic

$$\models \{\mathsf{P}\}\ \mathsf{c}\ \{\mathsf{Q}\} \implies \vdash \{\mathsf{P}\}\ \mathsf{c}\ \{\mathsf{Q}\} \tag{6}$$

it falls short of the corresponding completeness statement in the literature. The reason is our semantic view of assertions. Although (6) shows that all valid triples can be derived, this does not preclude that the derivation makes use of semantic assertions which cannot be expressed syntactically. Such non-expressible assertions can enter a derivation even if the derived triple itself only contains expressible assertions: in the rule of consequence P and Q occur in the premises but not the conclusion. In fact, our proof follows Winskel's up to the point where he uses the rule of consequence but shows that the chosen P and Q are expressible.

Winskel's proof hinges on the fact that the weakest liberal precondition is expressible. We follow Winskel and drop the adjective "liberal" in the sequel. Semantically, the weakest precondition is trivial:

consts $\mathsf{swp} :: \mathsf{com} \Rightarrow \mathsf{assn} \Rightarrow \mathsf{assn}$
defs $\mathsf{swp}\ \mathsf{c}\ \mathsf{Q}\ \equiv\ (\lambda \mathsf{s}.\ \forall \mathsf{t}.\ (\mathsf{s},\mathsf{t}) \in \mathsf{C}(\mathsf{c}) \longrightarrow \mathsf{Q}(\mathsf{t}))$

The assertion $\mathsf{swp}\ \mathsf{c}\ \mathsf{Q}$ characterizes all initial states which lead to Q, provided c terminates. The name swp was chosen to distinguish it from a second wp-like operator further down.

Winskel calls an assertion language **expressive** if for every command c and assertion Q, there is an assertion P with semantics $\mathsf{swp}\ \mathsf{c}\ \mathsf{Q}$. He shows that a) first order arithmetic is expressive and b) expressiveness implies completeness. In our setting expressiveness is not an issue because we can use swp. It remains to be shown that it leads to completeness.

The key lemma looks trivial

$$\forall \mathsf{Q}.\ \vdash \{\mathsf{swp}\ \mathsf{c}\ \mathsf{Q}\}\ \mathsf{c}\ \{\mathsf{Q}\} \tag{7}$$

but is a bit tricky. The proof is by induction on c. All cases except IF and WHILE are proved automatically. The IF-case needs the rule of consequence, which has to be applied "by hand". Fortunately, Isabelle's logical variables save us from having to supply instantiations as well. The really interesting case is WHILE, because this is the only major slip in any of Winskel's proofs. Let us look at it in some detail. The subgoal is

$$\forall \mathsf{Q}.\ \vdash \{\mathsf{swp}\ \mathsf{c}\ \mathsf{Q}\}\ \mathsf{c}\ \{\mathsf{Q}\} \implies$$
$$\vdash \{\mathsf{swp}\ (\mathsf{WHILE}\ \mathsf{b}\ \mathsf{DO}\ \mathsf{c})\ \mathsf{Q}\}\ \mathsf{WHILE}\ \mathsf{b}\ \mathsf{DO}\ \mathsf{c}\ \{\mathsf{Q}\}$$

Winskel claims (in our notation) that from $\models \{\lambda \mathsf{s}.\ \mathsf{P}(\mathsf{s}) \land \mathsf{b}(\mathsf{s})\}\ \mathsf{c}\ \{\mathsf{P}\}$, where P $\equiv \mathsf{swp}\ (\mathsf{WHILE}\ \mathsf{b}\ \mathsf{DO}\ \mathsf{c})\ \mathsf{Q}$, it follows by induction hypothesis that $\vdash \{\lambda \mathsf{s}.\ \mathsf{P}(\mathsf{s})$

$\wedge$ b(s)} c {P}. At this point he is jumping the gun, as a look at the induction hypothesis shows. It appears that his mind is already fixed on completeness.

Fortunately the proof can be repaired quite easily. Let us first show

$$\mathsf{P(s)} \wedge \mathsf{b(s)} \longrightarrow \mathsf{swp\ c\ P\ s} \tag{8}$$

$$\mathsf{P(s)} \wedge \neg\, \mathsf{b(s)} \longrightarrow \mathsf{Q} \tag{9}$$

$$\{\mathsf{swp\ c\ P}\}\ \mathsf{c}\ \{\mathsf{P}\} \tag{10}$$

where again P $\equiv$ swp (WHILE b DO c) Q. The first two follow essentially from (4) via the lemmas

```
  b(s) ⟹ swp (WHILE b DO c) Q s = swp (c;WHILE b DO c) Q s
¬ b(s) ⟹ swp (WHILE b DO c) Q s = Q s
```

The third one is an instance of the induction hypothesis.

From (8) and (10) we obtain {λs. P(s) $\wedge$ b(s)} c {P} by the rule of consequence. The WHILE-rule turns this into {P} WHILE b DO c {λs. P(s) $\wedge$ $\neg$ b(s)}. Using (9) and the rule of consequence this yields the desired {P} WHILE b DO c {Q}. The Isabelle proof requires some guidance but no instantiations.

Having proved (7), completeness (6) follows easily by the rule of consequence.

5.3 Verification Conditions

Using Hoare logic directly is tedious, and some of that tedium can be automated. The idea is to extract a set of assertions, the **verification conditions**, from the program such that the program is correct if the assertions are, thus bypassing Hoare logic and reducing the verification problem to the assertion language. The automatic extraction of verification conditions requires a program where all loops are annotated with their invariants.

Winskel follows Gordon [2] in his treatment of verification conditions, who inserts rather more annotations than strictly speaking necessary. This complicates the syntax of annotated commands. We go for the minimal amount of annotation, namely loop invariants. Since there is no such thing as inheritance in HOL, we have to define a new type of **annotated commands**

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

and a function for stripping an annotated command of its annotations:

```
consts astrip :: acom ⇒ com
primrec
  astrip Askip = SKIP
  astrip (Aass x a) = (x:=a)
  astrip (Asemi c d) = (astrip c ; astrip d)
  astrip (Aif b c d) = (IF b THEN astrip c ELSE astrip d)
  astrip (Awhile b I c) = (WHILE b DO astrip c)
```

The computation of the verification conditions needs two functions

consts vc,wp :: acom $\Rightarrow$ assn $\Rightarrow$ assn

where vc returns the verification condition and wp the weakest (liberal) precondition. The soundness property we are aiming for is

$$(\forall s.\ \mathsf{vc}\ c\ Q\ s) \Longrightarrow\ \vdash \{\mathsf{wp}\ c\ Q\}\ \mathsf{astrip}\ c\ \{Q\} \tag{11}$$

The reason for dragging in wp is twofold: vc only takes the postcondition (hence we need to know w.r.t. which precondition vc guarantees correctness) and there are no annotations between sequential commands (hence we need wp c2 in order to compute vc (Asemi c1 c2)). This time we cannot take the semantic way out (see swp) because we are really meant to compute an assertion, i.e. some syntactic entity. The following definition of wp is classical, except for the final clause:

primrec
wp Askip Q = Q
wp (Aass x a) Q = (λs.Q(s[a(s)/x]))
wp (Asemi c1 c2) Q = wp c1 (wp c2 Q)
wp (Aif b c1 c2) Q = (λs. (b(s) $\longrightarrow$ wp c1 Q s) $\wedge$ ($\neg$ b(s) $\longrightarrow$ wp c2 Q s))
wp (Awhile b I c) Q = I

In the final clause we use the the invariant I as the weakest precondition. What if the programmer has supplied an invariant which is too strong or not invariant? If I is too strong, wp computes too strong a precondition and the conclusion of (11) is weaker than it could be. If I is not even invariant, the verification conditions computed by vc are not valid either, thus rendering (11) trivial because its premise cannot be discharged.

The verification condition generator vc is defined by primitive recursion:

primrec
vc Askip Q = (λs.True)
vc (Aass x a) Q = (λs.True)
vc (Asemi c d) Q = (λs. vc c (wp d Q) s $\wedge$ vc d Q s)
vc (Aif b c d) Q = (λs. vc c Q s $\wedge$ vc d Q s)
vc (Awhile b I c) Q = (λs. (I(s) $\wedge$ b(s) $\longrightarrow$ wp c I s) $\wedge$ (I(s) $\wedge$ $\neg$ b(s) $\longrightarrow$ Q(s)) $\wedge$
vc c I s)

The final clause again deserves some comments: I(s) $\wedge$ b(s) $\longrightarrow$ wp c I s guarantees that I is an invariant, I(s) $\wedge$ $\neg$ b(s) $\longrightarrow$ Q(s) guarantees that upon exit the postcondition Q holds, and vc c I takes care of the verification conditions in the loop body.

The proof of soundness (11) is by induction on c. It needs a little guidance in the Aif and Awhile cases but is automatic otherwise.

Completeness requires two monotonicity properties:

$\forall$P Q. ($\forall$s. P s $\longrightarrow$ Q s) $\longrightarrow$ ($\forall$s. wp c P s $\longrightarrow$ wp c Q s)
$\forall$P Q. ($\forall$s. P s $\longrightarrow$ Q s) $\longrightarrow$ ($\forall$s. vc c P s $\longrightarrow$ vc c Q s)

Both proofs are by induction on c and automatic except for the Awhile case. The proof of the completeness theorem

$\vdash \{P\}\ c\ \{Q\} \Longrightarrow (\exists ac.\ \mathsf{astrip}\ ac = c \wedge (\forall s.\ \mathsf{vc}\ ac\ Q\ s) \wedge (\forall s.\ P\ s \longrightarrow \mathsf{wp}\ ac\ Q\ s))$

is a straightforward rule induction on $\vdash \{P\}\ c\ \{Q\}$. This time we have to provide the witness ac by hand in each case.

If one is interested in a more efficient computation of verification conditions, vc and wp can be combined into a single function returning a pair of the weakest precondition and the verification condition. This function only traverses the command once, in contrast to the above solution. For lack of space we do not present the details. The structure of the combined function can also be found in the work of Homeier and Martin [5], who prove soundness but not completeness of their verification condition generator.

6 Denotational Semantics in HOLCF

Although Winskel introduces partial functions as relations and uses the complete powerset lattice to introduce the basics of fixpoint theory (see Section 4 above), he goes on to develop domain theory based on complete partial orders (cpos). This section recasts IMP's denotational semantics in a cpo framework. Winskel just sketches the necessary steps because on the surface not much changes. Logically, however, it implies leaving the safe haven of HOL's total functions and venturing into the sea of continuity and undefinedness. The purpose of this section is to demonstrate that, given the right infrastructure (HOLCF!), this step need not be painful.

HOLCF [10, 11] is a conservative extension of HOL with the notions of domain theory [7]. In particular it provides

- a class pcpo (pointed cpo) of types which come equipped with a complete partial order $\sqsubseteq$ and a least element $\perp$.
- a space $\rightarrow$ of continuous functions between pcpos, together with its own abstraction Λ, infix application ‘, composition oo, and fixpoint operator fix.

We only want to move into HOLCF for those aspects which require domain theory. To embed HOL types into pcpos we use the concept of **lifting**:

datatype $(\alpha)\mathsf{lift} = \mathsf{Def}\ \alpha \mid \mathsf{Undef}$

We turn (α)lift into a pcpo by defining $\perp \equiv \mathsf{Undef}$ and $x \sqsubseteq y \equiv x{=}\mathsf{Undef} \vee x{=}y$. The type (τ)lift is usually written $\tau_\perp$. The functional

consts $\mathsf{lift1} :: (\alpha \Rightarrow (\beta{::}\mathsf{pcpo})) \Rightarrow ((\alpha)\mathsf{lift} \rightarrow \beta)$
defs $\mathsf{lift1}\ f \equiv (\Lambda al.\ \mathsf{case}\ al\ \mathsf{of}\ \mathsf{Def}(a) \Rightarrow f(a) \mid \mathsf{Undef} \Rightarrow \perp)$

lifts HOL functions into HOLCF by lifting their domain, thus turning it into a pcpo. The type β is constrained to be a pcpo already. Winskel's notation $f_\perp$ is close to but not identical with lift1 f because he lifts both domain and range.

Now we can define D, the HOLCF version of C:

```
consts  D :: com ⇒ (state)lift → (state)lift
primrec
  D(SKIP) = (Λs.s)
  D(x := a) = lift1(λs. Def(s[a(s)/x]))
  D(c1;c2) = (D c2) oo (D c1)
  D(IF b THEN c1 ELSE c2) = lift1(λs. if b(s) then D c1‘(Def s) else D c2‘(Def s))
  D(WHILE b DO c) = fix‘(Λw. lift1(λs. if b(s) then w‘(D c‘(Def s)) else Def s))
```

Winskel suggests (in our notation) D :: com ⇒ state → (state)lift, but we cannot follow him: state → (state)lift is not a legal type in HOLCF because state is not a pcpo and is thus not a valid argument type for →. Winskel gets away with it because instead of pcpos he works with cpos which do not require $\perp$ elements. Using the discrete ordering, any type, in particular state, can be turned into a cpo, and the continuous functions between cpos again form a cpo.

To show the equivalence of C and D it seems natural to prove

$$(\mathsf{D}\ \mathsf{c}`\mathsf{Def(s)} = \mathsf{Def(t)}) \ = \ ((\mathsf{s},\mathsf{t}) \in \mathsf{C(c)})$$

by induction on c. This should work fine, except that it requires a good deal of machinery to relate the two fixpoint operators lfp on sets and fix on cpos. This is beyond the scope of this paper and we turn to proof reuse: the equivalence

$$(\mathsf{D}\ \mathsf{c}`\mathsf{Def(s)} = \mathsf{Def(t)}) \ = \ (<\mathsf{c},\mathsf{s}> \stackrel{c}{\to} \mathsf{t})$$

is proved like its relative (5): the $\Longrightarrow$-direction by induction on c with nested fixpoint induction in the WHILE-case, the $\Longleftarrow$-direction by rule induction. However, this time things are not quite as straightforward. Without going into details, let us just say that an unexpected amount of HOLCF specific reasoning is necessary. For example, we needed to show that D is strict: D c‘$\perp$ = $\perp$. None of these proofs are very hard, but they do require familiarity with domain theory.

7 The Future

We have only presented part of our actual formalization. There is also a simple compiler to a stack machine and its correctness proof. But this is still only a beginning. We expect to extend our formalization in two directions: the language will become richer (e.g. procedures) and the semantics finer (e.g. time and space complexity). Ideally, all formal reasoning about programs should be based upon a unified semantic framework like the one presented in this paper.

Acknowledgements. I wish to thank Glynn Winskel for his excellent book, Heiko Lötzbeyer and Robert Sandner for their initial formalization, Robert Sandner for his HOLCF proofs, Franz Regensburger and Birgit Schieder for (amongst other things) their debugging of my initial vc, Olaf Müller for HOLCF expertise, and Larry Paulson for the subtitle.

References

1. Y. Bertot and R. Fraer. Reasoning with executable specifications. In *TAPSOFT '95: Theory and Practice of Software Development*, volume 915 of *Lect. Notes in Comp. Sci.*, pages 531–545. Springer-Verlag, 1995.
2. M. Gordon. *Programming Language Theory and its Implementation.* Prentice-Hall, 1988.
3. M. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving.* Springer-Verlag, 1989.
4. M. Gordon and T. Melham. *Introduction to HOL: a theorem-proving environment for higher-order logic.* Cambridge University Press, 1993.
5. P. V. Homeier and D. F. Martin. Trustworthy tools for trustworthy programs: A verified verification condition generator. In T. Melham and J. Camilleri, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 859 of *Lect. Notes in Comp. Sci.*, pages 269–284. Springer-Verlag, 1994.
6. H. Lötzbeyer and R. Sandner. Proof of the equivalence of the operational and denotational semantics of IMP in Isabelle/ZF. Project report, Institut für Informatik, TU München, 1994.
7. L. C. Paulson. *Logic and Computation.* Cambridge University Press, 1987.
8. L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1994.
9. L. C. Paulson. Generic automatic proof tools. Technical Report 396, University of Cambridge, Computer Laboratory, 1996.
10. F. Regensburger. *HOLCF: Eine konservative Erweiterung von HOL um LCF.* PhD thesis, Technische Universität München, 1994.
11. F. Regensburger. HOLCF: Higher Order Logic of Computable Functions. In E. Schubert, P. Windley, and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 971 of *Lect. Notes in Comp. Sci.*, pages 293–307. Springer-Verlag, 1995.
12. L. van Benthem Jutting. *Checking Landau's "Grundlagen" in the AUTOMATH System.* PhD thesis, Eindhoven University of Technology, 1977.
13. G. Winskel. *The Formal Semantics of Programming Languages.* MIT Press, 1993.

This article was processed using the LaTeX macro package with LLNCS style