Mechanized Operational Semantics via (Co)Induction (Preliminary Report)

Simon J. Ambler and Roy L. Crole

January 4, 1999

Abstract

We give a fully automated description of a small programming language in the theorem prover Isabelle-98. The language syntax and semantics are encoded, and we formally verify a range of semantic properties. This is achieved via uniform (co)inductive methods.

Keywords: automated deduction, tactical theorem proving (Isabelle), operational semantics, induction and coinduction, software specification and verification.

Submitted for publication.

1 Introduction

The design of new programming languages which are well-principled, reliable and expressive is an important part of Computer Science. In this paper *we contribute towards the techniques for specification, design and development of programming languages by specifying and verifying properties of a core language, using tactical verification within the theorem prover Isabelle 98. We show how this can be done uniformly, so that our methodology readily adapts to new languages, using suitable variations of standard methods to ensure rapid mechanizations: these are key aspects of our work.*

One framework for giving formalized semantic specifications is the structured operational semantics (SOS) introduced by Plotkin [29]. 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 [24, 24, 9]). 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 [11, 14, 26]). 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 [2] and in fact this can be adapted to give a general notion of program equivalence. Applicative bisimulation often implies contextual equivalence and for some languages the two notions coincide. Howe [17] 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 [26], a functional language with IO [11], the Object-calculus of Abadi and Cardelli [1, 12], and the linear λ-calculus [3, 7]. The papers [20, 22, 27] 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 [23]. Recently, a number of people have made advances in semantic verification. Syme has mechanized a proof of type soundness for a fragment of Java [33] using DECLARE [32]; a similar verification has been made by Nipkow and von Oheimb [25] within Isabelle-HOL. Börger and Schulte [4] have used ASMs to formalize a Java fragment. Collins and others [6, 21, 31] have coded fragments of Standard ML in HOL.

Our contribution is to provide, within Isabelle-HOL, *a fully mechanized account of a small programming language* called *PL*. For us, mechanization means producing theories within Classical Higher Order Logic presented in a natural deduction style. Each constituent part of *PL* is encoded as a theory; for example the type system is represented by `Type.thy`. Each theory gives rise to new HOL definitions, together with introduction and elimination rules. Automated reasoning is achieved (in part) by computer application of these rules until proofs result. 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. Let us give a little more explanation: we have coded the following Isabelle theories which capture precisely the specification and properties of *PL*. We state, underlined below, the main results which are proved mechanically.

The Syntax of *PL*:

The theory `Expr.thy` defines the syntax of programs; each program is either a predefined constant, variable, function definition or application. The theory `Type.thy` defines the types (Booleans, numbers, functions, pairing and lists) and a type assignment system with judgements of the form $\Gamma \vdash e{:}\sigma$ which assert that in the typing environment $\Gamma =$

$v_1{:}\sigma_1, \ldots, v_n{:}\sigma_n$ the expression e has the type σ. A simple example is

$$z{:}\,bool, y{:}\,list(nat) \vdash \lambda x.\mathsf{cons}\,x\,y{:}\,nat \to list(nat)$$

See Section 2.

Operational Evaluation of *PL*:

The theory `Eval.thy` defines an evaluation system with judgements of the form $e \Downarrow ve$ which assert that the program expression e computes to the result ve. An example is $\mathsf{cond}\,\mathsf{true}\,(\mathsf{suc}\,\mathsf{zero})\,\mathsf{zero} \Downarrow \mathsf{suc}\,\mathsf{zero}$—see Section 3. Isabelle proves Subject Reduction, that if $\Gamma \vdash e{:}\sigma$ and $e \Downarrow ve$ then $\Gamma \vdash ve{:}\sigma$ (evaluation preserves typing); and also Determinism of Evaluation, that if $e \Downarrow ve$ and $e \Downarrow ve'$ then $ve = ve'$.

Bisimilarity for *PL*:

The theory `Sim.thy` defines a notion of simulation with judgements of the form $\Gamma \vdash e \preccurlyeq^o e'{:}\sigma$. The idea is that any evaluation step performed by e is matched by e', and the two results simulate each other. Bisimilarity [13] is the symmetrization of similarity; this is a useful notion of program equivalence, as described in on page 1. See also Section 4.

The Candidate Relation for *PL*:

The theory `Cand.thy` defines an auxiliary relation between expressions with judgements of the form $\Gamma \vdash e \preccurlyeq^\bullet e'{:}\sigma$. This relation is easily proved to be a precongruence [26]—loosely this means we can reason about (bi)similarity using simple algebraic manipulations. Further, Isabelle proves that $\Gamma \vdash e \preccurlyeq^\bullet e'{:}\sigma$ if and only if $\Gamma \vdash e \preccurlyeq^o e'{:}\sigma$, and thus we see that $\Gamma \vdash e \preccurlyeq^o e'{:}\sigma$ is a precongruence. Congruence of bisimilarity follows immediately. The congruence property is also critical for further proofs. See Section 5.

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. Divergence is also used in our proofs involving contextual equivalence. See Section 6.

Contextual Equivalence for *PL*:

The theory `Context.thy` defines a notion of program context with judgements of the form $v_1{:}\sigma_1, \ldots, v_n{:}\sigma_n \Vdash \mathcal{C}{:}\sigma$. The intuitive meaning of a context $\mathcal{C}$ is that it is a program module containing a parameter $-^\tau_{par}$ which indicates a position at which another program may be inserted to yield a new program. 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. Isabelle proves 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. See Section 7.

We have achieved a *uniform mechanization* of PL by the use of inductive and coinductive methods. Recall [10] 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 $\nu\,\theta$ 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. **I-Set** (etc) are referenced later on.

(Co)Inductive Sets and Functions

- To 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 define a function $f : \mathcal{P}(U) \to \mathcal{P}(U)$ we define $\theta_{f;X} : \mathcal{P}(U) \to \mathcal{P}(U)$ 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**).
- To encode $\theta : \mathcal{P}(U) \to \mathcal{P}(U)$ in Isabelle, we define a HOL formula $\Psi^{HOL}(z, X)$ such that $\quad \forall z.\, \forall X \in \mathcal{P}(U).\, z \in \theta(X) \iff \Psi^{HOL}(z, X)$

The remainder of this paper proceeds thus: Each section corresponds to one of the titled paragraphs given on page 2. Within each section we describe the mathematical specification of the key ideas, and then give an account of the issues involved in the corresponding mechanization. We finish with a summary and conclusion of our work, along with ideas for future research.

2 The Syntax of PL

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 $\mathcal{T}$. The set Cst contains the *constants* cond, zero, suc, ncase, and rec. These capture conditional tests, numbers and recursion. There are also constants which define pairs and lists. We define a subset $Cty \subseteq Cst \times \mathcal{T}$ which supplies types for each constant, such as $(\mathsf{cond}, bool \to \sigma \to \sigma)$, and (zero, nat); we omit the remaining clauses. The *terms* of PL are given by the grammar $e ::= v \mid k \mid \lambda v.e \mid e\,e$ where v ranges over a set Var of *variables*. The *expressions* of PL are the α-equivalence classes of terms. $\mathcal{E}$ denotes the set of expressions. The set $\mathcal{G}$ consists of lists $[v_1 : \sigma_1, \ldots, v_n : \sigma_n]$ of (variable,type) pairs, each referred to as a *typing environment*. The type assignment relation is generated by the usual rules of λ-calculus, together with $\Gamma \vdash k : \sigma$ whenever $(k, \sigma) \in Cty$. We write $e : \sigma$ for $[\,] \vdash e : \sigma$ and $\mathcal{E}_\sigma \stackrel{\text{def}}{=} \{\, e \mid e : \sigma \,\}$.

2.1 Mechanized Representation of Syntax

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 [15] 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. Syntax expressions in $\mathcal{E}$ are encoded in de Bruijn notation using a simple datatype for a λ-calculus with constants; in particular, *PL* constants k are encoded as `Con k_name`, and variables as `Var i` where the variable index `i` is a natural number. The de Bruijn notation leads to reasoning about bound variables via elementary arithmetic manipulation of the variable indices. One cost is that expressions can be difficult to read. However, this is not a substantial problem as we are usually working with program expressions denoted by Isabelle variables, and the underlying variable manipulations are managed via a small number of Isabelle theorems. We next discuss the key issues of weakening and substitution. Note that `env |- e ::: a` corresponds to $\Gamma \vdash e{:}a$ and also that `[| h1, .. ,hn |] ==> c` a natural deduction of `c` from a finite number of hypotheses.

2.2 Weakening and Substitution

This requires careful coding. The environments Γ of *PL* are encoded as a list of types, the ith entry indicating the type of the ith free variable. The function `splice` inserts an element x into the ith position of a list l. The result of splicing a type into an environment is to add a new variable with that type and index i. 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. Thus, Isabelle proves `typing_lift`

```
[| env |- e ::: a; j <= length env |]

                          ==> splice j env b |- lift e j ::: a
```

where the function `lift` applied to e and 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. Isabelle proves `typing_subst`

```
[| j <= length env; splice j env a |- s ::: b; env |- t ::: a |]

                         ==> env |- s [t / j] ::: b
```

$$\frac{}{k \Downarrow k} \qquad \frac{}{\lambda v.e \Downarrow \lambda v.e} \qquad \frac{e \Downarrow \lambda v.e' \qquad e'\{^{e''}/_v\} \Downarrow ve}{e\,e'' \Downarrow ve}$$

$$\frac{e \Downarrow \mathsf{cond} \quad e' \Downarrow \mathsf{true}}{e\,e' \Downarrow \lambda x.\lambda y.x} \qquad \frac{e \Downarrow \mathsf{cond} \quad e' \Downarrow \mathsf{false}}{e\,e' \Downarrow \lambda x.\lambda y.y}$$

$$\frac{e \Downarrow \mathsf{suc}}{e\,e' \Downarrow \mathsf{suc}\,e'} \qquad \frac{e \Downarrow \mathsf{ncase} \quad e' \Downarrow \mathsf{zero}}{e\,e' \Downarrow \lambda x.\lambda y.x} \qquad \frac{e \Downarrow \mathsf{ncase} \quad e' \Downarrow \mathsf{suc}\,e''}{e\,e' \Downarrow \lambda x.\lambda f.f\,e''} \qquad \frac{e \Downarrow \mathsf{rec} \quad e'\,(\mathsf{rec}\,e') \Downarrow ve}{e\,e' \Downarrow ve}$$

Figure 1: *PL* Evaluation Relation—Function Application, Conditionals, Numbers and Recursion

3 Operational Evaluation

We define *operational evaluation* through judgements of the form $e \Downarrow ve$ which denote a relation between expressions. This 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.

3.1 Subject Reduction

The Isabelle proof of Subject Reduction is very neat.

```
val prems = goal thy "p >> v
                      ==> ! a. ([] |- p ::: a) --> ([] |- v ::: a)";
by (cut_facts_tac prems 1);
be eval.induct 1;
by (ALLGOALS (fast_tac (claset()
                addIs (typing.intrs @ typeof.intrs @ [typing_subst_0])
                addEs (typingEs @ typeofEs @ [weakening_0])
                addss simpset())));  qed_spec_mp "subject_reduction";
```

The first two lines set up the goal (theorem) to be proved. As the formula `p >> v` (which encodes $p \Downarrow v$) appears on the left as an assumption, we reason by seeking an appropriate elimination rule. Now, induction rules provide eliminations, and coinduction rules provide introductions. In fact `eval.induct` is the rule we seek. Its use yields a total of seventeen subgoals, each corresponding to a last step in a derivation of `p >> 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. We supply a *classical reasoning set* as argument—this details the rules which `fast_tac` may use. In this case (see the

code above) 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

```
!!p q t x. [| p >> Lam t; ! a. [] |- p ::: a --> [] |- Lam t ::: a;

   t [q / 0] >> x; ! a. [] |- t [q / 0] ::: a --> [] |- x ::: a |]

                ==> ! a. [] |- p @@ q ::: a --> [] |- x ::: a
```

The resulting tactic is applied to all goals and solves them completely, which finishes the proof.

3.2 Determinism of Evaluation

A similar example is the determinism of evaluation

```
val prems = goal thy "p >> u ==> ! v. p >> v --> v=u";
by (cut_facts_tac prems 1);
be eval.induct 1;

....

qed_spec_mp "eval_determ";
```

Again the proof is by induction on the derivation of `p >> 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 `evalAppE` for function application, we generate a further fifteen subgoals. There are thus a total of 227 subgoals in the proof—most are dismissed immediately!

3.3 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

$$\frac{e \Downarrow ve \quad ve\, e' \Downarrow ve'}{e\, e' \Downarrow ve'} \qquad \frac{e \Downarrow ve \quad \mathsf{cond}\, ve \Downarrow ve'}{\mathsf{cond}\, e \Downarrow ve'} \qquad \frac{}{\mathsf{cond\ true} \Downarrow \lambda x.\lambda y.x} \qquad \frac{}{\mathsf{cond\ false} \Downarrow \lambda x.\lambda y.y}$$

Resolving against these rules in a left to right 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.

4 Bisimilarity for *PL*

We can now define a notion of program similarity for *PL*. This has been studied by many people (for example [11, 14, 26]). Our definition of similarity is equivalent to Pitts' [26], but, crucially, *our presentation greatly simplifies the mechanized proofs—see Section 4.2.* The usual definition of simulation is given as a type indexed family of binary relations between closed expressions of the same type: $\preccurlyeq = (\preccurlyeq_\sigma |\ \sigma) \in \Pi_\sigma \mathcal{P}(\mathcal{E}_\sigma \times \mathcal{E}_\sigma)$. 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 [26] 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. Each is specified using a HOL formula; $\bigwedge$ is logical and, $\bigvee$ is or.

- $simBool \subseteq \mathcal{E} \times \mathcal{E}$ where $(e, e') \in simBool$ if and only if
  $(\neg\exists ve.\ e \Downarrow ve) \bigvee (e \Downarrow \mathsf{true} \bigwedge e' \Downarrow \mathsf{true}) \bigvee (e \Downarrow \mathsf{false} \bigwedge e' \Downarrow \mathsf{false})$
- $simFun{:}\,\mathcal{T} \times \mathcal{T} \times \mathcal{P}(\mathcal{E} \times \mathcal{E} \times \mathcal{T}) \to \mathcal{P}(\mathcal{E} \times \mathcal{E})$ where $(e, e') \in simFun(\sigma, \tau, R)$ if and only if
  $(\neg\exists ve.\ e \Downarrow ve) \bigvee (\exists ve, ve'.\ e \Downarrow ve \bigwedge e' \Downarrow ve' \bigwedge \forall p.p{:}\,\sigma \Longrightarrow (ve\, p, ve'\, p, \tau) \in R)$.
- $simNat{:}\,\mathcal{P}(\mathcal{E} \times \mathcal{E} \times \mathcal{T}) \to \mathcal{P}(\mathcal{E} \times \mathcal{E})$ where $(e, e') \in simNat(R)$ if and only if
  $(\neg\exists ve.\ e \Downarrow ve) \bigvee (e \Downarrow \mathsf{zero} \bigwedge e' \Downarrow \mathsf{zero}) \bigwedge$
  $(\exists e'', e'''.\ e \Downarrow \mathsf{suc}\, e'' \bigwedge e' \Downarrow \mathsf{suc}\, e''' \bigwedge (e'', e''', nat) \in R)$
- $simPair{:}\,\mathcal{T} \times \mathcal{T} \times \mathcal{P}(\mathcal{E} \times \mathcal{E} \times \mathcal{T}) \to \mathcal{P}(\mathcal{E} \times \mathcal{E})$ where $(e, e') \in simPair(\sigma, \tau, R)$ if and only if $(\neg\exists ve.\ e \Downarrow ve) \bigvee$
  $(\exists e'_1, e'_2.\ e \Downarrow \mathsf{pr}\, e_1\, e_2 \bigwedge e' \Downarrow \mathsf{pr}\, e'_1\, e'_2 \bigwedge (e_1, e'_1, \sigma) \in R \bigwedge (e_2, e'_2, \tau) \in R)$.

- $simList: \mathcal{T} \times \mathcal{P}(\mathcal{E} \times \mathcal{E} \times \mathcal{T}) \to \mathcal{P}(\mathcal{E} \times \mathcal{E})$ where $(e, e') \in simList(\sigma, R)$ if and only if
  $(\neg\exists ve.\ e \Downarrow ve) \bigvee (e \Downarrow \mathsf{nil} \Longrightarrow e' \Downarrow \mathsf{nil}) \bigwedge$
  $(\exists e_1, e_2, e_1', e_2'.\ e \Downarrow \mathsf{cons}\, e_1\, e_2 \bigwedge e' \Downarrow \mathsf{cons}\, e_1'\, e_2' \bigwedge (e_1, e_1', \sigma) \in R \bigwedge (e_2, e_2', list(\sigma)) \in R)$.

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 4) $z \in \theta_{sim}(X)$ if and only if

$$\begin{array}{l}
\exists e, e'.\ z = (e, e', bool) \bigwedge e: bool \bigwedge e': bool \bigwedge (e, e') \in simBool \bigvee \\
\exists e, e'.\ z = (e, e', nat) \bigwedge e: nat \bigwedge e': nat \bigwedge (e, e') \in simNat \bigvee \\
\exists e, e', \sigma, \tau.\ z = (e, e', \sigma \to \tau) \bigwedge e: \sigma \to \tau \bigwedge e': \sigma \to \tau \bigwedge (e, e') \in simFun(\sigma, \tau, X) \bigvee \\
\exists e, e', \sigma, \tau.\ z = (e, e', \sigma * \tau) \bigwedge e: \sigma * \tau \bigwedge e': \sigma * \tau \bigwedge (e, e') \in simPair(\sigma, \tau, X) \bigvee \\
\exists e, e', \sigma.\ z = (e, e', list(\sigma)) \bigwedge e: list(\sigma) \bigwedge e': list(\sigma) \bigwedge (e, e') \in simList(\sigma, X)
\end{array}$$

This HOL formula can be *directly* implemented in Isabelle—see Section 4.2.

We next define open similarity [26], 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 $osimrel$: $Var \times \mathcal{G} \times \mathcal{T} \times \mathcal{T} \times \mathcal{P}(\mathcal{G} \times \mathcal{E} \times \mathcal{E} \times \mathcal{T}) \to \mathcal{P}(\mathcal{E} \times \mathcal{E})$ where $(e, e') \in osimrel(y, \Gamma, \sigma, \tau, R)$ if and only if

$$y: \tau, \Gamma \vdash e: \sigma \bigwedge y: \tau, \Gamma \vdash e': \sigma \bigwedge \forall p.(p: \tau \Longrightarrow (\Gamma, e\{^p/_y\}, e'\{^p/_y\}, \tau) \in R)$$

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

$$\begin{array}{l}
\exists \sigma, e, e'.\ z = ([\,], e, e', \sigma) \bigwedge (e, e', \sigma) \in sim \bigvee \\
\exists \sigma, \tau, y, \Gamma, e, e'.\ z = (y: \tau, \Gamma, e, e', \sigma) \in X \bigwedge (e, e') \in osimrel(y, \Gamma, \sigma, \tau, X)
\end{array}$$

4.1 Extending Weakening and Substitution

We use Isabelle to prove variants of the syntax manipulating theorems `typing_lift` and `typing_subst` 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

```
opensim_lift [| env |- s <opensim> t ::: a; j <= length env |]

        ==> splice j env b |- lift s j <opensim> lift t j ::: a
```

```
opensim_subst [| splice j env b |- s <opensim> t ::: a;

                 j <= length env; env |- p ::: b |]

                      ==> env |- s [p / j] <opensim> t [p / j] ::: a
```

These results are not trivial. The latter is established in three steps by first proving the result in the special case where `[] |- p ::: b`, 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 `env`.

4.2 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 [26]. 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 following elimination rule was produced by `mk_cases`, a standard Isabelle procedure; it allows the definition of simulation at function types to be unwrapped.

```
[| r <sim> s ::: a -> b;

 !!f g. [| r >> f; [] |- r ::: a -> b; [] |- s ::: a -> b; s >> g;

    ! p. [] |- p ::: a --> f @@ p <sim> g @@ p ::: b |] ==> Q;

 [| [] |- r ::: a -> b; [] |- s ::: a -> b; ~ converges r |]

                                                  ==> Q |] ==> Q
```

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.

5 The Candidate Relation for *PL*

In order to reason about open similarity, we prove that it is a precongruence [26]. We do this by using the technique which originated from Howe [17]. We shall define a relation

which is clearly a precongruence, and show that it corresponds to open similarity. Our method is to use the notion of compatible closure [14], and candidate ordering [26] given by judgements of the form $\Gamma \vdash e \preccurlyeq^\bullet e' : \sigma$. In order to define these judgements mechanically, we shall define the set *cand* for which $(\Gamma, e, e', \sigma) \in \mathit{cand}$ corresponds to such a judgement. We first define the function

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

using **I-Function**. We define

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

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

$\exists \Gamma, k, \sigma.\ z = (\Gamma, k, k, \sigma) \bigwedge k : \sigma \bigvee$
$\exists \Gamma, f, f', a, a', \tau, \sigma.\ z = (\Gamma, f\, a, f'\, a', \tau) \bigwedge (\Gamma, f, f', \tau \to \sigma) \in R \bigwedge (\Gamma, a, a', \sigma) \in R \bigvee$
. . . clauses for other types

The idea is that the clauses defining $\theta_{\mathit{compat};R}$ capture precisely the properties of a precongruence. We can use them to define [26] $\mathit{cand} \subseteq \mathcal{G} \times \mathcal{E} \times \mathcal{E} \times \mathcal{T}$ using **I-Set**. We define $\theta_{\mathit{cand}} : \mathcal{G} \times \mathcal{E} \times \mathcal{E} \times \mathcal{T} \to \mathcal{G} \times \mathcal{E} \times \mathcal{E} \times \mathcal{T}$ where $z \in \theta_{\mathit{cand}}(X)$ if and only if

$$\exists \Gamma, e, e', e'', \sigma . z = (\Gamma, e, e'', \sigma) \bigwedge (\Gamma, e, e', \sigma) \in \mathit{compat}(X) \bigwedge (\Gamma, e', e'', \sigma) \in \mathit{osim}$$

5.1 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 `cand_subst`

```
[| splice j ctxt b |- s <cand> t ::: a;

        j <= length ctxt; ctxt |- p <cand> q ::: b |]

                    ==> ctxt |- s [p / j] <cand> t [q / j] ::: a
```

Gordon [13] states that the proof "follows by routine rule induction . . . ". The corresponding machine proof has 77 steps. The longest example is the proof of `cand_down_closed`, completed in 254 steps:

```
[| s >> v; [] |- s <cand> t ::: a |] ==> [] |- v <cand> t ::: a
```

Theorem

Isabelle proves $\mathit{cand} = \mathit{osim}$, that is $\underline{\Gamma \vdash e \preccurlyeq^\bullet e' : \sigma \iff \Gamma \vdash e \preccurlyeq^o e' : \sigma}$

5.2 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 Isabelle proves `[] |- s <cand> t ::: a ==> s <sim> t ::: a`. 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 `cand_eval_zero`

```
[] |- Con zero <cand> n ::: Nat ==> n >> Con zero
```

and `cand_eval_succ`

```
[] |- Con succ @@ m <cand> n ::: Nat
          ==> ? k. n >> Con succ @@ k & [] |- m <cand> 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

```
cand_opensim ctxt |- s <cand> t ::: a ==> ctxt |- s <opensim> t ::: a
```

because the relation *cand* is substitutive (as we proved in Section 5.1).

6 Divergence for PL

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

$$(\exists e, e', e''.\ z = e\,e' \bigwedge e \Downarrow \lambda x.e'' \bigwedge e''\{^{e'}/_{x}\} \in X) \bigvee (\exists e, e'.\ z = e\,e' \bigwedge e \in X) \bigvee$$
$$\ldots \textit{ clauses for other types}$$

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.

7 Contextual Equivalence for PL

We wish to define judgements of the form [11, 14, 26] $\Gamma \vdash e \leq e' : \sigma$, 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

$\exists \Gamma, \rho. z = (([\,], \rho, \Gamma), id, \rho) \bigvee$
$\exists \sigma, \tau, \Gamma, \Gamma', \rho, e, e'. z = ((\Gamma, \rho, \Gamma'), \xi \mapsto (e\,\xi)\,e', \tau) \bigwedge ((\Gamma, \rho, \Gamma'), e, \sigma \to \tau) \in X \bigwedge \Gamma' \vdash e' : \sigma)) \bigvee$
$\exists \sigma, \tau, \Gamma, \Gamma', \rho, e, e'. z = ((\Gamma, \rho, \Gamma'), \xi \mapsto e\,(e'\,\xi), \tau) \bigwedge \Gamma' \vdash e : \sigma \to \tau \bigwedge ((\Gamma, \rho, \Gamma'), e', \sigma) \in X \bigvee$
... clauses for other types

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

$$\Gamma \vdash e : \sigma \bigwedge \Gamma \vdash e' : \sigma \bigwedge \forall \mathcal{C}, \tau. ((\Gamma, \sigma, [\,]), \mathcal{C}, \tau) \in ctyping$$
$$\Longrightarrow \exists ve.\ (\mathcal{C}\,e) \Downarrow ve \Longrightarrow \exists ve.\ (\mathcal{C}\,e') \Downarrow ve$$

Theorem

Isabelle proves that $cord = osim$, that is $\underline{\Gamma \vdash e \leq e' : \sigma \text{ if and only if } \Gamma \vdash e \preccurlyeq^o e' : \sigma}$

7.1 Comments on the Proof

The main consequence of the Theorem in 5.1 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, in Isabelle we have `ctyping_opensim_lemma`

```
[| (env1, a, env2) |-- c ::: b; env1 @ env2 |- s <opensim> t ::: a |]
     ==> env2 |- c s <opensim> c t ::: b
```

where `c` is a context. We deduce that *osim* is contained in *cord*. The other direction is more difficult. Much as in the proof of Theorem 5.1, we show that the restriction of *cord* to closed terms is a simulation, and hence by coinduction, `cord_sim [] |- s <cord> t ::: a ==> s <sim> t ::: a`. 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 `conv_succ_lemma`

```
 [] |- m ::: Nat ==> (? n. m >> Con succ @@ n) =
                     converges (Con ncase @@ m @@ undef @@ Lam (Con zero))
```

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 `cord_opensim`

```
 ctxt |- s <cord> t ::: a ==> ctxt |- s <opensim> t ::: a
```

8 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 which should scale up to larger languages. We have shown that reformulation of some standard theoretical techniques results in significant increases in efficiency and clarity of automated proof. We are 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

[1] M. Abadi and L. Cardelli. A Theory of Objects. Springer Verlag, 1996.

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

[3] G. M. Bierman. Observations on a Linear PCF (Preliminary Report). Tech Report 412, Univ of Camb Comp Lab, 1997.

[4] E. Börger and W. Schulte. A Programmer Friendly Modular Definition of the Semantics of Java. Formal Syntax and Semantics of Java. Springer LNCS, 1998, to appear.

[5] L. Cardelli. *Type Systems*, chapter 103, pages 2208–2236. CRC Press, 1997.

[6] G. Collins and S. Gilmore. Supporting Formal Reasoning about Standard ML. Tech Report, LFCS, Univ of Edinburgh, 1994.

[7] R. L. Crole. How Linear is Howe? In *Advances in Theory and Formal Methods 1996*, pages 60–72. Imperial College Press, 1996.

[8] R. L. Crole. Implementing Operational Semantics (Preliminary Report). Tech Report 1997/42, Dept of Mathematics and Computer Science, Univ of Leicester, 1997.

[9] R. L. Crole. The KOREL Programming Language (Preliminary Report). Tech Report 1997/43, Dept of Mathematics and Computer Science, Univ of Leicester, 1997.

[10] R. L. Crole. Lectures on (Co)Induction and (Co)Algebras. Tech Report 1998/12, Dept of Mathematics and Computer Science, Univ of Leicester, 1998.

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

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

[13] A. D. Gordon. Bisimilarity as a theory of functional programming: Mini-Course. Notes to accompany graduate lectures at BRICS in the Computer Science Department Aarhus University, March 1995.

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

[15] A. D. Gordon and T. Melham. Five axioms of alpha-conversion. In *Proc. of TPHOLs'96*, volume 1125 of *LNCS*, pages 173–190, Turku, Finland, August 1996.

[16] J. F. Groote and F. Vaandrager. Structured Operational Semantics and Bisimulation as a Congruence. *Information and Computation*, 100:202–260, 1992.

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

[18] B. Joy J. Gosling and G. Steele. *The Java Language Specification, version 1.0*. Addison-Wesley, 1996. ISBN: 0-201-63451-1.

[19] G. Kahn. Natural semantics. In *Programming of Future Generation Computers*, pages 237–258. Elsevier, 1988.

[20] S. B. Lassen. Relational Reasoning about Functions and Nondeterminism. PhD Thesis, Dept of Computer Science, Univ of Aarhus.

[21] S. Maharaj and E. Gunter. Studying the ML module system in HOL. In *Proc. of the 7th Workshop on HOL Theorem Proving and Its Applications*, volume 859 of *LNCS*, 346–361, 1994. Springer-Verlag.

[22] I. A. Mason and C. L. Talcott. Equivalence in Functional Languages with Effects. *Journal of Functional Programming*, 1:287–327, 1991.

[23] T. F. Melham and M. J. C. Gordon. *Introduction to HOL*. Camb Univ Press, 1993.

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

[25] T. Nipkow and D. von Oheimb. Machine-Checking the Java Specification: Proving Type Safety, Formal Syntax and Semantics of Java. Springer LNCS, 1998, to appear.

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

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

[28] G.D. Plotkin. L.C.F. Considered as a Programming Language. *Theoretical Computer Science*, 5:223–255, 1977.

[29] G.D. Plotkin. A structural approach to operational semantics. Tech Report DAIMI–FN 19, Dept of Computer Science, Univ of Aarhus, Denmark, 1981.

[30] F. Regensburger. HOLCF: Higher order logic of computable functions. In *Proc. of the 8th Workshop on HOL Theorem Proving and Its Applications*, volume 971 of *LNCS*, pages 293–307, USA, September 1995.

[31] D. Syme. Reasoning with the formal definition of Standard ML in HOL. In *Proc. of the 6th Workshop on HOL and its Applications*, volume 780 of *LNCS*, pages 43–60, 1993.

[32] D. Syme. DECLARE: A Prototype Declarative Proof System for Higher Order Logic. Tech Report, Comp Lab, Univ of Camb, 1997.

[33] D. Syme. Proving JavaS Type Soundness. Tech Report, Comp Lab, Univ of Camb, 1997.