Mechanized Operational Semantics via (Co)Induction

Simon J. Ambler and Roy L. Crole

Leicester University, Leicester LE1 7RH, U.K.

Abstract. We give a fully automated description of a small programming language PL in the theorem prover Isabelle98. The language syntax and semantics are encoded, and we formally verify a range of semantic properties. This is achieved via uniform (co)inductive methods. We encode notions of bisimulation and contextual equivalence. The main original contribution of this paper is a fully automated proof that PL bisimulation coincides with PL contextual equivalence.

1 Introduction

The design of new programming languages which are well-principled, reliable and expressive is an important goal of Computer Science. Semantics has a role to play here, in that it provides a firm basis for establishing the properties and behaviour of language features and programs. A number of advances in the methods of operational semantics have been made over the last few years, in particular, in the study of higher order operational semantics and program equivalences—many of these advances are detailed below. In this paper we make a contribution by considering mechanized support for reasoning about operational semantics. In particular, we specify and verify properties of a core language, using tactical verification within the theorem prover Isabelle98 [Pau94b]. We give a presentation in which the concepts are expressed as uniformly as possible using the framework for (co)inductive definitions within Isabelle-HOL. We hope that the key principles of our methodology are sufficiently flexible that they can be adapted to new languages, provided that their semantics can be given using inductive and coinductive definitions.

One framework for giving formalized semantic specifications is the structured operational semantics (SOS) introduced by Plotkin [Plo81]. This allows high level specification of programming languages, abstracting from machine level detail. It is possible to present the full semantics of a non-trivial language in this form (for example [MMH97,Cro97]). SOS allows reasoning about programs and language properties, such as the verification that program execution is deterministic or that execution respects the type discipline (for example [CG99,Gor95a,Pit97]). At a more refined level, one might want to reason about the equivalence of different programs or their termination properties. This sort of reasoning is an essential basis for program refinement techniques and for verifying compiler optimization steps. Contextual equivalence is a useful and intuitive notion of program equivalence: two programs are regarded as equivalent if they behave in the same way in all possible situations. Abramsky's applicative bisimulation gives a notion of program equivalence in the lazy λ-calculus [Abr90] and in fact this can be adapted to give a general notion of program equivalence. In more general situations,

- applicative bisimulation often implies contextual equivalence (*); and
- for *some* languages the two notions coincide.

Howe [How89] showed that bisimilarity is a congruence—this essentially means we can reason about bisimilarity using simple algebraic manipulations. The idea is that contextual equivalence is a *natural* way to compare programs—but it is difficult to establish. As noted above (*), it is often sufficient to show bisimilarity and this is frequently more tractable. So we can show two programs to be contextually equivalent by instead showing they are bisimilar. These sorts of results have been obtained for a variety of languages such as PCFL [Pit97], a functional language with IO [CG99], the Object-calculus of Abadi and Cardelli [AC96,GHL97], and the linear λ-calculus [Bie97,Cro96]. The papers [Las98,MT91,Pit98] discuss when variants of the "standard" theories apply.

A key step towards turning these theoretical developments to practical advantage is to produce tools which correctly embody the theory. Over the last 10 to 15 years, automated support has been developed in packages such as Isabelle and HOL [MG93]. Recently, a number of people have made advances in semantic verification. Syme has mechanized a proof of type soundness for a fragment of Java [Sym97b] using DECLARE [Sym97a]; a similar verification has been made by Nipkow and von Oheimb [NvO98] within Isabelle-HOL. Börger and Schulte [BS98] have used ASMs to formalize a Java fragment. Collins and others [CG94,MG94,Sym93] have coded fragments of Standard ML in HOL. Nipkow has verified the equivalence of alternative semantic presentations of a simple imperative language [Nip98].

There is also a body of work on the automated verifications of type theory properties which are less directly concerned with the issues of programming language semantics. Altenkirch has given a proof of strong normalization of System F in the LEGO theorem prover [Alt93], and Coquand has shown normalization for simply typed lambda calculus written in ALF [Coq92]. The π-calculus has been encoded in the Calculus of Constructions by Hirschkoff [Hir97].

Our contribution is to provide, within Isabelle-HOL, the operational semantics of a small programming language called *PL* together with *a fully mechanized account* of program equivalence in *PL*. In particular, we define the notions applicative bisimulation and contextual equivalence for this language and use Howe's method to prove that these coincide. This is the first time, as far as we are aware, that such a proof has been fully mechanized. Each constituent part of *PL* is encoded as a theory. Each theory gives rise to new HOL definitions, together with introduction and elimination rules. We make substantial use of the Isabelle packages for datatypes, induction and coinduction. Overall, our mechanization specifies *PL* types, expressions, a type assignment system, an operational evaluation semantics, Kleene equality of programs, a divergence predicate, contextual equivalence, and bisimilarity. We have *formally verified*:

- *determinism of evaluation*,
- *type soundness*,
- *congruence of bisimilarity*, and the
- *coincidence of bisimilarity and contextual equivalence*,

together with a number of examples of properties of particular programs.

The paper proceeds as follows. In Section 2 we give a mathematical framework for induction and coinduction. Each of our Isabelle encodings can be seen as an instance of one of the definitions from this framework. The idea is that this enables us to give a uniform account of each of the many inductive and coinductive definitions which appear in this paper. In Section 3 we describe the syntax of a small programming language *PL*, and its type system. In Section 4 we provide an operational semantics. In Section 5 we define a notion of similarity and hence bisimilarity for *PL*. In Section 6 we define a binary relation (due to Howe [How89]) which is used to facilitate proofs in Section 8. In Section 7 we encode program divergence, which is also used in Section 8. In this final section, we prove that contextual equivalence and bisimilarity do coincide.

2 Inductive and Coinductive Definitions

We have achieved a uniform mechanization of *PL* by the use of inductive and coinductive methods. Recall [Cro98] that if $\theta: \mathcal{P}(X) \to \mathcal{P}(X)$ is a monotone endofunction on the powerset of a set X, then the least pre-fixed point $\mu\,\theta$ of θ is the subset of X *inductively* defined by θ and the greatest post-fixed point $\nu\,\theta$ is the subset of X *coinductively* defined by θ. The specification and properties of *PL* can be given (co)inductively. We shall show that each of these *PL* inductive and coinductive definitions is an instance of one of the following forms; and each is itself readily coded within Isabelle using formulas Ψ^{HOL} of higher order logic generated with the assistance of the (co)induction package [Pau94a]. The labels **I-Set** (etc) are referenced later on in the paper.

(Co)Inductive Sets and Functions

- To $\boxed{\text{define a set } A \in \mathcal{P}(U)}$ we specify a function $\theta_A: \mathcal{P}(U) \to \mathcal{P}(U)$ and take A to be either $\mu\,\theta_A$ (**I-Set**) or $\nu\,\theta_A$ (**C-Set**).
- To $\boxed{\text{define a function } f: \mathcal{P}(U) \to \mathcal{P}(U)}$ we specify a function $\theta_{f;X}: \mathcal{P}(U) \to \mathcal{P}(U)$ for each subset $X \subseteq U$ and take $f(X)$ to be either $\mu\,\theta_{f;X}$ (**I-Function**) or $\nu\,\theta_{f;X}$ (**C-Function**).

Throughout this paper, we show that each of semantic definitions we wish to encode in Isabelle conforms to one of the definitions of a set A or a function f given above. *Both* of these are themselves specified by defining some other function of the form $\theta: \mathcal{P}(U) \to \mathcal{P}(U)$. However, to encode $\theta: \mathcal{P}(U) \to \mathcal{P}(U)$ in Isabelle, we simply have to define a HOL formula $\Psi^{HOL}(z, X)$ such that $\forall z.\, \forall X \in \mathcal{P}(U).\, z \in \theta(X) \Longleftrightarrow \Psi^{HOL}(z, X)$.

3 The Syntax of *PL*

Syntax expressions in $\mathcal{E}$ are encoded in de Bruijn notation [dB72] That is, each variable is represented by a natural number, the index of its position in an environment given by the current scope of binding. We use a simple datatype for a λ-calculus with constants. The *expressions* of *PL* are given by the grammar

$$e ::= k \mid i \mid \lambda(e) \mid e\,e$$

where k ranges over constants and i ranges over variable indices. The set Cst of constants includes, for instance, cond, zero, suc, ncase, and rec to capture conditional tests, numbers and recursion. The theory `Expr.thy` defines the syntax of programs.

The *types* of *PL* are given by the grammar

$$\sigma ::= nat \mid bool \mid \sigma \to \sigma \mid \sigma * \sigma \mid list(\sigma)$$

The set of types is denoted by $\mathcal{T}$. A *typing environment* is a list of types $[\sigma_0, \ldots, \sigma_{n-1}]$ with σ_i indicating the type of the variable with index i in the current environment. A typical typing environment is denoted by Γ. The set $\mathcal{G}$ consists of all such environments. We define a subset $Cty \subseteq Cst \times \mathcal{T}$ which supplies types for each constant, for instance,

$$(\mathsf{zero}, nat) \in Cty \qquad \text{and} \qquad (\mathsf{cond}, bool \to \sigma \to \sigma) \in Cty$$

for all types σ. The type assignment relation is generated by the usual rules of λ-calculus, together with the assignment $\Gamma \vdash k : \sigma$ whenever $(k, \sigma) \in Cty$. We write $e : \sigma$ for $[\,] \vdash e : \sigma$ and $\mathcal{E}_\sigma$ for the set $\{\ e \ \mid\ e : \sigma\ \}$ of *programs* of type σ. The theory `Type.thy` defines the type system within Isabelle.

We have coded the syntax via λ-calculus using de Bruijn indices. This removes the need to axiomatize α-conversion which would constitute a significant body of work before establishing any non-trivial results. Others [GM96] have shown that axiomatizing α-conversion for implementation is a non-trivial task; moreover, our initial work with explicit variable names highlighted the difficulties of reasoning about the choice of fresh names for bound variables. We also investigated the use of higher order abstract syntax [JDH95] but this raised a problem in obtaining the schemes of recursion and induction over the structure of syntax needed for our proofs. Some of these issues are addressed in [JDS97] and in the recent work of Gabbay and Pitts [GP99] and Fiore, Plotkin and Turi [FPT99]. These approaches may eventually provide a viable alternative to de Bruijn encoding but, for the moment, the cost of taking a more concrete representation is well repaid in the benefit of being able to use standard schemes for dataype representation and induction. There is now a reasonable body of experience to demonstrate the effectiveness of the de Bruijn approach in proving meta-theoretic results about syntax with binding operators [Hir97,Hue94,Nip].

One cost of using de Bruijn notation is, of course, that expressions can be difficult to read. This is not a substantial problem in proving theorems about the general properties of the language as we are usually working with program expressions denoted by Isabelle variables. For work involving large explicit program fragments one might have to write a parser and pretty printer to provide a more usable interface. More significant is that de Bruijn notation replaces reasoning about the renaming of bound variables via elementary arithmetic manipulation of the variable indices. It is fortunate that this manipulation can be managed by establishing a few well chosen lemmas because the proof of these can involve some quite tedious arithmetic reasoning. We next discuss the key issues of weakening and substitution.

3.1 Weakening and Substitution

The operational semantics which we have described is a substitution based system. As such, the usual properties of variable weakening, and expression substitution, must be encoded. This requires care. If σ is a type and Γ a typing environment, then $\texttt{newVar}\, i\, \Gamma\, \sigma$ is the new environment formed by inserting σ into the ith position. The weakening of a type assignment $\Gamma \vdash e : \sigma$ in PL requires that the indices of existing variables in e must be adjusted to take account of their new position. In Isabelle, we prove

$$[|\ \Gamma \vdash e : \sigma,\ j \leq \texttt{length}(\Gamma)\ |] \Longrightarrow \texttt{newVar}\, j\, \Gamma\, \tau\ \vdash \texttt{lift}\, e\, j : \sigma$$

where $\texttt{lift}\, e\ j$ is the result of incrementing each of the indices in e corresponding to a free variable beyond the jth position, by one. Substitution within type assignments eliminates free variables from the current environment. In Isabelle, we prove

$$[|\ j \leq \texttt{length}(\Gamma),\ \texttt{newVar}\, j\, \Gamma\, \sigma\ \vdash e_1 : \tau,\ \Gamma \vdash e_2 : \sigma\ |] \Longrightarrow \Gamma \vdash e_1[e_2/j] : \tau$$

4 Operational Evaluation

We define *operational evaluation* through judgements of the form $e \Downarrow ve$ which denote a relation between expressions; the Isabelle theory is called `Eval.thy`. This binary relation is inductively defined by rules such as the examples in Figure 1. Note that we have opted for a presentation in which destructors such as cond are evaluated immediately when given a first argument. This reduces the total number of rules.

4.1 Subject Reduction

Recall that Subject Reduction is the property that the types of expressions do not change on execution. The Isabelle proof of Subject Reduction is quite elegant. We show by induction on the derivation of $e \Downarrow ve$ that

$$e \Downarrow ve \Longrightarrow \forall \sigma . e : \sigma \rightarrow ve : \sigma$$

$$\frac{}{k \Downarrow k} \qquad \frac{}{\lambda(e) \Downarrow \lambda(e)} \qquad \frac{e \Downarrow \lambda(e') \quad e'[e''/0] \Downarrow ve}{e\,e'' \Downarrow ve}$$

$$\frac{e \Downarrow \mathsf{cond} \quad e' \Downarrow \mathsf{true}}{e\,e' \Downarrow \lambda(\lambda(1))} \qquad \frac{e \Downarrow \mathsf{cond} \quad e' \Downarrow \mathsf{false}}{e\,e' \Downarrow \lambda(\lambda(0))} \qquad \frac{e \Downarrow \mathsf{rec} \quad e'\,(\mathsf{rec}\,e') \Downarrow ve}{e\,e' \Downarrow ve}$$

$$\frac{e \Downarrow \mathsf{suc}}{e\,e' \Downarrow \mathsf{suc}\,e'} \qquad \frac{e \Downarrow \mathsf{ncase} \quad e' \Downarrow \mathsf{zero}}{e\,e' \Downarrow \lambda(\lambda(1))} \qquad \frac{e \Downarrow \mathsf{ncase} \quad e' \Downarrow \mathsf{suc}\,e''}{e\,e' \Downarrow \lambda(\lambda(0\,e''))}$$

Fig. 1. *PL* Evaluation Relation—Function Application, Conditionals, Numbers and Recursion

The inductive definition package supplied in Isabelle generates a definition in the form **I-Set** automatically from a set of introduction rules. The definition of a set A comes with equipped with an appropriate elimination rule and induction rule which are can be used on the premises of a sequent to eliminate an assumption of the form $a \in A$. In particular, the set $eval = \{(p, v) \mid p \Downarrow v\}$ is defined from the rules in Figure 1 and corresponding induction rules allows us to eliminate an assumption of the form $p \Downarrow v$. Applying this to the goal above, we obtain a total of seventeen subgoals, each corresponding to a last step in a derivation of $p \Downarrow v$. These are all relatively simple and can be established by the tactic `fast_tac` which examines elimination and introduction rules by depth-first search.

```
by (ALLGOALS (fast_tac (claset()
        addIs (typing.intrs @ typeof.intrs @ [typing_subst_0])
        addEs (typingEs @ typeofEs @ [weakening_0])
        addss simpset())));
```

Let us examine this code. We supply a *classical reasoning set* as argument—this details the rules which `fast_tac` may use. In this case it is the default `claset()` augmented with the introduction and elimination rules for typing, together with weakening and typing substitution lemmas. The latter are required for the subgoal

$$\bigwedge e_1\, e_2\, t\, ve.\ [\![\ e_1 \Downarrow \lambda(t),\ \forall \sigma. e_1 : \sigma \to \lambda(t) : \sigma,\\ t[e_2/0] \Downarrow ve,\ \forall \sigma. t[e_2/0] : \sigma \to ve : \sigma\ ]\!]\\ \Longrightarrow \forall \sigma. e_1\, e_2 : \sigma \to ve : \sigma$$

which arises from the third rule of Figure 1. The resulting tactic is applied to all goals and solves them completely, which finishes the proof.

This example exhibits a good style of mechanical proof. It follows closely the structure of the corresponding pen-and-paper proof and uses automation effectively to perform the case analysis via a single tactic applied uniformly. It is unfortunate, but few of our proofs were so easy. Many involved a large case analysis where the individual cases were more complex. These had to be proved one-by-one using, sometimes, rather ad-hoc methods. Here mechanical assistance gave less support.

4.2 Determinism of Evaluation

Another example of an elegant proof is that of the determinism of evaluation. We show that

$$p \Downarrow u \Longrightarrow \forall v.\, p \Downarrow v \to v = u$$

by induction, again on the derivation of $p \Downarrow u$. The subgoals are even simpler this time. The proof in each case uses just one application of an elimination rule for *eval* and then elementary logical reasoning. The only complication is that the last subgoal must be handled separately. There are two possible ways to apply the elimination rule for the evaluation of an application, so we instantiate the rule to force the correct choice. The induction rule generates seventeen subgoals. The first two are trivial, but, for each of the others, applying the elimination rule for evaluation of a function application, we generate a further fifteen subgoals. There are thus a total of 227 subgoals in the proof—most are dismissed immediately!

4.3 Special Tactics

Most of the automation in the proofs has been supplied by standard search tactics such as `fast_tac`, `depth_tac` and `blast_tac` or simplification tactics such as `simp_tac`, `asm_simp_tac` and `asm_full_simp_tac`. There is one special purpose tactic which deserves mention. The rules (like those) in Figure 1 define the evaluation relation. However if we try to prove a goal of the form $e \Downarrow ?v$ where $?v$ is an uninstantiated meta-variable then backwards proof search using these rules is non-deterministic. But the evaluation strategy that they represent is deterministic. The solution is to write a special purpose tactic for evaluation based on a series of derived rules. For example, a conditional would be evaluated via the rules

$$\frac{e \Downarrow ve \quad ve\, e' \Downarrow ve'}{e\, e' \Downarrow ve'} \qquad \frac{e \Downarrow ve \quad \mathsf{cond}\, ve \Downarrow ve'}{\mathsf{cond}\, e \Downarrow ve'}$$

$$\frac{}{\mathsf{cond}\,\mathsf{true} \Downarrow \lambda(\lambda(1))} \qquad \frac{}{\mathsf{cond}\,\mathsf{false} \Downarrow \lambda(\lambda(0))}$$

Resolving against these rules in order, we ensure that the function expression is evaluated first (to the constant cond), then its argument (to either true or false), and finally the application is evaluated appropriately. The function `eval_tac` encodes this deterministic evaluation strategy for an arbitrary expression.

5 Bisimilarity for *PL*

We can now define a notion of program similarity for *PL*, coded as the Isabelle theory `Sim.thy`. Bisimilarity [Gor95b] is the symmetrization of similarity; this is a useful notion of program equivalence, as described in on page 1. This has been studied by many people (for example [CG99,Gor95a,Pit97]). Our definition of similarity is equivalent to Pitts' [Pit97], but is slightly simpler for the purposes of mechanized proof—see Section 5.2. The definition of simulation is given as a type indexed family of binary relations between closed expressions of the same type: $\preccurlyeq = (\preccurlyeq_\sigma \mid \sigma) \in \Pi_\sigma \mathcal{P}(\mathcal{E}_\sigma \times \mathcal{E}_\sigma)$. The definition of simulation is the coinductively defined set given by a certain monotone function of the form $\psi: \Pi_\sigma \mathcal{P}(\mathcal{E}_\sigma \times \mathcal{E}_\sigma) \to \Pi_\sigma \mathcal{P}(\mathcal{E}_\sigma \times \mathcal{E}_\sigma)$. Given any $R \in \Pi_\sigma \mathcal{P}(\mathcal{E}_\sigma \times \mathcal{E}_\sigma)$, one can define $\psi(R)$ by specifying [Pit97] the components $\psi(R)_\sigma$ at each type σ; if we consider the type *bool*, our *simBool* (see below) corresponds to $\psi(R)_{bool}$. The function ψ then corresponds to our θ_{sim}, and $e \preccurlyeq e'$ corresponds to our $(e, e') \in sim$. Let us define *simBool*, and similar sets at the remaining types.

- $simBool \subseteq \mathcal{E} \times \mathcal{E}$ where $(e, e') \in simBool$ if and only if

$$(\neg \exists ve.\ e \Downarrow ve) \vee (e \Downarrow \mathsf{true} \wedge e' \Downarrow \mathsf{true}) \vee (e \Downarrow \mathsf{false} \wedge e' \Downarrow \mathsf{false})$$

- $simFun: \mathcal{T} \times \mathcal{T} \times \mathcal{P}(\mathcal{E} \times \mathcal{E} \times \mathcal{T}) \to \mathcal{P}(\mathcal{E} \times \mathcal{E})$ where $(e, e') \in simFun(\sigma, \tau, R)$ if and only if

$$\begin{aligned}(\neg \exists ve.\ e \Downarrow ve) \vee (\exists ve, ve'.\ & e \Downarrow ve \wedge e' \Downarrow ve' \\ & \wedge \forall p. p : \sigma \Longrightarrow (ve\, p, ve'\, p, \tau) \in R)\end{aligned}$$

- $simNat: \mathcal{P}(\mathcal{E} \times \mathcal{E} \times \mathcal{T}) \to \mathcal{P}(\mathcal{E} \times \mathcal{E})$ where $(e, e') \in simNat(R)$ if and only if

$$\begin{aligned}&(\neg \exists ve.\ e \Downarrow ve) \vee (e \Downarrow \mathsf{zero} \wedge e' \Downarrow \mathsf{zero}) \\ &\qquad \vee (\exists e'', e'''.\ e \Downarrow \mathsf{suc}\, e'' \wedge e' \Downarrow \mathsf{suc}\, e''' \wedge (e'', e''', nat) \in R)\end{aligned}$$

- $simPair: \mathcal{T} \times \mathcal{T} \times \mathcal{P}(\mathcal{E} \times \mathcal{E} \times \mathcal{T}) \to \mathcal{P}(\mathcal{E} \times \mathcal{E})$ where $(e, e') \in simPair(\sigma, \tau, R)$ if and only if

$$\begin{aligned}(\neg \exists ve.\ e \Downarrow ve) \vee (\exists e_1, e_2, e_1', e_2'.\ & e \Downarrow \mathsf{pr}\, e_1\, e_2 \wedge e' \Downarrow \mathsf{pr}\, e_1'\, e_2' \\ & \wedge (e_1, e_1', \sigma) \in R \wedge (e_2, e_2', \tau) \in R)\end{aligned}$$

- $simList: \mathcal{T} \times \mathcal{P}(\mathcal{E} \times \mathcal{E} \times \mathcal{T}) \to \mathcal{P}(\mathcal{E} \times \mathcal{E})$ where $(e, e') \in simList(\sigma, R)$ if and only if

$$\begin{aligned}&(\neg \exists ve.\ e \Downarrow ve) \vee (e \Downarrow \mathsf{nil} \wedge e' \Downarrow \mathsf{nil}) \\ &\qquad \vee (\exists e_1, e_2, e_1', e_2'.\ e \Downarrow \mathsf{cons}\, e_1\, e_2 \wedge e' \Downarrow \mathsf{cons}\, e_1'\, e_2' \\ &\qquad\qquad \wedge (e_1, e_1', \sigma) \in R \wedge (e_2, e_2', list(\sigma)) \in R)\end{aligned}$$

We then define the set of *simulations*, $sim \subseteq \mathcal{E} \times \mathcal{E} \times \mathcal{T}$, using **C-Set** by taking

$$\theta_{sim}: \mathcal{P}(\mathcal{E} \times \mathcal{E} \times \mathcal{T}) \to \mathcal{P}(\mathcal{E} \times \mathcal{E} \times \mathcal{T})$$

where (recall page 3) $z \in \theta_{sim}(X)$ if and only if

$$\begin{aligned}&\exists e, e'.\ z = (e, e', bool) \wedge e : bool \wedge e' : bool \wedge (e, e') \in simBool \\ &\vee \exists e, e'.\ z = (e, e', nat) \wedge e : nat \wedge e' : nat \wedge (e, e') \in simNat \\ &\vee \exists e, e', \sigma, \tau.\ z = (e, e', \sigma \to \tau) \\ &\qquad\qquad \wedge e : \sigma \to \tau \wedge e' : \sigma \to \tau \wedge (e, e') \in simFun(\sigma, \tau, X) \\ &\vee \exists e, e', \sigma, \tau.\ z = (e, e', \sigma * \tau) \wedge e : \sigma * \tau \wedge e' : \sigma * \tau \wedge (e, e') \in simPair(\sigma, \tau, X) \\ &\vee \exists e, e', \sigma.\ z = (e, e', list(\sigma)) \wedge e : list(\sigma) \wedge e' : list(\sigma) \wedge (e, e') \in simList(\sigma, X)\end{aligned}$$

This formula can be implemented in Isabelle-HOL using the (co)-induction package—see Section 5.2.

We next define open similarity [Pit97], given by judgements of the form $\Gamma \vdash e \preccurlyeq^o e' : \sigma$. In order to define these judgements mechanically, we shall define the set *osim* for which $(\Gamma, e, e', \sigma) \in osim$ corresponds to such a judgement.

First we define the function $osimrel : \mathcal{G} \times \mathcal{T} \times \mathcal{T} \times \mathcal{P}(\mathcal{G} \times \mathcal{E} \times \mathcal{E} \times \mathcal{T}) \to \mathcal{P}(\mathcal{E} \times \mathcal{E})$ where $(e, e') \in osimrel(\Gamma, \sigma, \tau, R)$ if and only if

$$\tau, \Gamma \vdash e : \sigma \wedge \tau, \Gamma \vdash e' : \sigma \wedge \forall p.(p : \tau \Longrightarrow (\Gamma, e[p/0], e'[p/0], \tau) \in R)$$

We can then define $osim \subset \mathcal{G} \times \mathcal{E} \times \mathcal{E} \times \mathcal{T}$ using **C-Set**, which can also be implemented directly. Set

$$\theta_{osim} : \mathcal{P}(\mathcal{G} \times \mathcal{E} \times \mathcal{E} \times \mathcal{T}) \to \mathcal{P}(\mathcal{G} \times \mathcal{E} \times \mathcal{E} \times \mathcal{T})$$

where $z \in \theta_{osim}(X)$ if and only if

$$\begin{aligned}&\exists \sigma, e, e'.\ z = ([\,], e, e', \sigma) \wedge (e, e', \sigma) \in sim \\ &\vee \exists \sigma, \tau, \Gamma, e, e'.\ z = (\mathtt{newVar}\, 0\, \Gamma\, \tau\ , e, e', \sigma) \in X \wedge (e, e') \in osimrel(\Gamma, \sigma, \tau, X)\end{aligned}$$

5.1 Extending Weakening and Substitution

We use Isabelle to prove variants of the syntax manipulating theorems of Section 3.1 for the *PL* judgements which take the general form $\Gamma \vdash e\ R\ e' : \sigma$. For example, in the encoding of *osim*, we show that

$$\begin{aligned}&[\!|\ \Gamma \vdash e \preccurlyeq^o e' : \sigma,\ j \leq \mathtt{length}(\Gamma)\ |\!] \\ &\qquad \Longrightarrow \mathtt{newVar}\, j\, \Gamma\, \tau\ \vdash \mathtt{lift}\, e\ j \preccurlyeq^o \mathtt{lift}\, e'\ j : \sigma\end{aligned}$$

$$\begin{aligned}&[\!|\ \mathtt{newVar}\, j\, \Gamma\, \tau\ \vdash e \preccurlyeq^o e' : \sigma,\ j \leq \mathtt{length}(\Gamma),\ \Gamma \vdash p : \tau\ |\!] \\ &\qquad \Longrightarrow \Gamma \vdash e[p/j] \preccurlyeq^o e'[p/j] : \sigma\end{aligned}$$

These results are not trivial. The latter is established in three steps by first proving the result in the special case where $p : \tau$, and then in the case where j is zero, before proving the general result. Each of these proofs proceeds by induction on the structure of the list Γ.

5.2 Mechanizing Simulations

The definition of the relations *sim* and *osim* uses auxiliary functions such as *simBool* and *simFun*. Originally, our definitions of these functions matched those of Pitts [Pit97]. It was then necessary to mechanically verify their monotonicity. However, our current formulation fits more naturally with the Isabelle implementation of rule coinduction which ensures that no direct proof of monotonicity is required. (We have, for completeness, automatically checked the equivalence of the presentations.) It is convenient to use the definition of *sim* to produce special elimination rules—such rules ensure that future proofs are well structured and greatly simplified. The elimination rule below was produced by `mk_cases`, a standard Isabelle procedure; it allows the definition of simulation at function types to be unwrapped.

$$
\begin{array}{l}
[\mid\ e \precsim e' : \sigma \to \tau, \\
\quad \forall ve\ ve'.\ [\mid e \Downarrow ve,\ e : \sigma \to \tau, \\
\qquad\qquad e' \Downarrow ve',\ e' : \sigma \to \tau, \\
\qquad\qquad \forall p. p : \sigma \to ve\, p \precsim ve'\, p : \tau \mid] \Longrightarrow Q, \\
\quad [\mid e : \sigma \to \tau,\ e' : \sigma \to \tau,\ \forall ve''. \neg(e \Downarrow ve'') \mid] \Longrightarrow Q \quad \mid] \Longrightarrow Q
\end{array}
$$

There are similar rules for the other types. Well chosen elimination rules are essential for directing automatic proof procedures such as `fast_tac` and we have used the `mk_cases` function to generate these for most of the (co)inductive definitions in the implementation.

6 The Candidate Relation for *PL*

The theory `Cand.thy` defines an auxiliary relation between expressions with judgements of the form $\Gamma \vdash e \precsim^\bullet e' : \sigma$. This relation is a precongruence, which, informally, means that we can reason about instances of the relation using simple algebraic manipulations. Further, we use Isabelle to prove that $\Gamma \vdash e \precsim^\bullet e' : \sigma$ if and only if $\Gamma \vdash e \precsim^o e' : \sigma$ (a proof methodology was first given by [How89]). Hence $\Gamma \vdash e \precsim^o e' : \sigma$ is a precongruence, and thus we can also reason about similarity (and bisimilarity) using simple algebraic manipulations. The precongruence property is also used critically in other proofs.

In order to define the candidate relation mechanically, we shall define the set *cand* for which $(\Gamma, e, e', \sigma) \in cand$ corresponds to such a judgement. We first define the function

$$compat\colon \mathcal{P}(\mathcal{G} \times \mathcal{E} \times \mathcal{E} \times \mathcal{T}) \to \mathcal{P}(\mathcal{G} \times \mathcal{E} \times \mathcal{E} \times \mathcal{T})$$

using **I-Function**. We define

$$\theta_{compat;R}\colon \mathcal{P}(\mathcal{G} \times \mathcal{E} \times \mathcal{E} \times \mathcal{T}) \to \mathcal{P}(\mathcal{G} \times \mathcal{E} \times \mathcal{E} \times \mathcal{T})$$

where $z \in \theta_{compat;R}(X)$ if and only if

$$
\begin{array}{l}
\exists \Gamma, k, \sigma.\ z = (\Gamma, k, k, \sigma) \wedge k : \sigma \\
\vee\ \exists \Gamma, e_1, e_1', e_2, e_2', \tau, \sigma.\ z = (\Gamma, e_1\, e_2, e_1'\, e_2', \tau) \wedge \\
\qquad\qquad\qquad\qquad (\Gamma, e_1, e_1', \tau \to \sigma) \in R \wedge (\Gamma, e_2, e_2', \sigma) \in R \\
\vee \ldots \textit{ further clauses for other types}
\end{array}
$$

The idea is that the clauses defining $\theta_{compat;R}$ capture precisely the properties of a precongruence. We can use them to define [Pit97] $cand \subseteq \mathcal{G} \times \mathcal{E} \times \mathcal{E} \times \mathcal{T}$ using **I-Set**. We define $\theta_{cand}\colon \mathcal{G} \times \mathcal{E} \times \mathcal{E} \times \mathcal{T} \to \mathcal{G} \times \mathcal{E} \times \mathcal{E} \times \mathcal{T}$ where $z \in \theta_{cand}(X)$ if and only if

$$\exists \Gamma, e, e', e'', \sigma.\ z = (\Gamma, e, e'', \sigma) \wedge (\Gamma, e, e', \sigma) \in compat(X) \wedge (\Gamma, e', e'', \sigma) \in osim$$

6.1 Mechanization of Candidate Order

The candidate order yields an example where proofs are more difficult to automate. Many sublemmas arise in proofs. Consider, for instance, the sublemma which states that the candidate relation is substitutive.

$$[\!|\ \texttt{newVar}\, j\, \Gamma\, \tau\ \vdash e_1 \preccurlyeq^{\bullet} e_1' : \sigma,\ j \leq \texttt{length}(\Gamma),\ \Gamma \vdash e_2 \preccurlyeq^{\bullet} e_2' : \tau\ |\!]$$
$$\Longrightarrow \Gamma \vdash e_1[e_2/j] \preccurlyeq^{\bullet} e_1'[e_2'/j] : \sigma$$

Gordon [Gor95b] states that the proof "follows by routine rule induction ...". The corresponding machine proof has 77 steps. The longest example is the proof that the candidate relation is closed under evaluation,

$$[\!|\ e \Downarrow v,\ [\,] \vdash e \preccurlyeq^{\bullet} e' : \sigma\ |\!] \Longrightarrow [\,] \vdash v \preccurlyeq^{\bullet} e' : \sigma$$

which is completed in 254 steps.

Theorem 1. *We have used Isabelle to prove mechanically that the candidate relation coincides with similarity, that is cand = osim. More informatively*

$$\Gamma \vdash e \preccurlyeq^{\bullet} e' : \sigma \Longleftrightarrow \Gamma \vdash e \preccurlyeq^{o} e' : \sigma$$

6.2 Comments on the Proof

That *osim* is contained in *cand* follows from the definition of *cand* and that the compatible refinement of *cand* is reflexive. The other direction is more difficult. We show that restriction of the candidate relation to closed terms is a simulation, and hence by coinduction,

$$[\,] \vdash e \preccurlyeq^{\bullet} e' : \sigma \Longrightarrow e \preccurlyeq e' : \sigma$$

This follows via a series of lemmas which combine to show that *cand* has the properties required in the definition of *sim*. For example, in the case of natural numbers, we have

$$[\,] \vdash \mathsf{zero} \preccurlyeq^{\bullet} n : nat \Longrightarrow n \Downarrow \mathsf{zero}$$

and

$$[\,] \vdash \mathsf{suc}\, m \preccurlyeq^{\bullet} n : nat \Longrightarrow \exists k.\ n \Downarrow \mathsf{suc}\, k \wedge [\,] \vdash m \preccurlyeq^{\bullet} k : nat$$

These two results together with the downward closure of *cand* establish that *cand* has the requisite property of a simulation on expressions of type *nat*. The other cases follow a similar pattern. Once we have shown that the restriction *cand* is contained in *sim*, then this can be extended to open terms via

$$\Gamma \vdash e \preccurlyeq^{\bullet} e' : \sigma \Longrightarrow \Gamma \vdash e \preccurlyeq^{o} e' : \sigma$$

because the relation *cand* is substitutive (as we proved in Section 6.1).

7 Divergence for *PL*

The theory `Diverges.thy` defines program divergence (looping). It formalizes the judgement $e \Uparrow$ whose meaning is that the expression e does not converge to a result. Note that divergence is also used in our proofs involving contextual equivalence; in particular, note that the clauses used in the definition of similarity on page 8 are given using instances of the divergence predicate.

We define the set $divg \subseteq \mathcal{E}$ using **C-Set** by taking $\theta_{divg}: \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$ where $z \in \theta_{divg}(X)$ if and only if

$$
\begin{aligned}
&(\exists e, e', e''.\ z = e\,e' \wedge e \Downarrow \lambda(e'') \wedge e''[e''/0] \in X) \vee (\exists e, e'.\ z = e\,e' \wedge e \in X) \\
&\vee \ldots\ \textit{further clauses for other types}
\end{aligned}
$$

This coinductive definition of divergence can be shown to correspond to the usual definition given in terms of an infinite sequence of program transitions arising from a single-step operational semantics. We can use Isabelle to establish instances of program divergence, and this is important in reasoning about contextual equivalence.

8 Contextual Equivalence for *PL*

The theory `Context.thy` defines a notion of program context with judgements of the form $(\Gamma, \rho, \Gamma') \Vdash \mathcal{C} : \sigma$. The intuitive idea of a context $\mathcal{C}$ is that it is a program module containing a parameter $-^{\rho}_{par}$ which indicates a position at which another program may be inserted to yield a new program. The meaning of the judgement $(\Gamma, \rho, \Gamma') \Vdash \mathcal{C} : \sigma$ is that an expression e with typing $\Gamma \vdash e : \rho$ may be inserted into $\mathcal{C}$ to yield an expression $(\mathcal{C}\, e)$ with typing $\Gamma' \vdash (\mathcal{C}\, e) : \sigma$. The theory `Cord.thy` defines a notion of contextual ordering with judgements of the form $\Gamma \vdash e \leq e' : \sigma$. The intended meaning is that if the evaluation of $(\mathcal{C}\, e)$ (denoting the context $\mathcal{C}$ in which e replaces the parameter) results in ve, then so too does the evaluation of $(\mathcal{C}\, e')$. Contextual equivalence is the symmetrization of contextual ordering. We use Isabelle to prove that $\Gamma \vdash e \leq e' : \sigma$ if and only if $\Gamma \vdash e \preccurlyeq^{o} e' : \sigma$. Thus we can establish instances of contextual equivalence by instead proving bisimilarity.

We wish to define judgements of the form [CG99,Gor95a,Pit97] $\Gamma \vdash e \leq e' : \sigma$, or equivalently a subset of the form $cord \subset \mathcal{G} \times \mathcal{E} \times \mathcal{E} \times \mathcal{T}$. Given a set X, let $X \Rightarrow X$ denote the set of endofunctions on X; program contexts $\mathcal{C}$ are of Isabelle type $\mathcal{E} \Rightarrow \mathcal{E}$ and we omit their definition. We first define

$$
ctyping \subseteq (\mathcal{G} \times \mathcal{T} \times \mathcal{G}) \times (\mathcal{E} \Rightarrow \mathcal{E}) \times \mathcal{T}
$$

using **I-Set** by taking

$$
\theta_{ctyping}: \mathcal{P}((\mathcal{G} \times \mathcal{T} \times \mathcal{G}) \times (\mathcal{E} \Rightarrow \mathcal{E}) \times \mathcal{T}) \to \mathcal{P}((\mathcal{G} \times \mathcal{T} \times \mathcal{G}) \times (\mathcal{E} \Rightarrow \mathcal{E}) \times \mathcal{T})
$$

where $z \in \theta_{ctyping}(X)$ if and only if

$$
\begin{aligned}
&\exists \Gamma, \rho. z = (([\,], \rho, \Gamma), id, \rho) \\
&\vee\ \exists \sigma, \tau, \Gamma, \Gamma', \rho, e, e'. z = ((\Gamma, \rho, \Gamma'), \xi \mapsto (e\,\xi)\, e', \tau) \\
&\qquad\qquad\qquad\qquad\qquad \wedge\ ((\Gamma, \rho, \Gamma'), e, \sigma \to \tau) \in X \wedge \Gamma' \vdash e' : \sigma)) \\
&\vee\ \exists \sigma, \tau, \Gamma, \Gamma', \rho, e, e'. z = ((\Gamma, \rho, \Gamma'), \xi \mapsto e\,(e'\,\xi), \tau) \\
&\qquad\qquad\qquad\qquad\qquad \wedge\ \Gamma' \vdash e : \sigma \to \tau \wedge ((\Gamma, \rho, \Gamma'), e', \sigma) \in X \\
&\vee \ldots \textit{ further clauses for other types}
\end{aligned}
$$

We define the *contextual ordering* to be the set $cord \subset \mathcal{G} \times \mathcal{E} \times \mathcal{E} \times \mathcal{T}$ by $(\Gamma, e, e', \sigma) \in cord$ if and only if

$$
\begin{aligned}
&\Gamma \vdash e : \sigma \wedge \Gamma \vdash e' : \sigma \wedge \forall \mathcal{C}, \tau. ((\Gamma, \sigma, [\,]), \mathcal{C}, \tau) \in ctyping \\
&\qquad\qquad\qquad\qquad \to \exists ve.\ (\mathcal{C}\, e) \Downarrow ve \to \exists ve.\ (\mathcal{C}\, e') \Downarrow ve
\end{aligned}
$$

Theorem 2. *We have used Isabelle to prove that the contextual preorder coincides with similarity, that is cord = osim. More informatively,*

$$\Gamma \vdash e \leq e' : \sigma \Longleftrightarrow \Gamma \vdash e \preccurlyeq^{o} e' : \sigma$$

8.1 Comments on the Proof

The main consequence of Theorem 1 is to show that *osim* is a precongruence, since *cand* is easily proved to be precongruence. It follows that if $\Gamma, \Gamma' \vdash e \preccurlyeq^{o} e' : \rho$ and $\mathcal{C}$ is any context then we can substitute e and e' into the context to get $\Gamma' \vdash (\mathcal{C}\, e) \preccurlyeq^{o} (\mathcal{C}\, e') : \sigma$. More precisely, we have

$$[|\ (\Gamma, \rho, \Gamma') \Vdash \mathcal{C} : \sigma,\ \Gamma, \Gamma' \vdash e \preccurlyeq^{o} e' : \rho\ |] \Longrightarrow \Gamma' \vdash (\mathcal{C}\, e) \preccurlyeq^{o} (\mathcal{C}\, e') : \sigma$$

We deduce that *osim* is contained in *cord*. The other direction is more difficult. Much as in the proof of Theorem 1, we show that the restriction of *cord* to closed terms is a simulation, and hence by coinduction, $[\,] \vdash e \leq e' : \sigma \Longrightarrow e \preccurlyeq e' : \sigma$. We make use of a series of lemmas to characterize the evaluation of an expression to a particular value in terms of a convergence predicate, for example, for the natural numbers,

$$m : nat \Longrightarrow \exists n.\ m \Downarrow \mathsf{suc}\, n \to \mathsf{ncase}\, m \perp \lambda(\mathsf{zero}) \Downarrow$$

The proof is quite long. There are 144 steps to establish all of the cases. Once it is done then it is not too difficult to establish that *cord* is substitutive, and so

$$\Gamma \vdash e \leq e' : \sigma \Longrightarrow \Gamma \vdash e \preccurlyeq^{o} e' : \sigma$$

9 Conclusions and Further Research

We have successfully mechanized a core programming language, and used the encoding to verify some fundamental language properties. This has been executed with a direct and uniform methodology, much of which can be smoothly refined to deal with larger languages. We are currently undertaking a similar programme to the one described in this paper for the second order lambda calculus with fixed point recursion. Much of the infrastructure we have built up for the simple language *PL* can in fact be modified quite easily to deal with this more powerful system. We will prove once again that bisimilarity and contextual equivalence coincide (though when we move to other languages, we will only be able to show that contextual equivalence is implied by bisimilarity). We are also extending this work to procedural and object oriented languages; in particular we hope to demonstrate that many of the Isabelle tactics developed in the work reported here can be used *generically*, or at least with minor modifications, within new languages. The semantics of *PL* is presented using a SOS evaluation relation. We will also be studying other semantics including low level abstract machines. In particular, we aim to automate proofs of correspondences between high and low level semantic specifications. Each verification will increase confidence that the lower level implementation correctly encodes the higher level. The long term aim is to build a front end which will enable language designers to use such a verification system with little knowledge of Isabelle. This is crucial if such tools are to be used in real language design. We thank Andy Gordon (Microsoft) and Andy Pitts (Cambridge) who have made useful comments on our preliminary work.

References

[Abr90] S. Abramsky. The lazy lambda calculus. In David Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects.* Monographs in Computer Science. Springer-Verlag, 1996.

[Alt93] T. Altenkirch. A formalization of the strong normalization proof for system F in LEGO. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, pages 13–28. Springer-Verlag, 1993.

[Bie97] G. M. Bierman. Observations on a Linear PCF (Preliminary Report). Technical Report 412, University of Cambridge Computer Laboratory, 1997.

[BS98] E. Börger and W. Schulte. A Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, 1998. To appear.

[CG94] G. Collins and S. Gilmore. Supporting Formal Reasoning about Standard ML. Technical Report ECS-LFCS-94-310, Laboratory for the Foundations of Computer Science, University of Edinburgh, November 1994.

[CG99] R. L. Crole and A. D. Gordon. Relating Operational and Denotational Semantics for Input/Output Effects. *Mathematical Structures in Computer Science*, 9:1–34, 1999.

[Coq92] C. Coquand. A proof normalization for simply typed lambda calculus written in ALF. In K. Petersson B. Nordström and G. Plotkin, editors, *Proc. of the 1992 Workshop on Types for Proofs and Programs*, 1992.

[Cro96] R. L. Crole. How Linear is Howe? In G. McCusker, A. Edalat, and S. Jourdan, editors, *Advances in Theory and Formal Methods 1996*, pages 60–72. Imperial College Press, 1996.

[Cro97] R. L. Crole. The KOREL Programming Language (Preliminary Report). Technical Report 1997/43, Department of Mathematics and Computer Science, University of Leicester, 1997.

[Cro98] R. L. Crole. Lectures on [Co]Induction and [Co]Algebras. Technical Report 1998/12, Department of Mathematics and Computer Science, University of Leicester, 1998.

[dB72] N. de Bruijn. Lambda Calculus Notation with Nameless Dummies: a Tool for Automatic Formula Manipulation, with Application to the Church Rosser Theorem. *Indagationes Mathematicae*, 34:381–391, 1972.

[FPT99] M. Fiore, G. D. Plotkin, and D. Turi. Abstract Syntax and Variable Binding. To appear in LICS 99, 1999.

[GHL97] A. D. Gordon, P. D. Hankin, and S. B. Lassen. Compilation and Equivalence of Imperative Objects. Draft manuscript, 1997.

[GM96] A. D. Gordon and T. Melham. Five axioms of alpha-conversion. In J. von Wright, J. Grundy, and J. Harrison, editors, *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *Lecture Notes in Computer Science*, pages 173–190, Turku, Finland, August 1996. Springer-Verlag.

[Gor95a] A. D. Gordon. Bisimilarity as a theory of functional programming. *Electronic Notes in Theoretical Computer Science*, 1, 1995.

[Gor95b] A. D. Gordon. Bisimilarity as a theory of functional programming. Technical report, Aarhus University, Denmark, 1995. BRICS Notes Series NS–95–3, BRICS, Aarhus University.

[GP99] M. J. Gabbay and A. M. Pitts. A New Approach to Abstract Syntax Involving Binders. To appear in LICS 99, 1999.

[Hir97] D. Hirschkoff. A full formalisation of π-calculus theory in the calculus of constructions. In *Proceedings of TPHOL'97*, volume 1275 of *LNCS*. Springer-Verlag, 1997.

[How89] D. J. Howe. Equality in lazy computation systems. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, pages 198–203, 1989.

[Hue94] G. Huet. Residual theory in λ-calculus: a complete Gallina development. *Journal of Functional Programming*, 4(3):371–394, 1994.

[JDH95] A. Felty J. Despeyroux and A. Hirschowitz. Higher order syntax in Coq. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Typed Lambda Calculi and Applications*, volume 902 of *LNCS*. Springer-Verlag, 1995.

[JDS97] F. Pfenning J. Despeyroux and C. Schüermann. Primitive recursion for higher-order abstract syntax. In *Proceedings of the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)*, volume LNCS. Springer-Verlag, 1997.

[Las98] S. B. Lassen. *Relational Reasoning about Functions and Nondeterminism*. PhD thesis, Dept of Computer Science, Univ of Aarhus, 1998.

[MG93] T. F. Melham and M. J. C. Gordon. *Introduction to HOL*. Cambridge University Press, 1993.

[MG94] S. Maharaj and E. Gunter. Studying the ML module system in HOL. In T. F. Melham and J. Camilleri, editors, *Proceedings of the 7th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, volume 859 of *Lecture Notes in Computer Science*, pages 346–361, Valletta, Malta, September 1994. Springer-Verlag.

[MMH97] R. Milner, M.Tofte, and R. Harper. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Mass., 1997.

[MT91] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 1991.

[Nip] T. Nipkow. More Church Rosser Proofs. To appear in the Journal of Automated Reasoning.

[Nip98] T. Nipkow. Winskel is (Amost) Right: Towards a Mechanized Semantics Textbook. *Formal Aspects of Computing*, 10, 1998.

[NvO98] T. Nipkow and D. von Oheimb. Machine-Checking the Java Specification: Proving Type Safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, 1998. To appear.

[Pau94a] L. C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In A. Bundy, editor, *12th International Conf. on Automated Deduction*, volume 814 of *LNAI*, pages 148–161. Springer-Verlag, 1994.

[Pau94b] L.C. Paulson. Isabelle: A Generic Theorem Prover. *Lecture Notes in Computer Science*, 828, 1994.

[Pit97] A. M. Pitts. Operationally Based Theories of Program Equivalence. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation*, 1997.

[Pit98] A. M. Pitts. Existential Types: Logical Relations and Operational Equivalence. 1998.

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

[Sym93] D. Syme. Reasoning with the formal definition of Standard ML in HOL. In J. J. Joyce and C.-J. H. Seger, editors, *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications (HUG'93)*, volume 780 of *Lecture Notes in Computer Science*, pages 43–60, Vancouver, B.C., Canada, August 1993. Springer-Verlag, 1994.

[Sym97a] D. Syme. DECLARE: A Prototype Declarative Proof System for Higher Order Logic. Technical Report 416, Computer Laboratory, University of Cambridge, February 1997.

[Sym97b] D. Syme. Proving JavaS Type Soundness. Technical Report 427, Computer Laboratory, University of Cambridge, June 1997.