Formal Aspects of Computing (1998) 3: 1–000

Winskel is (almost) Right

Towards a Mechanized Semantics Textbook

Tobias Nipkow [1]

Institut für Informatik, Technische Universität München, 80290 München, Germany.
http://www.in.tum.de/~nipkow/

Keywords: Semantics; Programming language; Theorem proving; Higher-order logic; HOL; Isabelle

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

1. Introduction

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 [vBJ77]. Our paper summarizes the formalization of the first 100 pages of a textbook on programming language semantics [Win93]. 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.
- To demonstrate not just the mere possibility of such an undertaking, but to show that, given the right logical infrastructure, it is reasonably straightforward and the result is both readable and fairly close to the original text.

[1] Research supported by DFG Schwerpunktprogramm *Deduktion*.

It is a tribute to Winskel's thoroughness that we found only 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 5.3) and is easily fixed.

The idea of embedding the semantics of a programming language in a theorem prover goes back at least to Gordon [Gor89]. 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 5.4) and treat the thorny issue of partial functions in a logic of total functions (Section 6).

No familiarity with Winskel's book itself is required, but the reader should have a basic understanding of programming language semantics.

The complete formalization (including proofs) is available on the internet at `http://www.in.tum.de/~isabelle/library/HOL/IMP/`.

2. Isabelle/HOL

Isabelle/HOL is the instantiation of the generic interactive theorem prover Isabelle [Pau94] with Church's formulation of Higher Order Logic and is very close to Gordon's HOL system [GM93]. In this paper HOL is short for Isabelle/HOL.

Below you find a short introduction to HOL's surface syntax; for readability, HOL's ASCII syntax is (automatically) turned into LaTeX throughout the paper:

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 solved by either of two workhorses, rewriting or the "classical reasoner" (`fast_tac` in Isabelle parlance [Pau97]). The latter is tableau-based and provides a reasonable degree of automation for predicate calculus proofs; its success depends on the right selection of lemmas supplied as parameters.

3. 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 [LS94]. 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 [Gor89]. We ignore these syntactic issues and focus on the semantic side of things.

4. Operational Semantics

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

4.1. Evaluation Semantics

Evaluation semantics (sometimes called "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:

$$\textbf{translations}\quad <c,s> \xrightarrow{c} t \;\equiv\; (c,s,t) \in evalc$$

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

inductive evalc

$$<SKIP,s> \xrightarrow{c} s$$

$$<x := a,s> \xrightarrow{c} s[a(s)/x]$$

$$[\![\ <\mathsf{c1,s}> \xrightarrow{c} \mathsf{s1};\ <\mathsf{c2,s1}> \xrightarrow{c} \mathsf{s2}\ ]\!] \implies\ <\mathsf{c1;c2,\ s}> \xrightarrow{c} \mathsf{s2}$$

$$[\![\ \ \ \mathsf{b\ s};\ <\mathsf{c1,s}> \xrightarrow{c} \mathsf{t}\ ]\!] \implies\ <\mathsf{IF\ b\ THEN\ c1\ ELSE\ c2,\ s}> \xrightarrow{c} \mathsf{t}$$
$$[\![\ \neg\ \mathsf{b\ s};\ <\mathsf{c2,s}> \xrightarrow{c} \mathsf{t}\ ]\!] \implies\ <\mathsf{IF\ b\ THEN\ c1\ ELSE\ c2,\ s}> \xrightarrow{c} \mathsf{t}$$

$$\neg\ \mathsf{b\ s} \implies\ <\mathsf{WHILE\ b\ DO\ c,\ s}> \xrightarrow{c} \mathsf{s}$$
$$[\![\ \mathsf{b\ s};\ <\mathsf{c,s}> \xrightarrow{c} \mathsf{s1};\ <\mathsf{WHILE\ b\ DO\ c,\ s1}> \xrightarrow{c} \mathsf{s2}\ ]\!] \implies\ <\mathsf{WHILE\ b\ DO\ c,\ s}> \xrightarrow{c} \mathsf{s2}$$

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 [Win93], which will be our major weapon in the proofs to come. The assignment command is defined in terms of an auxiliary function on states:

consts assign :: state ⇒ val ⇒ loc ⇒ state   ("_[_/_]")

Note that _[_/_] is left completely undefined: none of the proofs depend on it, something I only realized very late in the day. Of course, proofs about individual programs rather than the language do require a definition of _[_/_].

4.2. 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 evaluation 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 (SKIP, 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 $\mathsf{cs} \xrightarrow{1} \mathsf{cs'}\ \equiv\ \mathsf{(cs,cs')} \in \mathsf{evalc1}$

The inductive definition of evalc1 is straightforward:

inductive evalc1

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

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

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

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

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

The desired equivalence theorem is

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

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

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

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

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

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 $(\mathsf{c,s}) \xrightarrow{*} (\mathsf{SKIP,t})$ in the WHILE-case. The nested induction requires the following lemma:

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

This lemma is stronger than the one suggested by Winskel, where $\xrightarrow{n}$ and $\xrightarrow{m}$ 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 $\xrightarrow{n}$ at all. For a start, we can prove the generalized lemma

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

directly by induction on the structure (as opposed to the length) of $(\mathsf{c1,s1}) \xrightarrow{*} (\mathsf{SKIP,s2})$. As above, this yields the $\Longrightarrow$-direction of (1). Note that the generalization of (SKIP,s3) to 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 [BF95] for related material). The key lemma

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

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

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

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

5. 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 com_den = (state ∗ state)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  C :: com ⇒ com_den
primrec
  C(SKIP)                    = id
  C(x := a)                  = {(s,t). t = s[a(s)/x]}
  C(c1;c2)                   = C(c2) O C(c1)
  C(IF b THEN c1 ELSE c2)    = {(s,t). (s,t) ∈ C(c1) ∧ b(s)} ∪
                               {(s,t). (s,t) ∈ C(c2) ∧ ¬ b(s)}
  C(WHILE b DO c)            = lfp(Γ b (C c))
```

where Γ is an auxiliary function:

```
consts  Γ :: bexp ⇒ com_den ⇒ (com_den ⇒ com_den)
defs    Γ b cd  ≡  (λR.{(s,t). (s,t) ∈ (R O cd) ∧ b(s)} ∪ {(s,t). s=t ∧ ¬ b(s)})
```

This definition relies heavily on HOL's theory of sets and relations: O is composition of relations, lfp :: $(\alpha$ set $\Rightarrow \alpha$ set$) \Rightarrow \alpha$ set computes the least fixpoint of a monotone function on sets, and {e . P} is Isabelle's way of writing $\{e \mid P\}$. 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):

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

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

Monotonicity of Γ, i.e. mono(Γ b cd), is proved automatically. A simple consequence is the following recursion equation:

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

5.1. Denotational Semantics = Evaluation Semantics

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

$\langle c,s\rangle \xrightarrow{c} t \Longrightarrow (s,t) \in C(c)$

is proved by rule induction. The opposite direction

$(s,t) \in C(c) \Longrightarrow \langle c,s\rangle \xrightarrow{c} t$

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:

$$(\mathsf{s},\mathsf{t}) \in \mathsf{C}(\mathsf{c}) \;=\; (\langle \mathsf{c},\mathsf{s}\rangle \xrightarrow{c} \mathsf{t}) \tag{5}$$

6. 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).

6.1. Soundness of Hoare Logic

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).

6.2. Weakest (Liberal) Precondition Semantics

Weakest precondition semantics is Dijkstra's alternative [Dij76] to Hoare logic. Winskel treats it only briefly at the end of his chapter on axiomatic semantics. The concept of weakest preconditions is however already used in the proof of relative completeness of Hoare logic. Hence we cover weakest preconditions first.

The weakest precondition of a postcondition Q w.r.t. a command c is an assertion that characterizes all initial states which lead to Q, provided c terminates. Semantically, the weakest precondition is trivial:

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

Note that strictly speaking this is the weakest *liberal* precondition because we are in a partial correctness framework. We follow Winskel and drop the adjective "liberal" because there is no danger of confusion.

The following recursion equations for wp are derived almost automatically:

$$\mathsf{wp\ SKIP\ Q} = \mathsf{Q} \tag{6}$$
$$\mathsf{wp\ (x:=a)\ Q} = (\lambda \mathsf{s.Q(s[a\ s/x])}) \tag{7}$$
$$\mathsf{wp\ (c;d)\ Q} = \mathsf{wp\ c\ (wp\ d\ Q)} \tag{8}$$
$$\mathsf{wp\ (IF\ b\ THEN\ c\ ELSE\ d)\ Q} = (\lambda \mathsf{s}.\ (\mathsf{b\ s} \longrightarrow \mathsf{wp\ c\ Q\ s}) \wedge (\neg\ \mathsf{b\ s} \longrightarrow \mathsf{wp\ d\ Q\ s})) \tag{9}$$

$$\mathsf{b(s)} \Longrightarrow \mathsf{wp\ (WHILE\ b\ DO\ c)\ Q\ s} = \mathsf{wp\ (c; WHILE\ b\ DO\ c)\ Q\ s} \tag{10}$$
$$\neg\ \mathsf{b(s)} \Longrightarrow \mathsf{wp\ (WHILE\ b\ DO\ c)\ Q\ s} = \mathsf{Q\ s} \tag{11}$$

This is all we need for the proof of relative completeness of Hoare logic. However, we can go a bit further and derive a greatest fixpoint characterization of wp in the WHILE-case:

$$\mathsf{wp\ (WHILE\ b\ DO\ c)\ Q\ s} = (\mathsf{s} \in \mathsf{gfp}(\lambda \mathsf{S}.\{\mathsf{s.\ if\ b\ s\ then\ wp\ c\ }(\lambda \mathsf{s.s} \in \mathsf{S})\mathsf{\ s\ else\ Q\ s}\})) \tag{12}$$

The function $\mathsf{gfp} :: (\alpha\ \mathsf{set} \Rightarrow \alpha\ \mathsf{set}) \Rightarrow \alpha\ \mathsf{set}$ computes the greatest fixpoint of a monotone function on sets. It is inherited from the standard HOL libraries. Note that α set and $\alpha \Rightarrow$ bool are distinct types, which is why we need to convert between them explicitly via $\{\mathsf{s.\ P}\}$ and $\lambda \mathsf{s.\ s} \in \mathsf{S}$.

The two key theorems about gfp express that the result of gfp is indeed a fixpoint and satisfies a coinduction principle:

$$\mathsf{mono(f)} \Longrightarrow \mathsf{gfp(f)} = \mathsf{f(gfp(f))} \tag{13}$$
$$[\![\ \mathsf{a} \in \mathsf{X};\ \ \mathsf{X} \subseteq \mathsf{f(X)}\ ]\!] \Longrightarrow \mathsf{a} \in \mathsf{gfp(f)} \tag{14}$$

The proof of (12) is not automatic and quite instructive. The $\Longrightarrow$-direction

$\mathsf{wp\ (WHILE\ b\ DO\ c)\ Q\ s} \Longrightarrow \mathsf{s} \in \mathsf{gfp}(\lambda \mathsf{S}.\ \{\mathsf{s.}\ ...\})$

is proved by coinduction (14). The unknown set X is instantiated to $\{\mathsf{s.\ wp\ (WHILE\ b\ DO\ c)\ Q\ s}\}$ using the lemma $\mathsf{P\ a} \Longrightarrow \mathsf{a} \in \{\mathsf{x\ .\ P\ x}\}$, and the second premise $\mathsf{X} \subseteq \mathsf{f(X)}$ of the coinduction theorem is proved by logical reasoning and rewriting. For the proof of the $\Longleftarrow$-direction of (12) definitions are expanded and the subgoal is transformed into

$[\![\ (\mathsf{s,\ t}) \in \mathsf{lfp}(\lambda \mathsf{R}.\ \{(\mathsf{s,t}).\ ...\})\ ]\!] \Longrightarrow \mathsf{s} \in \mathsf{gfp}(\lambda \mathsf{S}.\ \{\mathsf{s.}\ ...\}) \longrightarrow \mathsf{Q\ t}$

which has an easy proof by lfp-induction (3) and an appeal to (13).

Note that Winskel proceeds differently: he defines both wp as above and a separate function $\mathcal{P}t$ of essentially the same type by primitive recursion on com as in (6)–(9) and (12) (but with $\mathcal{P}t$ in place of wp). Then he proves (as an exercise) the equivalence of wp and $\mathcal{P}t$, whereas we derive the recursion equations for $\mathcal{P}t$ from the definition of wp.

Armed with wp, we now return to Hoare logic.

6.3. Relative Completeness of Hoare Logic

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

$$\models \{\mathsf{P}\}\ \mathsf{c}\ \{\mathsf{Q}\} \implies \vdash \{\mathsf{P}\}\ \mathsf{c}\ \{\mathsf{Q}\} \tag{15}$$

it falls short of the corresponding completeness statement in the literature. The reason is our semantic view of assertions. Although (15) 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 calls an assertion language **expressive** if for every command c and assertion Q, there is an assertion P with semantics $\mathsf{wp\ c\ Q}$. 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 wp. It remains to be shown that it leads to completeness.

The key lemma looks trivial

$$\forall \mathsf{Q}.\ \vdash \{\mathsf{wp\ c\ Q}\}\ \mathsf{c}\ \{\mathsf{Q}\} \tag{16}$$

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

$$\begin{array}{l}(\forall \mathsf{Q}.\ \vdash \{\mathsf{wp\ c\ Q}\}\ \mathsf{c}\ \{\mathsf{Q}\}) \Longrightarrow \\ \vdash \{\mathsf{wp\ (WHILE\ b\ DO\ c)\ Q}\}\ \mathsf{WHILE\ b\ DO\ c}\ \{\mathsf{Q}\}\end{array}$$

In the proof of this subgoal, Winskel claims that from $\models \{\lambda \mathsf{s}.\ \mathsf{P(s)} \wedge \mathsf{b(s)}\}\ \mathsf{c}\ \{\mathsf{P}\}$, where $\mathsf{P} \equiv \mathsf{wp\ (WHILE\ b\ DO\ c)\ Q}$, it follows by induction hypothesis that $\vdash \{\lambda \mathsf{s}.\ \mathsf{P(s)} \wedge \mathsf{b(s)}\}\ \mathsf{c}\ \{\mathsf{P}\}$. 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

$$\mathsf{P(s)} \wedge \mathsf{b(s)} \longrightarrow \mathsf{wp\ c\ P\ s} \tag{17}$$

$$\mathsf{P(s)} \wedge \neg\, \mathsf{b(s)} \longrightarrow \mathsf{Q} \tag{18}$$

$$\vdash \{\mathsf{wp\ c\ P}\}\ \mathsf{c}\ \{\mathsf{P}\} \tag{19}$$

where again $\mathsf{P} \equiv \mathsf{wp}\ (\mathsf{WHILE\ b\ DO\ c})\ \mathsf{Q}$. The first two follow essentially from (4) via the lemmas (10) and (11). The third one is an instance of the induction hypothesis.

From (17) and (19) we obtain $\vdash \{\lambda \mathsf{s}.\ \mathsf{P(s)} \wedge \mathsf{b(s)}\}\ \mathsf{c}\ \{\mathsf{P}\}$ by the rule of consequence. The WHILE-rule turns this into $\vdash \{\mathsf{P}\}\ \mathsf{WHILE\ b\ DO\ c}\ \{\lambda \mathsf{s}.\ \mathsf{P(s)} \wedge \neg \mathsf{b(s)}\}$. Using (18) and the rule of consequence this yields the desired $\vdash \{\mathsf{P}\}\ \mathsf{WHILE\ b\ DO\ c}\ \{\mathsf{Q}\}$. The Isabelle proof requires some guidance but no instantiations. Unsurprisingly, I later learned that this is precisely the standard proof [AO91].

Having proved (16), completeness (15) follows easily by the rule of consequence.

6.4. 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 [Gor88] 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, awp :: acom ⇒ assn ⇒ assn
```

where vc returns the verification condition and awp the weakest (liberal) precondition. The "a" in awp stands for "annotated" and was chosen to distinguish it from wp above. The soundness property we are aiming for is

$$(\forall \mathsf{s}.\ \mathsf{vc\ c\ Q\ s}) \Longrightarrow\ \vdash \{\mathsf{awp\ c\ Q}\}\ \mathsf{astrip\ c}\ \{\mathsf{Q}\} \tag{20}$$

The reason for dragging in awp 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 awp c2 in order to compute vc ($\mathsf{Asemi\ c1\ c2}$)). This time we cannot take the semantic way out (see wp) because we are really meant to compute an assertion, i.e. some syntactic entity. The following definition of awp agrees with wp, except for the final clause:

```
primrec
  awp Askip Q              = Q
  awp (Aass x a) Q         = (λs.Q(s[a(s)/x]))
  awp (Asemi c1 c2) Q      = awp c1 (awp c2 Q)
  awp (Aif b c1 c2) Q      = (λs. (b(s) ⟶ awp c1 Q s) ∧ (¬ b(s) ⟶ awp c2 Q s))
  awp (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 that is too strong or not invariant? If I is too strong, awp computes too strong a precondition and the conclusion of (20) is weaker than it could be. If I is not even invariant, the verification conditions computed by vc are not valid either, thus rendering (20) 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 (awp d Q) s ∧ vc d Q s)
  vc (Aif b c d) Q         = (λs. vc c Q s ∧ vc d Q s)
  vc (Awhile b I c) Q      = (λs.  (I(s) ∧ b(s) ⟶ awp c I s) ∧ (I(s) ∧ ¬ b(s) ⟶ Q(s))
                                   ∧ vc c I s)
```

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

The proof of soundness (20) 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:

```
∀P Q. (∀s. P s ⟶ Q s) ⟶ (∀s. awp c P s ⟶ awp c Q s)
∀P Q. (∀s. P s ⟶ Q s) ⟶ (∀s. vc c P s ⟶ vc c Q s)
```

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

```
⊢ {P} c {Q} ⟹ (∃ac. astrip ac = c ∧ (∀s. vc ac Q s) ∧ (∀s. P s ⟶ awp ac Q s))
```

is a straightforward rule induction on $\vdash \{\mathsf{P}\}\ \mathsf{c}\ \{\mathsf{Q}\}$. 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 awp can be combined into a single function returning a pair of the weakest precondition and the verification condition.

```
consts  vcawp :: acom ⇒ assn ⇒ assn ∗ assn
primrec
  vcawp Askip Q            = (λs.True, Q)
  vcawp (Aass x a) Q       = (λs.True, λs.Q(s[a s/x]))
  vcawp (Asemi c d) Q      = (let (vcd,wpd) = vcawp d Q;
                                  (vcc,wpc) = vcawp c wpd
                              in (λs. vcc s ∧ vcd s, wpc))
  vcawp (Aif b c d) Q      = (let (vcd,wpd) = vcawp d Q;
                                  (vcc,wpc) = vcawp c Q
                              in (λs. vcc s ∧ vcd s, λs.(b s ⟶ wpc s) ∧ (¬ b s ⟶ wpd s)))
  vcawp(Awhile b I c) Q    = (let (vcc,wpc) = vcawp c I
```

in (λs. (I s ∧ ~b s ⟶ Q s) ∧ (I s ∧ b s ⟶ wpc s) ∧ vcc s, I))

This function only traverses the command once, in contrast to the above solution. The relationship vcawp c Q = (vc c Q, awp c Q) is proved automatically by induction on c. The structure of vcawp is essentially the same as that of the verification condition generator *vcg*1 by Homeier and Martin [HM95], who prove soundness but not completeness of their function.

7. 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 [Reg94, Reg95, MNOS98] is a conservative extension of HOL with the notions of domain theory [Pau87] employing Isabelle's axiomatic type classes [Wen97], which extend Haskell's type classes [HPJW92] with axioms. In particular, HOLCF provides

- a class cpo of types which come equipped with a complete partial order $\sqsubseteq$.
- a subclass pcpo (pointed cpo) of cpos that also have a least element $\perp$.
- a space $\to$ of continuous functions between cpos, together with its own abstraction Λ, infix application ‘, composition oo, and fixpoint operator fix. The latter is only defined on pcpos. If τ_1 is a cpo and τ_2 a pcpo, then $\tau_1 \to \tau_2$ is proved to be again a pcpo.

Winskel suggests to model the denotational semantics of a command as a continuous function of type $\Sigma \to \Sigma_\perp$, where Σ is the set of states. This works because he regards any set (in particular Σ) as a **discrete** cpo, where the ordering is the identity, and the "lifted" cpo $\Sigma_\perp$ is a cpo with least element $\perp$. Both discrete cpos and lifting also exist in HOLCF.

7.1. Discrete cpos

HOLCF provides a type constructor

datatype (α)discr = Discr α

which turns every HOL type τ into the discrete cpo (τ)discr (by equating $\sqsubseteq$ and =). It is shown that every function of type (α)discr $\Rightarrow \beta$, where β is a cpo, is continuous. There is also a function undiscr :: (α)discr $\Rightarrow \alpha$ which is the inverse of the constructor Discr.

Why this extra type constructor? In HOLCF, we cannot just regard an arbitrary type as a discrete cpo. In particular the proposition that state is a cpo is not supported because state is not an atomic type but an abbreviation for a compound type. This restriction of the type system is also shared by Haskell.

Mark Jones [Jon94] has developed a closely related type system which does not have this restriction.

7.2. Lifting

Lifting usually means the construction of one cpo from another one by adjoining a new $\perp$ element. In order to lift arbitrary HOL types without having to turn them into discrete cpos first, HOLCF provides

datatype (α)lift = Undef | Def α

and turns (α)lift into a pcpo by defining $x \sqsubseteq y \equiv x=\mathsf{Undef} \vee x=y$ and $\perp \equiv$ Undef.

7.3. D

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

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

This definition is pretty much what one expects, except that the explicit bijection undiscr spoils the view a little. There is also the auxiliary function dlift :: $((\alpha)\mathsf{discr} \to \beta) \Rightarrow ((\alpha)\mathsf{lift} \to \beta)$ that lifts a function from a discrete cpo to one from a lifted cpo:

```
dlift f ≡ (Λx.case x of Undef ⇒ ⊥ | Def(y) ⇒ f‘(Discr y))
```

Thanks to the infrastructure for type lift, dlift f is automatically shown to be continuous for every f.

A previous version of D, documented in [Nip96], had the type com $\Rightarrow$ (state)lift $\to$ (state)lift. At the time this was necessary because HOLCF was based only on class pcpo and discrete cpos were not available. This meant that the definition actually looked simpler, e.g. (D c2) oo (D c1) instead of dlift(D c2) oo (D c1). However, the fixpoint induction described below became tricky because one needed to adjoin a strictness statement to the theorem in order for the induction to go through. Hence we abandoned this design. It was only after the introduction of class pcpo [MNOS98] that the above improved semantics became possible.

The HOLCF semantics above vindicates Winskel's decision to spend hardly more than a page of prose on a domain theoretic semantics of IMP. The essence of the semantics is lost a little in the domain theoretic noise. Although the formalism stays quite manageable, it is preferable to move to domain theory only if necessary (e.g. when dealing with more complex semantic domains, in particular with infinite objects [MN97]) and to stick to sets otherwise. This is exactly what Winskel does.

7.4. Equivalence of C and D

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

(D c'(Discr s) = Def(t)) = ((s,t) $\in$ 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 pcpos. Instead we turn to proof reuse: the equivalence

(D c'(Discr s) = Def(t)) = (<c,s> $\xrightarrow{c}$ t)

is proved like its relative (5):

$\Longleftarrow$ by rule induction, which is again almost automatic.

$\Longrightarrow$ by induction on c with nested fixpoint induction in the WHILE case. The rule of fixpoint induction

$[\![$ adm P; P $\perp$; $\bigwedge$x. P x $\Longrightarrow$ P(F'x) $]\!]$ $\Longrightarrow$ P(fix'F)

is a theorem of HOLCF. It is applicable only if P is *admissible* [Pau87] (called *inclusive* by Winskel), expressed as the premise adm P, where adm :: (α $\Rightarrow$ bool) $\Rightarrow$ bool. Fortunately, HOLCF already contains the lemma adm(λu.P(u'x)), where the argument type of P must be a flat pcpo, and a lemma stating flatness of (α)lift. These two lemmas automatically prove admissibility of our particular application of fixpoint induction. The whole proof is a bit longer than its counterpart for C.

8. Conclusion

What have we gained in this whole exercise? Apart from the obvious answers (a foundation for formalizations of more exciting languages and more practical applications such as compiler verification; debugging of the original text; enhanced understanding of the concepts), to us the most important point is *simplicity*. This is the result of applying a theorem prover as Occam's razor: we have not tried to formalize everything precisely the way Winskel does, but have tried to find the simplest formalization. Paper is patient, but a theorem prover punishes every unnecessary complication. In two cases (the transition semantics and the verification condition generator) this has resulted in what we believe are logically simpler theories. In one case, the denotational semantics in HOLCF, our formalization is logically equivalent to Winskel's but looks a bit more complicated than his would look (had he spelled it out) because of a restriction in the type system.

It is impossible to estimate the time it took to develop the complete formalization. Starting from a student project [LS94] that formalized the relational and the natural semantics and proved their equivalence (in ZF set theory), more and more features were added in the course of the following 2 1/2 years. However, each extension took no more than a week. What we can quantify is the size of the resulting HOL theories in lines, including all the proofs:

Operational	300
Denotational (sets)	100
Axiomatic	300
Denotational (HOLCF)	100

But does it scale up? We believe it does, and have two significant data points in support:

- Nipkow and Oheimb [NO98, ON98] have formalized a large part of Java (excluding parallelism) and have proved type soundness. Their evaluation semantics is at least 10 times the size of the one for IMP.
- Müller [MN97, DGM97, Mül98a, Mül98b] has formalized many aspects of I/O automata, an abstract model of distributed systems. He found that HOLCF is extremely well suited for describing the semantics of I/O automata in terms of finite and infinite streams.

Coming back to the textbook perspective, one may wonder if and how our formalization and proofs can help when teaching this material. Unfortunately, Isabelle proof scripts are opaque, even for the expert. We are working on this aspect and hope that eventually these formal proofs could become part of a (possibly electronic) textbook. Going one step further, students could even perform their own proofs in Isabelle. We do this routinely in a special theorem proving course, but in a general semantics course the overhead of coming up to speed on Isabelle makes this very difficult.

A word of warning to the teacher: students rarely share the logician's enthusiasm for beautiful formalizations and proofs. In my own experience, the bright students gain from increased formality whereas the poor ones are left further behind. Thus I recommend to take advantage of the simplifications (and corrections!) that a formal proof may bring with it when teaching the material. But the proof styles suitable for the computer and the student are (still) quite different and should not be confused.

Acknowledgments. 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. Two anonymous referees encouraged me to say more about the 'why' and 'whither'.

References

[AO91] Krzysztof R. Apt and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, 1991.

[BF95] Yves Bertot and Ranan 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.

[DGM97] Marco Devillers, David Griffioen, and Olaf Müller. Possibly infinite sequences in theorem provers: A comparative study. In E.L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics*, volume 1275 of *Lect. Notes in Comp. Sci.*, pages 89–104. Springer-Verlag, 1997.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: a theorem-proving environment for higher order logic.* Cambridge University Press, 1993.

[Gor88] M.C.J. Gordon. *Programming Language Theory and its Implementation.* Prentice-Hall, 1988.

[Gor89] M.C.J. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving.* Springer-Verlag, 1989.

[HM95] Peter V. Homeier and David F. Martin. A mechanically verified verification condition generator. *The Computer Journal*, 38:131–141, 1995.

[HPJW92] Paul Hudak, Simon Peyton Jones, and Philip Wadler. Report on the programming language Haskell: A non-strict, purely functional language. *ACM SIGPLAN Notices*, 27(5), May 1992. Version 1.2.

[Jon94] Mark P. Jones. *Qualified Types: Theory and Practice.* Cambridge University Press, 1994.

[LS94] Heiko Lötzbeyer and Robert Sandner. Proof of the equivalence of the operational and denotational semantics of IMP in Isabelle/ZF. Project report, Institut für Informatik, TU München, 1994.

[MN97] Olaf Müller and Tobias Nipkow. Traces of I/O automata in Isabelle/HOLCF. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development*, volume 1214 of *Lect. Notes in Comp. Sci.*, pages 580–594. Springer-Verlag, 1997.

[MNOS98] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oskar Slotosch. HOLCF = HOL + LCF. Submitted for publication, 1998.

[Mül98a] Olaf Müller. I/O automata and beyond — temporal logic and abstraction in Isabelle. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics (TPHOLs'98)*, volume ? of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1998. To appear.

[Mül98b] Olaf Müller. *A Verification Environment for I/O-Automata Based on Formalized Meta-Theory.* PhD thesis, Institut für Informatik, Techn. Univ. München, 1998. Forthcoming.

[Nip96] Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *Lect. Notes in Comp. Sci.*, pages 180–192. Springer-Verlag, 1996.

[NO98] Tobias Nipkow and David von Oheimb. Java$_{\ell ight}$ is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 161–170. ACM Press, 1998.

[ON98] David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume ? of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1998. To appear.

[Pau87] Lawrence C. Paulson. *Logic and Computation.* Cambridge University Press, 1987.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1994.

[Pau97] Lawrence C. Paulson. Generic automatic proof tools. In R. Veroff, editor, *Automated Reasoning and its Applications.* MIT Press, 1997. Also Report 396, Computer Laboratory, University of Cambridge.

[Reg94] Franz Regensburger. *HOLCF: Eine konservative Erweiterung von HOL um LCF.* PhD thesis, Technische Universität München, 1994.

[Reg95] Franz Regensburger. HOLCF: Higher Order Logic of Computable Functions. In E.T. Schubert, P.J. 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.

[vBJ77] L.S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the AUTOMATH System.* PhD thesis, Eindhoven University of Technology, 1977.

[Wen97] Markus Wenzel. Type classes and overloading in higher-order logic. In E.L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics*, volume 1275 of *Lect. Notes in Comp. Sci.*, pages 307–322. Springer-Verlag, 1997.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages.* MIT Press, 1993.