An Operational Approach to Combining Classical Set Theory and Functional Programming Languages

Douglas J. Howe[1] and Scott D. Stoller[2]

[1] AT&T Bell Labs, 600 Mountain Ave., Room 2B-438
Murray Hill, NJ 07974, USA
howe@research.att.com.
[2] Department of Computer Science, Cornell University
Ithaca, NY 14853, USA.
stoller@cs.cornell.edu.

Abstract. We have designed a programming logic based on an integration of functional programming languages with classical set theory. The logic merges a classical view of equality with a constructive one by using equivalence classes, while at the same time allowing computation with representatives of equivalence classes. Given a programming language and its operational semantics, a logic is obtained by extending the language with the operators of set theory and classical logic, and extending the operational semantics with "evaluation" rules for these new operators. This operational approach permits us to give a generic design. We give a general formalism for specifying evaluation semantics, and parameterize our design with respect to languages specifiable in this formalism. This allows us to prove, once and for all, important properties of the semantics such as the coherence of the treatment of equality.

1 Introduction

Our goal is to develop a logic suitable for the following.

Metatheory of programming languages. Although programming languages differ dramatically in their syntax and semantics, a great deal of the work in proving the correctness of specific programs is language-independent, involving, for example, reasoning about properties of mathematical models of data. One way of sharing formal knowledge between different languages is to reason about them via an embedding into a single logic. Since the metatheory of programming languages is rich in computational content (*e.g.,* program transformation, interpretation, compilation), this logic, in addition to supporting a broad range of mathematical reasoning, should itself contain a powerful programming language.

Constructive mathematics. Constructive mathematics gives a powerful, high-level method for developing correct programs. We see no practical advantage to staying within a strictly constructive formalism, and instead prefer to view constructive mathematics as a special case of classical mathematics: one always does classical mathematics, but when the definitions are appropriate and the

reasoning is sufficiently constructive, a program can be extracted from a proof. This view is compatible with Bishop's style of constructive mathematics [3].

Mathematical modeling of programs and software systems. For example, we want to support at least the general kind of set-theoretic modeling used in the specification language Z [18].

Providing a highly expressive type system for functional programming languages. Types are a good way of organizing knowledge about functional programs. For example, the Calculus of Constructions [7] and Nuprl [6], have type systems rich enough to serve as specification languages for functional programs. Many of a program's properties can be expressed in its type, while in less expressive type systems, such as the simple type theory of HOL [9], almost all properties are formalized as a predicate over some simple type.

As a suitable logic for these purposes, we propose an integration of a functional programming language with full classical set theory (ZFC). Our approach to accomplishing this integration is to give precedence to operational semantics. Instead of starting with set theory and attempting to give denotational explanations within set theory of the constructs of the programming language, we start with a rule-based specification of the operational semantics of the programming language and add to it rules which let us "evaluate" programs that contain operators and constants from set theory and classical logic. The result of evaluation gives the set-theoretic meaning of the program.

A benefit of this approach is that it is possible to give a generic design. Adequate formalizations of set theory are well-known, but there is no general agreement on what should go into a functional programming language. We give a formalism for specifying evaluation semantics, similar to the one given in [12], and parameterize our account with respect to languages specifiable in this formalism. This allows us, for example, to prove, once and for all, such properties as congruence of equality and adequacy of our semantics for the interpreter (see the discussion of equality below).

The basic idea in our operational approach is simple. Let P_0 be the given programming language, and assume its operational semantics is presented as a set of rules inductively defining an evaluation relation $\Downarrow$. A language P is obtained is obtained by adding new operators and constants, and adding rules extending the definition of $\Downarrow$. The new evaluation relation can be thought of as specifying a meaning-finding procedure. Meaning of a program e is determined by evaluating e, analyzing the value, and possibly continuing on to find meanings of components of the value. This procedure either fails or results in a set-theoretic object which is the meaning of e.

If e evaluates to an atomic value, say a number n, then the meaning of e is just the set-theoretic encoding of n. If e evaluates to a use of a data constructor, say a pair (u, v), then first obtain the meanings u', v' of u, v. If these exist, the meaning of e is the set-theoretic pair (u', v').

For functions we need to make a restriction. Assigning a set-theoretic meaning to polymorphic functions such as $\lambda x.\ x$ is problematic [17]. We allow such programs, but do not give them a set-theoretic meaning, although they can ap-

pear in larger programs that do have a meaning. We could try to give a general account of the forms of functional abstraction we can give set-theoretic meaning to, but for expediency we fix one particular form. We assume P_0 has expressions of the form $\lambda x.\, b$. In P, these expressions are given "type bounds", so that they have the form $\lambda x : A.\, b$, where A is an arbitrary expression. The "type" A is present only for the purpose of assigning meaning — we do not impose a syntactic type discipline on P. To find the meaning of e when e evaluates to $\lambda x : A.\, b$, first find the meaning of A. If this is a constant representing a set α, then for each $\beta \in \alpha$, find the meaning f_β of $b[\beta/x]$. The meaning of e is then the representation in set theory of the mapping $\beta \mapsto f_\beta$. Of course, we need to explain how to evaluate programs like $b[\beta/x]$ that contain constants representing sets. This is not hard. For example, to evaluate the application $\phi(e)$, where ϕ is the set-theoretic graph of a function, first find the meaning α of e then return the β such that $(\alpha, \beta) \in \phi$.

This informal description suggests why our account can be made generic in the way described above: the procedure for finding meaning does not mention any constructs, or operators, that are "non-canonical", *i.e.*, whose uses do not construct values but instead require further computation. We are explicit about the forms of data, but other constructs are simply "evaluated away" in the meaning-finding procedure.

It is straightforward to include in P rules for evaluating operators from set theory and logic. For example, to evaluate a universally quantified expression $\forall x \in A.\, P$, first find the meaning α of A (if any), then find the meaning p_β of $P[\beta/x]$ for each $\beta \in \alpha$. If each p_β is either *true* or *false*, then return *true* if each p_β is *true*, and return *false* otherwise.

So that programs can call on uncomputable functions, and to provide a convenient way to name sets we can prove to exist, we add a choice operator similar to Hilbert's ε operator. In particular, if $P[v/x]$ is boolean-valued for all sets v and is true for some v_0, then $\varepsilon x.\, P$ returns the least such v_0 (according to some fixed ordering). Adding an evaluation rule for ε is straightforward.

The natural notion of equality of programs with meanings is to take $t = t'$ if t and t' have the same set as their meaning. Following [11], we can also define another equality, which we will denote by $\sim$, that is based directly on the operational semantics of P, is defined over all programs, and which justifies the usual kinds of equational reasoning about functional programs. These two equalities are compatible, in the sense that they agree on terms with meanings, and furthermore, if t has a meaning and $t \sim t'$, then t' has a meaning. This means, in particular, that if $t = t'$, then t can be replaced by t' in any program without affecting that program's meaning (if any). So, although we have given an operational account of set-theoretic semantics, we have recovered the basis for the usual forms of equality reasoning.

Fundamental to set-theoretic reasoning is the use of equivalence classes to impose new equalities on existing sets, as in forming the rational numbers as a quotient on pairs of integers. We would like to be able to write programs that manipulate members of quotient sets, and so we need to make computational

sense of equivalence classes.

We solve this problem as follows. We include in P an operator $[e]_A$ which, if (the meaning of) A is a set of equivalence classes, returns the equivalence class of A that contains (the meaning of) e. We also include an operator *qap* for applying a function to an equivalence class. $qap(f;t)$ evaluates as follows. First, it finds the meaning ξ of t. If ξ is an equivalence class, it checks that f returns the same result for all members of ξ, then returns that result. The checking of f's values on ξ is essential. It means that if we have successfully evaluated a program containing $qap(\xi, f)$, then the computation did not need the whole equivalence class and could have just taken an arbitrary representative. Because of this, we can delay equivalence class construction, using $[\cdot]_A$ as a data constructor. This point is expanded on below.

Call this extended language P, and use $\Downarrow$ to denote the evalution relations of both P_0 and P. A program P is *computable* if it is composed only from *qap*, $[\cdot]$ and the operators of P_0. We count expressions $\lambda x\!:\!A.\,b$ as members of P_0 for arbitrary A in P. We define an interpreter for computable programs as a new evaluation relation $\Downarrow'$ defined by the evaluation rules of P_0 together with two rules for equivalence classes. One rule simply treats $[\cdot]$ as a data constructor, so $[e]_A \Downarrow' [v]_A$ if $e \Downarrow' v$. For *qap*, if $t \Downarrow' [v]_A$ and $f(v) \Downarrow' v'$, then $qap(f;t) \Downarrow' v'$. Thus we delay the creation of an equivalence class from a representative of the class, and when the equivalence class gets used by *qap*, only the representative is needed.

We can prove that if e is a program of P and $e \Downarrow v$, then there is a v' in P such that $e \Downarrow' v'$ and v is equivalent to v'. This result says that if e has a meaning then we can compute a value expression for it with the same meaning. Of course, this value may contain uses of $[\cdot]$. The proof of the result is somewhat similar to the "inclusive predicate" proofs of computational adequacy in denotational semantics.

We can extend the set of computable programs to include type expressions as data. Type constructors, such as cartesian product, are encoded using the choice operator ε. Thus, an expression $A \times B$ evaluates to a term representing the set that is the product of (the meanings of) A and B. To include such exprssions as data, we introduce a variant ε' of ε which can return *only* types, i.e. none of its values can be interpreted as program data. We can now extend our interpreter to delay evaluation of ε'. We do not ever need to evaluate these delayed subprograms since no constructs in computable programs can analyze values returned by ε'. These type expressions can be passed around and used in expressions A in $\lambda x\!:\!A.\,b$, allowing a kind of polymorphism.

To ensure that our two notions of equality are compatible, we need to be careful with function application. Consider the functions $\lambda x\!:\!\emptyset.\,0$ and $\lambda x\!:\!\emptyset.\,1$, where $\emptyset$ is the empty set. These have the same set-theoretic meanings. If they are to be observationally congruent, we must ensure that their application to arguments outside their domain cannot be evaluated. We therefore modify the rule for application, inserting a "run-time" check that the argument is of the right type.

Our treatment of equality permits a simple explanation of implementation of abstract data types. Consider, for example, an abstract data type for the set Q of the rational numbers together with the usual operations and equations relating them. If our language P_0 is a typical functional language, and we try to use it to implement Q in the obvious way as a set of pairs of integers, then the equations for Q will not be satisfied, since the equality in Q will be equality of pairs of integers. We might still be able to establish suitable representation independence results, but this will require considerable extra work. (See [5] for an approach along these lines.) This is not required in our setting, since in P we can give an implementation of Q by taking the obvious one in P_0 and using quotienting to make it respect the equations of Q.

Our operational approach has a few drawbacks. First, since all the operators in the logic obtain their meaning through an inductive definition of an "operational" semantics, they must all be monotone with respect to a computational preorder (less-defined-than). One consequence of this is that the logic cannot contain a convergence predicate which tells whether a program is defined (has a value). We avoid this deficiency by using a sequent calculus formulation of the logic in which the free variables of sequents range over all programs (defined or not). A similar approach is taken in Nuprl, where it has proven to be convenient for a wide range of reasoning about termination (definedness).

This monotonicity requirement also affects the treatment of fixed points. We cannot have a fixed-point operator which returns a function whose domain is exactly the set of arguments on which computation terminates, since this set is a non-monotone value. We can still use fixed-point operators, but we must specify in advance a domain for the function being defined, and the function must be total on this domain. Our experience with Nuprl suggests that having to supply a domain is at most a minor practical problem.

A third problem is that our semantics is not compositional. We ultimately want compositional reasoning principles, but instead of fixing some in advance through a denotational semantics, we derive within the logic, from some axiomatization of symbolic computation, whatever principles seem useful. We give an example of this in Section 5, where we derive a rather conventional rule for proving termination of a recursive program by well-founded induction.

The following aspects of our work are new.

- *Computational interpretation of equivalence classes.* In [4], Breazu-Tannen and Subrahmanyam give a logic for reasoning about programs using structural recursion over data types involving constructors subject to some equations. Their idea, to assign a meaning of $\perp$ to definitions by structural recursion that do not respect the equations, is similar to our treatment of $qap(f;t)$, which is undefined if f does not respect equality as given by the equivalence class t. However, they do not deal with general equivalences or equivalence classes, and their proof of adequacy relies on a normalization property of their programming language.
- *General operational account.* This allows us to prove once, for a fairly large class of programming languages, that the operational equivalence of the lan-

guage is compatible with set-theoretic equality, and that equivalence classes can be given a computational interpretation.

– *Congruence proof.* Our program equivalence is a generalization of applicative bisimulation [1]. We prove it is a congruence using a new extension of the proof method introduced by the first author in [11]. The extension is needed to deal with typed λ-abstractions.

There have been at two other recent attempts to combine set theory with computation. In [2], Beeson extends to ZF set theory Feferman's idea for a classical model of his theory T_0 [8]. This might appear more general than our logic, since one can build function types containing untyped abstractions. For example, the type $N \to N$ would contain the polymorphic identity $\lambda x.\ x$, and, in general, any f such that $f(e)$ evaluates to a number whenever e does. However, programs enter the set-theoretic world essentially via encoding their syntax. Since programs can mention set theoretic objects, the type $N \to N$ is already semantically as large as the entire set-theoretic universe V used as a model. This means that $(N \to N) \to N$ cannot be understood as a function space in the traditional sense, since no function whose graph is in V can have a sufficiently large domain. Also, Beeson does not deal with equivalence classes.

Map theory [10] provides an alternate foundation to set theory, in which everything is reduced to a "map". ZF set theory can be interpreted in map theory, and a function found inside the interpreted set theory is itself a map, but the input-output behaviour of the map does not directly correspond to that of the function. The map is more like a recognizer, or generator, for the set of pairs of the function.

An important issue here is whether ZF itself is practical for large-scale formal reasoning. There is a substantial body of experience supporting a positive answer. Variants of ZF have been successfully implemented and applied in the theorem-provers Isabelle [15, 16] and Ontic [14]. Experience with Nuprl is also relevant, since it shows that a high level of automation of reasoning can be achieved even when the logic strongly favours expressive power over the ability to uniformly apply a powerful automated-reasoning method (such as resolution or term rewriting).

The rest of the paper is organized as follows. Section 2 discusses syntax, including how to extend the syntax of a programming language to incorporate sets and logic. Section 3 describes the class of languages P to which our construction can be applied, gives the extended operational semantics that incorporates set theory, and gives some basic results about equality. Section 4 gives an executable fragment of the extended language and defines an interpreter for it, and shows that our semantics is adequate for the interpreter. Section 5 sketches the design of a sequent-based proof systems for our logic. We close with some plans for future work.

2 Syntax

The remainder of the paper is parameterized by a functional programming language P_0. We take the syntax of a programming language P to be given by a *language* $L = (O, K, \alpha)$ where O is a set of *operators*,

$$\alpha \in O \to \{\, (k_1, \ldots, k_n) \mid n, k_i \geq 0 \,\}$$

is a function assigning to each operator an *arity*, and $K \subseteq O$. The members of K are called *canonical*; all other operators are *noncanonical*. The former are intended to be value constructors.

Fix an infinite set of variables. A *term* over L is either a variable, or

$$\tau(s_1; \ldots; s_n)$$

where $\tau \in O$, $\alpha(\tau) = (k_1, \ldots, k_n)$ for some $k_1, \ldots, k_n$, and each s_i is an *operand* of the form $\mathbf{x}_i.\ a_i$ where a_i is a term and $\mathbf{x}_i$ is a sequence of k_i distinct variables. Define

$$\mathit{outer}(\tau(s_1; \ldots; s_n)) = \tau.$$

A term t is *(non-) canonical* if and only if $\mathit{outer}(t)$ is (non-) canonical.

We impose binding structure on terms by specifying that in each operand $\mathbf{x}.\ a$ the variables in $\mathbf{x}$ bind in a. The usual notions of substitution and α-equality apply. We identify α-equal terms. Let T^L be the set of terms over L and T_0^L the set of closed terms (we drop the superscripts when clear from context). If X is a set of terms, define $X^2 = X \times X$.

For example, the λ-calculus can be cast as a language $L = (O, \alpha)$ by taking $O = \{\, \lambda, \mathit{ap} \,\}$, $\alpha(\lambda) = (1)$ and $\alpha(\mathit{ap}) = (0, 0)$, and using $\lambda x.\, b$ and $a(b)$ as shorthand for $\lambda(x.\ b)$ and $\mathit{ap}(a; b)$. Note the ambiguity here: we use the same notation, $u(v)$, for both function application and for applying an operator to a sequence of operands. However, the meaning should always be clear from context.

We need some definitions and notational conventions for binary relations between terms. Let $\eta \subset T^2$. For $a, b \in T$, write $a\ \eta\ b$ for $(a, b) \in \eta$. If $s = x_1, \ldots, x_n.\ a$ and $s' = x'_1, \ldots, x'_n.\ a'$ are operands, then define $s\ \eta\ s'$ if there are distinct variables $z_1, \ldots, z_n$, not free in s or s', such that

$$a[z_1, \ldots, z_n / x_1, \ldots, x_n]\ \eta\ a'[z_1, \ldots, z_n / x'_1, \ldots, x'_n].$$

If $\mathbf{s}$ and $\mathbf{s}'$ are operand sequences $s_1, \ldots, s_n$ and $s'_1, \ldots, s'_n$, respectively, such that $\tau(\mathbf{s})$ and $\tau(\mathbf{s}')$ are terms for some τ, then define $\mathbf{s}\ \eta\ \mathbf{s}'$ if $s_i\ \eta\ s'_i$ for each i, $1 \leq i \leq n$. If $\eta \subset T^2$, define $\eta^\circ \subset T^2$ by $a\ \eta^\circ\ a'$ if $\sigma(a)\ \eta\ \sigma(a')$ for every substitution σ such that $\sigma(a)$ and $\sigma(a')$ are closed. Finally, for $\eta \subset T^2$, define $\eta_0 = \eta \cap {T_0}^2$.

A preorder $\eta \subset T^2$ is a *precongruence* if for all $\tau(\mathbf{s}), \tau(\mathbf{s}') \in T$, if $\mathbf{s}\ \eta\ \mathbf{s}'$ then $\tau(\mathbf{s})\ \eta\ \tau(\mathbf{s}')$. It is a *precongruence on closed terms* if for all $\tau(\mathbf{s}), \tau(\mathbf{s}') \in T_0$, if $\mathbf{s}\ \eta^\circ\ \mathbf{s}'$ then $\tau(\mathbf{s})\ \eta\ \tau(\mathbf{s}')$. Note that $\eta \subset {T_0}^2$ is a precongruence on closed terms if and only if η° is a precongruence. A *congruence* is a precongruence that is an equivalence relation.

Let L_0 be the language of P_0. We add new operators to L_0 to obtain L which will be the language for the programming language P which mixes P_0 and set theory. Since the semantics of P will be given via evaluation, L must contain values (terms that evaluate to themselves) denoting all sets of interest. We achieve this as follows. We assume there is a set V which gives a model of ZFC when $\in$ is interpreted as the restriction of the ordinary membership relation to V. Such a set exists if there is an inaccessible cardinal.[3] Let $\rho, \sigma, \tau, \ldots$ range over V.

L will have enough new operators so that we can establish a bijection between V and the set of *normal terms* of L, which are defined below. We need to use slightly non-standard set-theoretic encodings of objects such as functions, pairs and equivalence classes, since, for technical reasons, we need to ensure that no member of V can be interpreted as more than one kind of object. Hence, all sets that are encodings of "data" (as opposed to the "pure" sets that correspond to types) are paired with a distinguishing tag. Choose distinct sets $fn, ec \in V$ to serve as tags for functions and equivalence classes, respectively. Let $(\cdot, \cdot)$ denote the standard set-theoretic pairing operation. We define operations that add and remove tags.

$$fun\,(\phi') = (fn, \phi) \qquad [\rho] = (ec, \rho) \qquad |(\sigma, \tau)| = \tau \tag{1}$$

We use ϕ to range over sets (in V) of the form (fn, ϕ'), where ϕ' ranges over graphs of functions. ξ ranges over sets of the form (ec, ρ) for non-empty sets ρ.

We assume that L_0 contains a collection C of canonical operators with arities of the form $(0, \ldots, 0)$; these are data constructors and will be used in normal terms. We also assume that L_0 contains a canonical operator λ of arity $(0, 1)$ and a corresponding application operator ap. λ will be used to build normal terms representing functions. Fix an injection i of C into $V - \{fn, ec\}$.

A member of V is *tagged* if it is a ϕ or a ξ, or if it has the form

$$(i(c), (\sigma_1, \ldots, \sigma_n))$$

for some $c \in C$ whose arity has length n. We will always used α to range over members of V that are not tagged.

L is obtained from L_0 by adding a canonical operator $\theta_\alpha, \theta_\phi, \theta_\xi$ of arity $(), (0), ()$ for each α, ϕ and ξ in V, respectively. We will usually identify α and θ_α, ϕ and θ_ϕ, and ξ and θ_ξ. We also add operators $qap, [], \varepsilon, \varepsilon', \forall, \Rightarrow, \in, =$. The arities of these operators will be apparent from their uses below. We will use the usual infix notation for uses of the operator $\in$, relying on context to distinguish it from the mathematical membership relation. We assume that none of the operators we add are present in L_0.

Definition 1 *Inductively define $\widehat{\rho}$, for $\rho \in V$, as follows.*

$$\widehat{\rho} = \begin{cases} c(\widehat{\sigma_1}, \ldots, \widehat{\sigma_n}) & \text{if } \rho = (i(c), (\sigma_1, \ldots, \sigma_n)) \\ \lambda x : \widehat{dom(\phi')}.\ \theta_\rho(x) & \text{if } \rho = (fn, \phi') \\ \theta\rho & \text{otherwise} \end{cases} \tag{2}$$

[3] We could take V to be a proper class for most of the results in the paper, but not for our treatment of polymorphism.

where $dom(\phi')$ is the domain of ϕ'. A term $t \in T_0^L$ is normal *if $t = \widehat{\rho}$ for some $\rho \in V$.*

We need truth values to give semantics to the logical operators, so we assume there are nullary operators $T, F \in C$. We also use T, F to denote their counterparts in V (namely, the ρ_1, ρ_2 such that $\widehat{\rho_1} = T$ and $\widehat{\rho_2} = F$).

3 Semantics

We assume that the operational semantics of P_0 is presented as a collection of evaluation rules. In this section, we first give a precise definition of a general form of evaluation rule, and then point out several restrictions on the set of rules of P_0. We then add to the given set of rules new evaluation rules for operators in L. This new set of rules inductively defines the evaluation relation $\Downarrow$, which provides a semantics for the terms of L. Finally, we establish some basic properties of our semantics.

In giving the new evaluation rules, it is convenient to introduce an auxiliary binary relation $\triangleright$, and rules for it, so that $\Downarrow$ and $\triangleright$ are mutually inductively defined. The interpretation of $e \triangleright \sigma$ is that the closed term e has σ as a set-theoretic meaning. This new relation is not strictly necessary: a premise in which it occurs can be regarded as a "macro" for a (large) set of premises not involving it. However, it is convenient in our proofs to reason directly about $\triangleright$.

3.1 A General Rule Format

Inference rules for evaluation are specified using an extension of the term language to include, for all $i \geq 0$, an infinite set of variables which we call the *metavariables of arity i*. A *term schema* is built in the same way as a term, except that it may also be an expression of the form $P[\mathbf{a}]$, where P is a metavariable of arity i ($i \geq 0$), and $\mathbf{a}$ is a sequence of i term schemas. We write P for $P[]$ and use capital letters in term schemas exclusively for metavariables. A *simple term schema* has the form $\tau(\mathbf{x}_1.\ P_1[\mathbf{x}_1]; \ldots; \mathbf{x}_n.\ P_n[\mathbf{x}_n])$, where the P_i are distinct metavariables.

A *second-order substitution* is a partial map σ from metavariables to operands such that if $\sigma(P)$ is defined then it has the form $x_1, \ldots, x_n.\ b$, where n is the arity of P. The application of σ to term schemas is similar to ordinary substitution, except that if $\sigma(P)$ is $x_1, \ldots, x_n.\ b$, then

$$\sigma(P[a_1, \ldots, a_n]) = b[\sigma(a_1), \ldots, \sigma(a_n)/x_1, \ldots, x_n].$$

σ is *closed* if $\sigma(P)$ is closed for all P in the domain of σ. For $\nu \subset T^2$, define $\sigma\ \nu\ \sigma'$ if for every P in the domain of σ, $\sigma'(P)$ is defined and $\sigma(P)\ \nu\ \sigma'(P)$.

An *evaluation rule* is an inference rule whose premises and conclusion are formulas of the form $a \Downarrow b$, where a and b are term schemas with no free ordinary variables. a and b are called the left-hand side and right-hand side, respectively, of the the formula. We impose the following conditions on rules. The set of

premises may be infinite but comes with a well-ordering. Thus, a rule consists of a set I, a well-ordering $<_I$ over I, and a schema of the form

$$\frac{\{a_i \Downarrow b_i\}_{i \in I}}{a \Downarrow b} \tag{3}$$

We say that the rule is *for* $outer(a)$.

We impose the following syntactic restrictions on each rule:

1. a is a simple term schema. Let $\theta = outer(a)$.
   (a) If θ is canonical, then b is a term schema with outer operator θ.
   (b) If θ is non-canonical, then b is a metavariable and $b = b_i$ for some $i \in I$.
2. For all $i \in I$, b_i is a metavariable or a simple term schema and has no metavariables in common with a or with b_j for $j \neq i$.
3. For each $i \in I$ and each metavariable P of a_i, P occurs in a or in b_j for some $j <_I i$.

As an example, below are rules for the pure lazy λ-calculus.

$$\frac{F \Downarrow \lambda x.\, B[x] \quad B[A] \Downarrow C}{F(A) \Downarrow C} \qquad \frac{}{\lambda x.\, B[x] \Downarrow \lambda x.\, B[x]}\,.$$

The standard constructs for performing case analysis and term decomposition by pattern matching (á la SML) can also be cast in this form.

A *closed instance* of a rule is the result of applying to it a closed second-order substitution whose domain contains all the metavariables occurring in the rule. The evaluation relation defined by a set of rules is the relation inductively defined by the set of all closed instances of rules.

In what follows, we will be somewhat sloppy about our use of second-order variables in rules, and will usually use conventional first-order notation, .g. writing $t[a/x]$ instead of $T[A]$. We will also make liberal use of *ad hoc* rule schemes that stand for sets of rules.

3.2 Assumptions

There are four restrictions we need to make on the set of rules defining the evaluation relation of P_0. The first restriction is that the rules may only mention operators from $L_0 - \{\lambda\}$. The second is that P_0 contains the following as the only rule for each $c \in C$.

$$\frac{t_1 \Downarrow v_1 \quad \cdots \quad t_n \Downarrow v_n}{c(t_1, \ldots, t_n) \Downarrow c(v_1, \ldots, v_n)} \tag{4}$$

The other two restrictions are needed to ensure that the evaluation relation of the extended language P has some basic properties. We could follow [12] and attempt to express these restrictions without mentioning P by quantifying over all possible extensions of P_0, but all the ways to do this that we know of are somewhat arbitrary and complicated. Instead, we will simply make these restrictions directly in terms of P. We need the evaluation relation of P to be

single-valued, in the sense that if $e \Downarrow v$ then $v \Downarrow v$, and *determinate*, in the sense that if $e \Downarrow v$ and $e \Downarrow v'$ then v is identical to v'. Furthermore, we require that these properties hold of the modified evaluation relations $\bar{\Downarrow}$ defined in Section 4.

3.3 New Rules

To give the semantics of P, we add evaluation rules to those of P_0. The new rules are all of the form described earlier if we take premises of the form $t \triangleright \rho$ to stand for sets of premises. The rules for λ-abstraction and application are as follows.

$$\frac{}{\lambda x : A.\ b \Downarrow \lambda x : A.\ b} \tag{5}$$

$$\frac{f \Downarrow \lambda x : A.\ b \quad t \Downarrow u \quad u \triangleright \rho \quad A \triangleright \sigma \quad \rho \in \sigma \quad b[u/x] \Downarrow v}{f(t) \Downarrow v} \tag{6}$$

$$\frac{t \triangleright \tau \quad (\tau, \rho) \in |\phi|}{\phi(t) \Downarrow \widehat{\rho}} \tag{7}$$

The premises involving $\triangleright$ in the rule for application accomplish the "run-time typecheck". The value u of the argument t must have a set theoretic meaning ρ that is a member of the set which is the set theoretic meaning of the "type" A.

For equivalence classes we have the following two rules.

$$\frac{t \triangleright \rho \qquad A \triangleright \tau \qquad \rho \in |\xi| \qquad \xi \in \tau \qquad disjoint(\tau)}{[t]_A \Downarrow \xi} \tag{8}$$

where $disjoint(\tau)$ holds if and only if τ is a set of disjoint equivalence classes, *i.e.*, for all $\rho \in \tau$ there is a ξ such that $\rho = \xi$, and for all $\xi, \xi' \in \tau$, if $|\xi| = |\xi'|$ then $|\xi| \cap |\xi'| = \emptyset$. To apply a function to an equivalence class, one must show that it produces the same result on all members of the equivalence class.

$$\frac{t \triangleright \xi \qquad \forall \rho \in |\xi|.\ f(\widehat{\rho}) \triangleright \tau}{qap(f; t) \Downarrow \widehat{\tau}} \tag{9}$$

Constants are values:

$$\frac{}{\xi \Downarrow \xi} \qquad \frac{}{\alpha \Downarrow \alpha} \tag{10}$$

Define the abbreviation $Bool \equiv \{\widehat{T, F}\}$. The choice operator evaluates as follows:

$$\frac{t[\widehat{\rho}/x] \Downarrow T \qquad \forall \sigma <_V \rho.\ t[\widehat{\sigma}/x] \Downarrow F \qquad \forall \sigma.\ (t[\widehat{\sigma}/x] \in Bool) \Downarrow T}{\varepsilon(x.t) \Downarrow \widehat{\rho}} \tag{11}$$

where $<_V$ is a fixed well-ordering of V. The choice of a value of ρ when there are multiple possibilities is arbitrary. ε' is like ε, except it can only return a "type".

$$\frac{t[\widehat{\alpha}/x] \Downarrow T \qquad \forall \alpha' <_V \alpha.\ t[\widehat{\alpha'}/x] \Downarrow F \qquad \forall \sigma.\ (t[\widehat{\sigma}/x] \in Bool) \Downarrow T}{\varepsilon'(x.t) \Downarrow \widehat{\alpha}} \tag{12}$$

The rules for the operators $\forall$, $\Rightarrow$, $\in$ and $=$ are straightforward and we omit them. For example, $(a = b) \Downarrow T$ if there is a ρ such that $a \triangleright \rho$ and $b \triangleright \rho$, and $(a \in A) \Downarrow F$ if $a \triangleright \sigma$, $A \triangleright \alpha$ and $\sigma \notin \alpha$.

We now give the rules for the auxiliary relation $\triangleright$. These rules capture the meaning-finding procedure described in the introduction.

$$\frac{t \Downarrow \xi}{t \triangleright \xi} \qquad \frac{t \Downarrow \alpha}{t \triangleright \alpha} \tag{13}$$

$$\frac{t \Downarrow c(s_1, \ldots, s_n) \qquad s_1 \triangleright \sigma_1 \quad \cdots \quad s_n \triangleright \sigma_n}{t \triangleright (i(c), (\sigma_1, \ldots, \sigma_n))} \tag{14}$$

$$\frac{t \Downarrow \lambda x : A.\, b \qquad A \triangleright \sigma \qquad dom(|\phi|) = \sigma \qquad \forall \rho \in \sigma.\, b[\widehat{\rho}/x] \triangleright |\phi|(\rho)}{t \triangleright \phi} \tag{15}$$

In 15, $|\phi|(\rho)$ denotes set-theoretic application of the function $|\phi|$.

Definition 2 *A term v is a* value *if and only if there is a t such that $t \Downarrow v$. A term e* denotes*, or* has a meaning*, if there exists $\rho \in V$ such that $e \triangleright \rho$.*

The proofs of the following are straightforward.

Lemma 1 *For all ρ, $\widehat{\rho} \triangleright \rho$.*

Lemma 2 *If $e \triangleright \rho$ and $e \triangleright \rho'$, then $\rho = \rho'$.*

Lemma 3 *For all t and ρ, $t \triangleright \rho$ if and only if there exists v such that $t \Downarrow v$ and $v \triangleright \rho$.*

3.4 Equality

Define $a \,\tilde{\in}\, A$ if for some $\beta \in \alpha \in V$, $a \triangleright \beta$ and $A \triangleright \alpha$.

Definition 3 *For $\eta \subset {T_0}^2$, define $[\eta] \subset {T_0}^2$ by $a\ [\eta]\ a'$ if*

1. *for all terms $\theta(\mathbf{s})$ where $\theta \neq \lambda$, if $a \Downarrow \theta(\mathbf{s})$ then there exists $\mathbf{s}'$ such that $a' \Downarrow \theta(\mathbf{s}')$, and $\mathbf{s}\ \eta^\circ\ \mathbf{s}'$, and*
2. *for all terms of the form $\lambda x{:}A.\, b$, if $a \Downarrow \lambda x{:}A.\, b$ then there exists A', b' such that $a' \Downarrow \lambda x{:}A'.\, b'$, $A\ \eta\ A'$, and for all t, if $t \,\tilde{\in}\, A$ then $b[t/x]\ \eta\ b'[t/x]$.*

Define $\leq$ as the largest relation η such that $\eta \subseteq [\eta]$.

It is easy to show that $\leq = [\leq]$ and that $\leq$ is a preorder. Define $a \sim b$ if $a \leq b$ and $b \leq a$.

Theorem 1 *$\leq^\circ$ is a precongruence (hence $\sim^\circ$ is a congruence).*

Proof. The proof is a straightforward adaptation of the method first presented in [11] and applied in [12]. We just outline the differences here. The key idea in the method is to define, using $\leq$, an auxiliary relation $\widehat{\leq}$ that is easy to show a congruence, and then to show that $\widehat{\leq} \subseteq \leq$ by coinduction, *i.e.*, by showing $\widehat{\leq} \subseteq [\widehat{\leq}]$. This is proved by induction on the definition of evaluation. We use the same definition of $\widehat{\leq}$ as in [12][4], but we adapt the induction to treat $\triangleright$ directly. The proof reduces to showing the following by induction on the definition of $\Downarrow$ and $\triangleright$.

1. If $a \triangleright \alpha$, then $a \mathrel{\widehat{\leq}} a'$ implies $a' \triangleright \alpha$.
2. If $a \Downarrow v$, then $a \mathrel{\widehat{\leq}} a'$ implies there exists v such that $a' \Downarrow v'$ and $v \mathrel{\widehat{\leq}} v'$.

The remainder of the proof is an easy adaptation of the original method.

The proof of the following is straightforward.

Lemma 4 *If $a \leq b$ and $t \Downarrow v$, then $t[b/a] \Downarrow v'$ and $v \leq v'$.*

We now define the relation we denoted by $=$ in the introduction.

Definition 4 *Define $s =_\triangleright t$ if there is a ρ such that $s \triangleright \rho$ and $t \triangleright \rho$.*

Thus $s =_\triangleright t$ if and only if s and t have the same meaning.

We have defined two equivalence relations on programs. The following theorems relate them.

Theorem 2 *If $s =_\triangleright t$, then $s \sim t$.*

Proof. By coinduction on $\leq$. It suffices to show that $s =_\triangleright t$ implies $s[=_\triangleright]t$. Use the following facts. For values v and v', $v =_\triangleright v'$ implies v and v' have the same outermost operator. For values $\theta(\mathbf{u})$ and $\theta(\mathbf{u}')$, $\theta(\mathbf{u}) =_\triangleright \theta(\mathbf{u}')$ implies $\theta(\mathbf{u})\ [=_\triangleright]\ \theta(\mathbf{u}')$.

Theorem 3 *If s denotes and $s \leq t$, then $s =_\triangleright t$.*

Proof. By induction on the derivation of $s \triangleright \sigma$.

Theorem 4 *If $s \sim t$, then either s and t both do not denote, or $s =_\triangleright t$.*

Proof. This follows directly from Lemma 3 and the symmetry of $\sim$.

3.5 Polymorphism

It is easy to add a universe U, or a cumulative hierarchy of universes, to the theory, if one postulates the existence of a sufficient number of inaccessible cardinals. Universes (probably one is sufficient) could play a role in our logic similar to the one they play in Martin-Löf's type theory [13]. For example, with one universe U, we could write a polymorphic list-append procedure and give it a type like

$$\Pi\ A \in U\ .\ A\ list \to A\ list \to A\ list.$$

[4] In [12], $\widehat{\leq}$ is named $\leq^*$

We could also use sigma-types to represent abstract data types.

Our delaying of the evaluation of ε' in the interpreter, described below, is essential if we are to obtain an executable programming language with explicit polymorphism of the kind we are suggesting. For example, an append function with the type above takes a type argument; the interpreter should not have to evaluate this argument.

4 The Interpreter

Consider the computable fragment of P sketched in the introduction. The interpreter for this fragment manipulates representatives of equivalence classes rather than equivalence classes *per se*. In other words, the representation of an equivalence class in the interpreter is a tagged member of the equivalence class. This is achieved by making $[\cdot]$ canonical.[5] This requires a modification to the evaluation rule for *qap*; the interpreter simply computes the specified function on the given member of the equivalence class. One can view this modification to the evaluator as delaying the evaluation of terms with a certain outer constructor (here, $[\cdot]$) until the value of such a term is needed by another rule (here, the *qap* rule), then computing only as much information as is needed to determine the result of that rule (here, it suffices to produce a member of the equivalence class).

For technical convenience, we define our interpreter in two stages. First, we define a new evaluation relation $\bar{\Downarrow}$ for the full language P. The new evaluator delays computation of $[\cdot]$ and ε', and omits the typechecking premise in the rule for application. The desired interpreter is easily derived from this set of rules.

The evaluation relation $\bar{\Downarrow}$ is inductively defined by the evaluation rules for $\Downarrow$ given above, with the following modifications:

1. Replace rule (6) for application with

$$\frac{f\bar{\Downarrow}\lambda x : A.\ b \qquad b[t/x]\bar{\Downarrow}v}{f(t)\bar{\Downarrow}v} \tag{16}$$

2. Replace rule (8) for equivalence classes with

$$\frac{t\bar{\Downarrow}v}{[t]_A\bar{\Downarrow}[v]_A} \tag{17}$$

3. Replace rule (9) for *qap* with the rules

$$\frac{t\bar{\Downarrow}[s]_A \qquad f(s)\bar{\Downarrow}v}{qap(f;t)\bar{\Downarrow}v} \qquad \frac{t\bar{\Downarrow}\xi \qquad \forall\rho\in|\xi|.\ f(\widehat{\rho})\triangleright\tau}{qap(f;t)\bar{\Downarrow}\widehat{\tau}} \tag{18}$$

4. Replace rule (12) for ε' with the rule

$$\frac{}{\varepsilon'(x.t)\bar{\Downarrow}\varepsilon'(x.t)} \tag{19}$$

[5] The theory works equally well whether it is eager or lazy.

Note that in the rules for $\bar{\Downarrow}$, $\triangleright$ still denotes the original "meaning" relation.

Let v and $\bar{v}$ be the results of evaluating and interpreting, respectively, a term s. We want to prove, by induction on the derivation of $s \Downarrow v$, that $v \sim \bar{v}$. In order to obtain a sufficiently strong induction hypothesis, we need to show a stronger relation, denoted $\sqsubseteq$, between v and $\bar{v}$. Informally, $s \sqsubseteq \bar{s}$ if $\bar{s}$ is obtained from s by replacing some normal subterms of s with terms of L that have the same meanings (and a few technical properties).

Definition 5 *Formally, $\sqsubseteq$ is the least binary relation on ${T_0}^2$ satisfying $s \sqsubseteq \bar{s}$ if there are $a_1, \ldots, a_n$ and $\rho_1, \ldots, \rho_n$ such that*

$$\bar{s} = s[a_1/\widehat{\rho_1}, \ldots, a_n/\widehat{\rho_n}]$$

and for all i, $1 \leq i \leq n$, $a_i \triangleright \rho_i$ and there exists a_i' such that

- $a_i \bar{\Downarrow} a_i'$,
- $a_i' \triangleright \rho_i$ *and*
- *if a_i' is $[t]_A$ and $\rho_i = (ec, \tau)$ then there is a $\sigma \in \tau$ such that $\widehat{\sigma} \sqsubseteq t$.*

We say that a term v is a $\overline{\text{value}}$ if $v \bar{\Downarrow} v$.

Lemma 5 *$\sqsubseteq$ is a precongruence, and if $\theta(\mathbf{u}) \sqsubseteq \theta(\mathbf{v})$ then $\mathbf{u} \sqsubseteq \mathbf{v}$.*

Lemma 6 *1. If $s =_\triangleright \bar{s}$ and $t[s/x] \triangleright \tau$, then $t[\bar{s}/x] \triangleright \tau$.*
2. If $s \triangleright \sigma$ and $s \sqsubseteq \bar{s}$, then $\bar{s} \triangleright \sigma$.
3. If s is closed and $s \sqsubseteq \bar{s}$, then $s \sim \bar{s}$.

Proof. For the first part, $s =_\triangleright \bar{s}$, so by Theorem 2, $s \sim \bar{s}$, so by Theorem 1, $t[s/x] \sim t[\bar{s}/x]$, so by Theorem 4, $t[s/x] =_\triangleright t[\bar{s}/x]$. The second part is immediate from the first and from the definition of $\sqsubseteq$. Finally, let $\bar{s} = s[\bar{a}_1/\widehat{\rho_1}, \ldots, \bar{a}_n/\widehat{\rho_n}]$. $a_i =_\triangleright \widehat{\rho_i}$, so by Theorem 2, $a_i \sim \widehat{\rho_i}$, so by Theorem 1, $s \sim \bar{s}$.

Lemma 7 *1. If v is a value, then $outer(v)$ is canonical.*
2. If v is a $\overline{\text{value}}$, then $outer(v)$ is canonical or $[\cdot]$.
3. If v is canonical and $v \Downarrow v'$, then $outer(v) = outer(v')$.
4. If v is a value and denotes an equivalence class, then v is of the form ξ.

Proof. Parts 1 and 2 are proved by induction on the derivation of $s \Downarrow v$ and $s \bar{\Downarrow} v$, respectively. Part 3 follows immediately from the restrictions on evaluation rules for canonical operators. For Part 4, suppose $v \triangleright \xi$. From the form of the rule for deriving $v \triangleright \xi$, it must be that $v \Downarrow \xi$. v is a value, so by Part 1, $outer(v)$ is canonical. By Part 3, $outer(v) = outer(\xi) = \xi$.

Lemma 8 *If v and $\bar{v}$ are canonical and $v =_\triangleright \bar{v}$, then $outer(v) = outer(\bar{v})$.*

Proof. By Lemma 3, there exist v' and $\bar{v}'$ such that $v \Downarrow v'$ and $\bar{v} \Downarrow \bar{v}'$. Note that $v' =_\triangleright \bar{v}'$. By the third part of Lemma 7, $outer(v) = outer(v')$ and $outer(\bar{v}) = outer(\bar{v}')$. By inspection of the rules defining $\triangleright$, if v is a denoting value, then $outer(v)$ is uniquely determined by its value. Since $v' = \bar{v}'$, $outer(v') = outer(\bar{v}')$. By transitivity, $outer(v) = outer(\bar{v})$.

Lemma 9 *If v is a value, $\bar{v}$ is a $\overline{value}$, $v \sqsubseteq \bar{v}$, and v is not of the form ξ, then $outer(v) = outer(\bar{v})$.*

Proof. Case 1: v is completely replaced to obtain $\bar{v}$. v must be a normal term $\widehat{\sigma}$. v is not of the form ξ, so by Lemma 7(4), v does not denote an equivalence class. By Lemma 6(2), $\bar{v} \rhd \sigma$. $outer(\bar{v})$ is not $[\cdot]$, since if it were, $\bar{v}$ could denote only an equivalence class. By Lemma 7(2), $outer(\bar{v})$ is canonical. By Lemma 8, $outer(v) = outer(\bar{v})$. Case 2, where v is not completely replaced, is trivial.

The main result of this section is:

Theorem 5 *If $s \Downarrow v$ and $s \sqsubseteq \bar{s}$, then $\bar{s} \bar{\Downarrow} \bar{v}$ and $v \sqsubseteq \bar{v}$.*

Proof. By induction on the derivation of $s \Downarrow v$. We describe two of the more interesting cases for the primitive rules (the other cases are similar but simpler) and the case for the general rule format. Suppose the last rule used is rule (9) for *qap*. The (end of the) derivation must be of the form

$$\frac{t \rhd \xi \qquad \forall \sigma \in |\xi|.\ f(\widehat{\sigma}) \rhd \rho}{qap(f;t) \Downarrow \widehat{\rho}}$$

By Lemma 3, there are subderivations showing $t \Downarrow \xi$ and for all $\sigma \in |\xi|$, $f(\widehat{\sigma}) \Downarrow v_\sigma$, where $v_\sigma \rhd \rho$. s is non-canonical hence non-normal, so it is not completely replaced, so $\bar{s}$ is of the form $\bar{s} \equiv qap(\bar{f};\bar{t})$, with $f \sqsubseteq \bar{f}$ and $t \sqsubseteq \bar{t}$. By the induction hypothesis applied to $t \bar{\Downarrow} \xi$, $\bar{t} \Downarrow \bar{e}$ and $\xi \sqsubseteq \bar{e}$. By Lemma 7(2, 4), $outer(\bar{e})$ is ξ or $[\cdot]$. Suppose $\bar{e} \equiv \xi$. For all $\sigma \in |\xi|$, $f(\widehat{\sigma}) \sqsubseteq \bar{f}(\widehat{\sigma})$, so by Lemma 6(2), $\bar{f}(\widehat{\sigma}) \rhd \rho$. Thus, by the second rule in (18), $qap(\bar{f};\bar{t}) \bar{\Downarrow} \widehat{\rho}$.

Suppose $\bar{e} \equiv [a]_A$. $\xi \sqsubseteq [a]_A$, so $[a]_A \rhd \xi$ and there exists $\sigma \in |\xi|$ such that $\widehat{\sigma} \sqsubseteq a$. $f(\widehat{\sigma}) \sqsubseteq \bar{f}(\bar{a})$, so by the induction hypothesis on $f(\widehat{\sigma}) \bar{\Downarrow} v_\sigma$, $\bar{f}(\bar{a}) \Downarrow \bar{v}$ and $v_\sigma \sqsubseteq \bar{v}$. By the first rule in (18), $\underline{qap}(\bar{f};\bar{t}) \bar{\Downarrow} \bar{v}$, so it suffices to show $\widehat{\rho} \sqsubseteq \bar{v}$. By Lemma 6(2), $\bar{v} \rhd \rho$. Since $\bar{v}$ is a $\underline{\text{value}}$, $\bar{v} \bar{\Downarrow} \bar{v}$. If $outer(\bar{v}) \neq [\cdot]$, we are done. Suppose $\bar{v} \equiv [a_1]_{A_1}$ and $\rho = \xi_1$. We need to show there is a $\sigma_1 \in |\xi_1|$ such that $\widehat{\sigma_1} \sqsubseteq a_1$. By Lemma 7(1,4), $v_\sigma \equiv \xi_1$. $\xi_1 \sqsubseteq [a_1]_{A_1}$, so by definition of $\sqsubseteq$, we are done.

Suppose the last rule used in the derivation of $s \Downarrow v$ is rule (8) for $[\cdot]$. s is non-canonical hence non-normal, so it is not completely replaced, so $\bar{s}$ is of the form $\bar{s} \equiv [\bar{t}]_{\bar{A}}$, with $t \sqsubseteq \bar{t}$ and $A \sqsubseteq \bar{A}$. Since $t \rhd \rho$, $t \Downarrow v_\rho$ and $v_\rho \rhd \rho$. By the induction hypothesis on $t \Downarrow v_\rho$, $\bar{t} \Downarrow \bar{v}_\rho$ and $v_\rho \sqsubseteq \bar{v}_\rho$. So by rule 17, $\bar{s} \bar{\Downarrow} [\bar{v}_\rho]_{\bar{A}}$. We need to show $\xi \sqsubseteq [\bar{v}_\rho]_{\bar{A}}$. Since $\bar{\Downarrow}$ is single-valued, it suffices to show that $[\bar{v}_\rho]_{\bar{A}} \rhd \xi$ and that there exists $\rho \in |\xi|$ such that $\widehat{\rho} \sqsubseteq \bar{v}_\rho$. The former holds by Lemma 6.

Suppose that $outer(\bar{v}_\rho) \neq [\cdot]$. Then $\widehat{\rho} \sqsubseteq \bar{v}_\rho$ follows from $\bar{v}_\rho \rhd \rho$ and the fact that $\bar{\Downarrow}$ is single-valued.

Suppose that $outer(\bar{v}_\rho) = [\cdot]$, *i.e.*, $\bar{v}_\rho$ is of the form $[t_1]_{A_1}$. Since $\bar{v}_\rho \rhd \rho$, ρ must be an equivalence class, $\rho = \xi_1$. It suffices to show there exists $\rho_1 \in |\xi_1|$ such that $\rho_1 \sqsubseteq t_1$. $\bar{v}_\rho$ is a value and denotes an equivalence class, so by Lemma 7(4), $\bar{v}_\rho$ is ξ_1, so $\xi_1 \sqsubseteq [t_1]_{A_1}$. By definition of $\sqsubseteq$, we are done.

Finally, we consider the general case. Suppose the last rule used is a substitution instance of rule 3. Let η denote the second-order substitution that was used. Let underlining denote application of η, so, *e.g.*, $\underline{a} = a\eta$. Note that s is $\underline{a}$.

Extend $\sqsubseteq$ to closed second-order substitutions by $\eta \sqsubseteq \bar{\eta}$ if for all $P \in dom(\bar{\eta})$, $\eta(P) \sqsubseteq \bar{\eta}(P)$. Note that if $\eta \sqsubseteq \bar{\eta}$, then $a\eta \sqsubseteq a\bar{\eta}$.

case: $\underline{a}$ is completely replaced to obtain $\bar{s}$. Then $\underline{a}$ is normal, $\underline{a} \equiv \widehat{\sigma}$. By definition of $\sqsubseteq$, $\bar{s} \triangleright \sigma$, $\bar{s}\bar{\Downarrow}\bar{v}$, and so on. It is easy to check that $\widehat{\sigma} \sqsubseteq \bar{s}$ and $\bar{s}\bar{\Downarrow}\bar{v}$ implies $\widehat{\sigma} \sqsubseteq \bar{v}$. Normal terms are values, so $\widehat{\sigma} \Downarrow \widehat{\sigma}$, so $\underline{b} \equiv \widehat{\sigma}$. Thus, $\bar{s}\bar{\Downarrow}\bar{v}$ and $\underline{b} \sqsubseteq \bar{v}$, as desired.

case: $\underline{a}$ is not completely replaced. In this case, we construct a second-order substitution $\bar{\eta}$ such that $\bar{s} = a\bar{\eta}$, $\eta \sqsubseteq \bar{\eta}$, and $\forall i \in I.\ a_i\bar{\eta}\bar{\Downarrow}b_i\bar{\eta}$. Then $\underline{b} \sqsubseteq b\bar{\eta}$, and we conclude by instantiating this rule with $\bar{\eta}$ that $\bar{s}\bar{\Downarrow}b\bar{\eta}$, as desired. We define $\bar{\eta}$ first on metavariables that appear in a, then on metavariables that appear in the b_i.

We construct a substitution $\bar{\eta}^0$ whose domain is the set of metavariables appearing in a. $\underline{a}$ is not completely replaced to obtain $\bar{s}$, so $outer(\bar{s}) = outer(\underline{a}) = outer(a)$. Since a is a simple term schema and $outer(\bar{s}) = outer(a)$, there exists a substitution $\bar{\eta}^0$ such that $\underline{a} = a\bar{\eta}^0$. For all P appearing in a, $\eta(P)$ and $\bar{\eta}^0(P)$ are corresponding subterms of $\underline{a}$ and $\bar{s}$, so by Lemma 5, $\eta(P) \sqsubseteq \bar{\eta}^0(P)$. So $\eta \sqsubseteq \bar{\eta}^0$.

We show by induction on i that for all $i \in I$ there exists $\bar{\eta}_i$ such that $a_i\bar{\eta}_i \Downarrow b_i\bar{\eta}_i$, $\eta \sqsubseteq \bar{\eta}_i$ and $dom(\bar{\eta}_i) = MV(b_i)$, where $MV(t)$ is the set of metavariables appearing in t. Define $\bar{\eta}'_i = \bar{\eta}^0 \cup (\cup_{j<_I i}\bar{\eta}_j)$. By the induction hypothesis and the above result about $\bar{\eta}^0$, $\eta \sqsubseteq \bar{\eta}'_i$, so $\underline{a}_i \sqsubseteq a_i\bar{\eta}'_i$, so by the induction hypothesis applied to $\underline{a}_i \Downarrow \underline{b}_i$, $a_i\bar{\eta}'_i \Downarrow b'_i$ and $\underline{b}_i \sqsubseteq b'_i$.

case: b_i is a metavariable. Define $\bar{\eta}'_i$ to be the substitution that maps b_i to b'_i.

case: b_i is a simple term schema. Claim: $outer(b'_i) = outer(\underline{b}_i)$. Proof:

case: $\underline{b}_i$ is completely replaced to obtain b'_i. By the restrictions on evaluation rules, b_i is not of the form ξ, hence neither is $\underline{b}_i$, so by Lemma 9, $outer(b'_i) = outer(\underline{b}_i)$.

case: $\underline{b}_i$ is not completely replaced to obtain b'_i.

Thus, $outer(b'_i) = outer(\underline{b}_i) = outer(b_i)$, so b'_i is an instance of b_i. Define $\bar{\eta}_i$ to be the substitution obtained by matching b'_i to b_i. For all metavariables P in b_i, $\eta(P)$ and $\bar{\eta}_i(P)$ are corresponding subterms of $\underline{b}_i$ and b'_i, and $\underline{b}_i \sqsubseteq b'_i$, so $\eta(P) \sqsubseteq \bar{\eta}_i(P)$. By this argument and the induction hypothesis, $\eta \sqsubseteq \bar{\eta}_i$.

Finally, define $\bar{\eta} = \bar{\eta}^0 \cup (\cup_{i\in I}\bar{\eta}_i)$. It is easy to see that $\bar{\eta}$ has the required properties.

Corollary 1 *If $\bar{s} \Downarrow v$, then $\bar{s}\bar{\Downarrow}\bar{v}$ and $v \sim \bar{v}$.*

Finally, we want to distinguish an computable fragment of L. Let E be the set of closed terms e of L such that for all operators τ of L, either $\tau \in \{qap, [\cdot]\}$, τ is an operator of L_0, or all occurrences of τ in e are within a subterm of the form $\varepsilon'(x.t)$ or are within A in a subterm of the form $\lambda x : A.\ b$ or $[t]_A$.

The restricted interpreter is obtained from the rules for $\bar{\Downarrow}$ by simply omitting the rules for the omitted constructs, and omitting the second rule for *qap* in (18). Let $\Downarrow'$ be the evaluation relation defined by this new set of rules.

Theorem 6 *For all e in E, if $e \Downarrow v$ then there is a $v' \in E$ such that $e \Downarrow' v'$ and $v \sim v'$.*

Proof. First show that E is closed under $\Downarrow'$, which is easily proven by induction on the definition of $\Downarrow'$. The desired result then follows immediately from Corollary 1.

Since the interpreter ignores type information on λ's and equivalence classes, it is easy to justify optimizations in which this information is removed (erased).

5 A Sequent Calculus

The term language for the logic is L' obtained from L by removing all operators ξ, α and ϕ. A *sequent* has the form $\Gamma \vdash t$, where Γ is a set $\{s_1, \ldots, s_n\}$ of possibly open terms. It is *true* if for all closed (first-order) substitutions η, if for all i, $1 \leq i \leq n$, $s_i\eta \Downarrow T$ (where $s_i\eta$ is the application of the substitution η to s_i), then $t\eta \Downarrow T$. Note the difference between a variable bound by $\forall$ and a free variable: the former is quantified over all set-theoretic objects; the latter is implicitly quantified over all closed terms.

We only sketch a logic for our semantics. Variants of most of the types used in Nuprl can easily be defined here, and the rules used in Nuprl can be made true with minor modifications. For example, Nuprl's dependent function space becomes its classical analogue, generalized cartesian product. If we use the logic given by our analogue of Nuprl's propositions as types, then we will be able to extract a "program" from any proof. Of course, this program may contain uncomputable operators (defined, *e.g.*, using ε), but if one avoids classical reasoning, the program will be computable.

Nuprl's rules for its quotient type can be adapted for reasoning about equivalence classes. Another important set of rules we can adapt from Nuprl are the *direct computation* rules. These allow direct reasoning about symbolic computation in the programming language. They are justified by the fact that $\leq$ is a precongruence.

One general rule which is an improvement on the Nuprl version is the following, which allows free substitution of equals for equals. We present our rules "refinement style", with the conclusion of the rule first, and the premises listed below it.

$$\begin{array}{l} \Gamma \vdash t \\ \quad \Gamma \vdash a = b \\ \quad \Gamma \vdash t[b/a] \end{array} \tag{20}$$

The soundness of this rule follows easily from Theorem 2. The Nuprl analogue of this rule has a third premise requiring that t be appropriate for the given substitution.

The only difficulty in stating the axioms of ZF set theory is dealing with formula parameters. For example, the comprehension axiom becomes a rule with a premise to show that the formula parameter is well formed (*i.e.*, has a meaning):

$$\begin{array}{l} \Gamma \vdash \forall y.\, \exists z.\, \forall x.\, x \in z \Leftrightarrow x \in y \;\&\; \phi(x) \\ \quad \Gamma \vdash \forall x.\, WF(\phi(x)) \end{array} \tag{21}$$

where $WF(t)$ abbreviates $t = t$.

We also include axioms relating normal terms to their set-theoretic codes. These axioms will be needed only to establish some derived rules. Two such derived rules are the following.

$$\begin{array}{l} \vdash \lambda x : A.\ b = \lambda x : A'.\ b' \\ \quad \vdash A = A' \\ \quad x \in A \vdash b = b' \end{array} \tag{22}$$

$$\begin{array}{l} \vdash f(a) = t \\ \quad \vdash f = \lambda x : A.\ b \\ \quad \vdash a \in A \\ \quad a \in A \vdash t = b[a/x] \end{array} \tag{23}$$

The only subtle rule is the rule for typing recursively-defined functions. Suppose the programming language P contains the untyped lazy λ-calculus. We use λ' to denote the untyped lambda operator. We can define a fixed-pointed combinator Y in the usual way. Let WellFounded$(A; <)$ be a predicate that holds exactly when A is well-ordered by $<$.

$$\begin{array}{l} \vdash Y(\lambda' f.\lambda x : A.\ b) \in A \to B \\ \quad \vdash \text{WellFounded}(A; <) \\ \quad x \in A, f \in \{y : A \mid y < x\} \to B \vdash b[(\lambda z : A.\ f(z))/f] \in B \end{array} \tag{24}$$

It is easy to show by induction on A that this rule is sound. The following fact is needed. If $\sigma' \subseteq \sigma$, then

$$\lambda z : \widehat{\sigma}.\ (\lambda y : \widehat{\sigma'}.\ b[y/z])z \ \leq \ \lambda z : \widehat{\sigma}.\ b.$$

6 Future Work

We are considering implementation of a theorem-prover, based on Nuprl, and a compiler (to Lisp) for this theory. We plan to develop a version of the theory containing unbounded λ-abstraction, building on the ideas in [12], which gives a classical model of a type theory of Martin-Löf. This will allow more polymorphism, but will likely place strong restrictions on the permissible kinds of set-theoretic principles for constructing sets.

Acknowledgments

The second author would like to thank Professor Constable and Stuart Allen for many illuminating discussions of type theory and program verification. The second author is supported by an IBM Graduate Fellowship.

References

1. S. Abramsky. The lazy lambda calculus. Proceedings of the Institute of Declarative Programming, August 1987.
2. M. J. Beeson. Towards a computation system based on set theory. *Theoretical Computer Science*, 60:297–340, 1988.
3. E. Bishop. *Foundations of Constructive Analysis*. McGraw-Hill, New York, 1967.
4. V. Breazu-Tannen and R. Subrahmanyam. Logical and computational aspects of programming with sets/bags/lists. In *Automata, Languages and Programming: 18^{th} International Colloquium*, Lecture Notes in Computer Science, pages 60–75. Springer-Verlag, 1991.
5. V. Breazu-Tannen and R. Subrahmanyam. On extending computational adequacy by data abstraction. In *Proc. ACM Symposium on Lisp and Functional Programming*, pages 161–169. ACM Press, 1992.
6. R. L. Constable, et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
7. T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.
8. S. Feferman. A language and axioms for explicit mathematics. In Dold, A. and B. Eckmann, editor, *Algebra and Logic*, volume 450 of *Lecture Notes in Mathematics*, pages 87–139. Springer-Verlag, 1975.
9. M. Gordon. A proof generating system for higher-order logic. In *Proceedings of the Hardware Verification Workshop*, 1989.
10. K. Grue. Map theory. *Theoretical Computer Science*, 102:1–133, 1992.
11. D. J. Howe. Equality in lazy computation systems. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 198–203. IEEE Computer Society, June 1989.
12. D. J. Howe. On computational open-endedness in Martin-Löf's type theory. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science*, pages 162–172. IEEE Computer Society, 1991.
13. P. Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175. North Holland, 1982. .
14. D. A. McAllester. *Ontic: A Knowledge Representation System for Mathematics*. MIT Press, 1989.
15. L. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–385. Academic Press, 1990.
16. L. C. Paulson. Set theory for verification: I. from foundations to functions. Technical report, University of Cambridge, 1993.
17. J. C. Reynolds. Polymorphism is not set-theoretic. In G. Kahn, D. B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types: International Symposium*, volume 173 of *Lecture Notes in Computer Science*, pages 145–156. Springer-Verlag, 1984.
18. J. Spivey. *The Z Notation*. Prentice Hall, 1989.

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